

# Contents

[Advanced WPF Areas](#)

[WPF Architecture](#)

[XAML in WPF](#)

[XAML Overview \(WPF\)](#)

[XAML Syntax In Detail](#)

[Code-Behind and XAML in WPF](#)

[XAML and Custom Classes for WPF](#)

[Markup Extensions and WPF XAML](#)

[XAML Namespaces and Namespace Mapping for WPF XAML](#)

[WPF XAML Namescopes](#)

[Inline Styles and Templates](#)

[TypeConverters and XAML](#)

[WPF XAML Extensions](#)

[Binding Markup Extension](#)

[ColorConvertedBitmap Markup Extension](#)

[ComponentResourceKey Markup Extension](#)

[DateTime XAML Syntax](#)

[DynamicResource Markup Extension](#)

[RelativeSource MarkupExtension](#)

[StaticResource Markup Extension](#)

[TemplateBinding Markup Extension](#)

[ThemeDictionary Markup Extension](#)

[PropertyPath XAML Syntax](#)

[PresentationOptions:Freeze Attribute](#)

[Markup Compatibility \(mc:\) Language Features](#)

[mc:Ignorable Attribute](#)

[mc:ProcessContent Attribute](#)

[Base Element Classes](#)

[Base Elements Overview](#)

[Freezable Objects Overview](#)

[Alignment, Margins, and Padding Overview](#)

[How-to Topics](#)

[Make a UIElement Transparent or Semi-Transparent](#)

[Animate the Size of a FrameworkElement](#)

[Determine Whether a Freezable Is Frozen](#)

[Handle a Loaded Event](#)

[Set Margins of Elements and Controls](#)

[Make a Freezable Read-Only](#)

[Obtain a Writable Copy of a Read-Only Freezable](#)

[Flip a UIElement Horizontally or Vertically](#)

[Use a ThicknessConverter Object](#)

[Handle the ContextMenuOpening Event](#)

[Element Tree and Serialization](#)

[Trees in WPF](#)

[Serialization Limitations of XamlWriter.Save](#)

[Initialization for Object Elements Not in an Object Tree](#)

[How-to Topics](#)

[Find an Element by Its Name](#)

[Override the Logical Tree](#)

[WPF Property System](#)

[Dependency Properties Overview](#)

[Attached Properties Overview](#)

[Custom Dependency Properties](#)

[Dependency Property Metadata](#)

[Dependency Property Callbacks and Validation](#)

[Framework Property Metadata](#)

[Dependency Property Value Precedence](#)

[Read-Only Dependency Properties](#)

[Property Value Inheritance](#)

[Dependency Property Security](#)

[Safe Constructor Patterns for DependencyObjects](#)

[Collection-Type Dependency Properties](#)

[XAML Loading and Dependency Properties](#)

[How-to Topics](#)

[Implement a Dependency Property](#)

[Add an Owner Type for a Dependency Property](#)

[Register an Attached Property](#)

[Override Metadata for a Dependency Property](#)

[Events](#)

[Routed Events Overview](#)

[Attached Events Overview](#)

[Object Lifetime Events](#)

[Marking Routed Events as Handled, and Class Handling](#)

[Preview Events](#)

[Property Change Events](#)

[Visual Basic and WPF Event Handling](#)

[Weak Event Patterns](#)

[How-to Topics](#)

[Add an Event Handler Using Code](#)

[Handle a Routed Event](#)

[Create a Custom Routed Event](#)

[Find the Source Element in an Event Handler](#)

[Add Class Handling for a Routed Event](#)

[Input](#)

[Input Overview](#)

[Commanding Overview](#)

[Focus Overview](#)

[Styling for Focus in Controls, and FocusVisualStyle](#)

[Walkthrough: Creating Your First Touch Application](#)

[How-to Topics](#)

[Enable a Command](#)

[Change the Cursor Type](#)

[Change the Color of an Element Using Focus Events](#)

- [Apply a FocusVisualStyle to a Control](#)
- [Detect When the Enter Key is Pressed](#)
- [Create a Rollover Effect Using Events](#)
- [Make an Object Follow the Mouse Pointer](#)
- [Create a RoutedCommand](#)
- [Implement ICommandSource](#)
- [Hook Up a Command to a Control with No Command Support](#)
- [Hook Up a Command to a Control with Command Support](#)
- [Digital Ink](#)
  - [Overviews](#)
    - [Getting Started with Ink](#)
    - [Collecting Ink](#)
    - [Handwriting Recognition](#)
    - [Storing Ink](#)
  - [The Ink Object Model: Windows Forms and COM versus WPF](#)
  - [Advanced Ink Handling](#)
    - [Custom Rendering Ink](#)
    - [Intercepting Input from the Stylus](#)
    - [Creating an Ink Input Control](#)
    - [The Ink Threading Model](#)
- [How-to Topics](#)
  - [Select Ink from a Custom Control](#)
  - [Add Custom Data to Ink Data](#)
  - [Erase Ink on a Custom Control](#)
  - [Recognize Application Gestures](#)
  - [Drag and Drop Ink](#)
  - [Data Bind to an InkCanvas](#)
  - [Analyze Ink with Analysis Hints](#)
  - [Rotate Ink](#)
  - [Disable the RealTimeStylus for WPF Applications](#)
- [Drag and Drop](#)
  - [Drag and Drop Overview](#)

## Data and Data Objects

[Walkthrough: Enabling Drag and Drop on a User Control](#)

### How-to Topics

[Open a File That is Dropped on a RichTextBox Control](#)

[Create a Data Object](#)

[Determine if a Data Format is Present in a Data Object](#)

[List the Data Formats in a Data Object](#)

[Retrieve Data in a Particular Data Format](#)

[Store Multiple Data Formats in a Data Object](#)

## Resources

[XAML Resources](#)

[Resources and Code](#)

[Merged Resource Dictionaries](#)

### How-to Topics

[Define and Reference a Resource](#)

[Use Application Resources](#)

[Use SystemFonts](#)

[Use System Fonts Keys](#)

[Use SystemParameters](#)

[Use System Parameters Keys](#)

## Documents

[Documents in WPF](#)

[Document Serialization and Storage](#)

### Annotations

[Annotations Overview](#)

[Annotations Schema](#)

### Flow Content

[Flow Document Overview](#)

[TextElement Content Model Overview](#)

[Table Overview](#)

### How-to Topics

[Adjust Spacing Between Paragraphs](#)

[Build a Table Programmatically](#)

[Change the FlowDirection of Content Programmatically](#)

[Change the TextWrapping Property Programmatically](#)

[Define a Table with XAML](#)

[Alter the Typography of Text](#)

[Enable Text Trimming](#)

[Insert an Element Into Text Programmatically](#)

[Manipulate Flow Content Elements through the Blocks Property](#)

[Manipulate Flow Content Elements through the Inlines Property](#)

[Manipulate a FlowDocument through the Blocks Property](#)

[Manipulate a Table's Columns through the Columns Property](#)

[Manipulate a Table's Row Groups through the RowGroups Property](#)

[Use Flow Content Elements](#)

[Use FlowDocument Column-Separating Attributes](#)

## Typography

[Typography in WPF](#)

[ClearType Overview](#)

[ClearType Registry Settings](#)

[Drawing Formatted Text](#)

[Advanced Text Formatting](#)

## Fonts

[OpenType Font Features](#)

[Packaging Fonts with Applications](#)

[Sample OpenType Font Pack](#)

## How-to Topics

[Enumerate System Fonts](#)

[Use the FontSizeConverter Class](#)

## Glyphs

[Introduction to the GlyphRun Object and Glyphs Element](#)

[How to: Draw Text Using Glyphs](#)

## How-to Topics

[Create a Text Decoration](#)

[Specify Whether a Hyperlink is Underlined](#)

[Apply Transforms to Text](#)

[Apply Animations to Text](#)

[Create Text with a Shadow](#)

[Create Outlined Text](#)

[Draw Text to a Control's Background](#)

[Draw Text to a Visual](#)

[Use Special Characters in XAML](#)

## [Printing and Print System Management](#)

[Printing Overview](#)

[How-to Topics](#)

[Invoke a Print Dialog](#)

[Clone a Printer](#)

[Diagnose Problematic Print Job](#)

[Discover Whether a Print Job Can Be Printed At This Time of Day](#)

[Enumerate a Subset of Print Queues](#)

[Get Print System Object Properties Without Reflection](#)

[Programmatically Print XPS Files](#)

[Remotely Survey the Status of Printers](#)

[Validate and Merge PrintTickets](#)

## [Globalization and Localization](#)

[WPF Globalization and Localization Overview](#)

[Globalization for WPF](#)

[Use Automatic Layout Overview](#)

[Localization Attributes and Comments](#)

[Bidirectional Features in WPF Overview](#)

[How-to Topics](#)

[Localize an Application](#)

[Use Automatic Layout to Create a Button](#)

[Use a Grid for Automatic Layout](#)

[Use a ResourceDictionary to Manage Localizable String Resources](#)

[Use Resources in Localizable Applications](#)

## Layout

### Migration and Interoperability

#### WPF and Windows Forms Interoperation

Windows Forms and WPF Interoperability Input Architecture

Layout Considerations for the WindowsFormsHost Element

Windows Forms Controls and Equivalent WPF Controls

Windows Forms and WPF Property Mapping

Troubleshooting Hybrid Applications

Walkthrough: Hosting a Windows Forms Control in WPF

Walkthrough: Hosting a Windows Forms Control in WPF by Using XAML

Walkthrough: Hosting a Windows Forms Composite Control in WPF

Walkthrough: Hosting an ActiveX Control in WPF

How to: Enable Visual Styles in a Hybrid Application

Walkthrough: Arranging Windows Forms Controls in WPF

Walkthrough: Binding to Data in Hybrid Applications

Walkthrough: Hosting a 3-D WPF Composite Control in Windows Forms

Walkthrough: Hosting a WPF Composite Control in Windows Forms

Walkthrough: Mapping Properties Using the ElementHost Control

Walkthrough: Mapping Properties Using the WindowsFormsHost Element

Walkthrough: Localizing a Hybrid Application

#### WPF and Win32 Interoperation

Technology Regions Overview

Sharing Message Loops Between Win32 and WPF

Hosting Win32 Content in WPF

Walkthrough: Hosting a Win32 Control in WPF

Walkthrough: Hosting WPF Content in Win32

Walkthrough: Hosting a WPF Clock in Win32

#### WPF and Direct3D9 Interoperation

Performance Considerations for Direct3D9 and WPF Interoperability

Walkthrough: Creating Direct3D9 Content for Hosting in WPF

Walkthrough: Hosting Direct3D9 Content in WPF

## Performance

### Graphics Rendering Tiers

## Optimizing WPF Application Performance

Planning for Application Performance

Taking Advantage of Hardware

Layout and Design

2D Graphics and Imaging

Object Behavior

Application Resources

Text

Data Binding

Controls

Other Performance Recommendations

Application Startup Time

## Walkthrough: Caching Application Data in a WPF Application

Threading Model

WPF Unmanaged API Reference

Activate Function

CreateIDispatchSTAForwarder Function

Deactivate Function

ForwardTranslateAccelerator Function

LoadFromHistory Function

ProcessUnhandledException Function

SaveToHistory Function

SetFakeActiveWindow Function

# Advanced (Windows Presentation Foundation)

8/1/2019 • 2 minutes to read • [Edit Online](#)

This section describes some of the advanced areas in WPF.

## In This Section

[WPF Architecture](#)

[XAML in WPF](#)

[Base Element Classes](#)

[Element Tree and Serialization](#)

[WPF Property System](#)

[Events in WPF](#)

[Input](#)

[Drag and Drop](#)

[Resources](#)

[Documents](#)

[Globalization and Localization](#)

[Layout](#)

[Migration and Interoperability](#)

[Performance](#)

[Threading Model](#)

[Unmanaged WPF API Reference](#)

# WPF Architecture

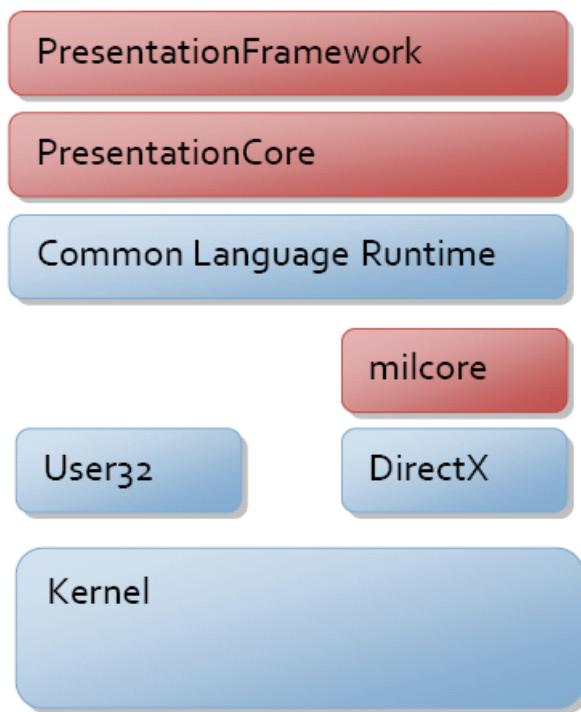
11/7/2019 • 15 minutes to read • [Edit Online](#)

This topic provides a guided tour of the Windows Presentation Foundation (WPF) class hierarchy. It covers most of the major subsystems of WPF, and describes how they interact. It also details some of the choices made by the architects of WPF.

## System.Object

The primary WPF programming model is exposed through managed code. Early in the design phase of WPF there were a number of debates about where the line should be drawn between the managed components of the system and the unmanaged ones. The CLR provides a number of features that make development more productive and robust (including memory management, error handling, common type system, etc.) but they come at a cost.

The major components of WPF are illustrated in the figure below. The red sections of the diagram (PresentationFramework, PresentationCore, and milcore) are the major code portions of WPF. Of these, only one is an unmanaged component – milcore. Milcore is written in unmanaged code in order to enable tight integration with DirectX. All display in WPF is done through the DirectX engine, allowing for efficient hardware and software rendering. WPF also required fine control over memory and execution. The composition engine in milcore is extremely performance sensitive, and required giving up many advantages of the CLR to gain performance.



Communication between the managed and unmanaged portions of WPF is discussed later in this topic. The remainder of the managed programming model is described below.

## System.Threading.DispatcherObject

Most objects in WPF derive from [DispatcherObject](#), which provides the basic constructs for dealing with concurrency and threading. WPF is based on a messaging system implemented by the dispatcher. This works much like the familiar Win32 message pump; in fact, the WPF dispatcher uses User32 messages for performing

cross thread calls.

There are really two core concepts to understand when discussing concurrency in WPF – the dispatcher and thread affinity.

During the design phase of WPF, the goal was to move to a single thread of execution, but a non-thread "affinitized" model. Thread affinity happens when a component uses the identity of the executing thread to store some type of state. The most common form of this is to use the thread local store (TLS) to store state. Thread affinity requires that each logical thread of execution be owned by only one physical thread in the operating system, which can become memory intensive. In the end, WPF's threading model was kept in sync with the existing User32 threading model of single threaded execution with thread affinity. The primary reason for this was interoperability – systems like OLE 2.0, the clipboard, and Internet Explorer all require single thread affinity (STA) execution.

Given that you have objects with STA threading, you need a way to communicate between threads, and validate that you are on the correct thread. Herein lies the role of the dispatcher. The dispatcher is a basic message dispatching system, with multiple prioritized queues. Examples of messages include raw input notifications (mouse moved), framework functions (layout), or user commands (execute this method). By deriving from [DispatcherObject](#), you create a CLR object that has STA behavior, and will be given a pointer to a dispatcher at creation time.

## System.Windows.DependencyObject

One of the primary architectural philosophies used in building WPF was a preference for properties over methods or events. Properties are declarative and allow you to more easily specify intent instead of action. This also supported a model driven, or data driven, system for displaying user interface content. This philosophy had the intended effect of creating more properties that you could bind to, in order to better control the behavior of an application.

In order to have more of the system driven by properties, a richer property system than what the CLR provides was needed. A simple example of this richness is change notifications. In order to enable two way binding, you need both sides of the bind to support change notification. In order to have behavior tied to property values, you need to be notified when the property value changes. The Microsoft .NET Framework has an interface, [INotifyPropertyChanged](#), which allows an object to publish change notifications, however it is optional.

WPF provides a richer property system, derived from the [DependencyObject](#) type. The property system is truly a "dependency" property system in that it tracks dependencies between property expressions and automatically revalidates property values when dependencies change. For example, if you have a property that inherits (like [FontSize](#)), the system is automatically updated if the property changes on a parent of an element that inherits the value.

The foundation of the WPF property system is the concept of a property expression. In this first release of WPF, the property expression system is closed, and the expressions are all provided as part of the framework. Expressions are why the property system doesn't have data binding, styling, or inheritance hard coded, but rather provided by later layers within the framework.

The property system also provides for sparse storage of property values. Because objects can have dozens (if not hundreds) of properties, and most of the values are in their default state (inherited, set by styles, etc.), not every instance of an object needs to have the full weight of every property defined on it.

The final new feature of the property system is the notion of attached properties. WPF elements are built on the principle of composition and component reuse. It is often the case that some containing element (like a [Grid](#) layout element) needs additional data on child elements to control its behavior (like the Row/Column information). Instead of associating all of these properties with every element, any object is allowed to provide property definitions for any other object. This is similar to the "expando" features of JavaScript.

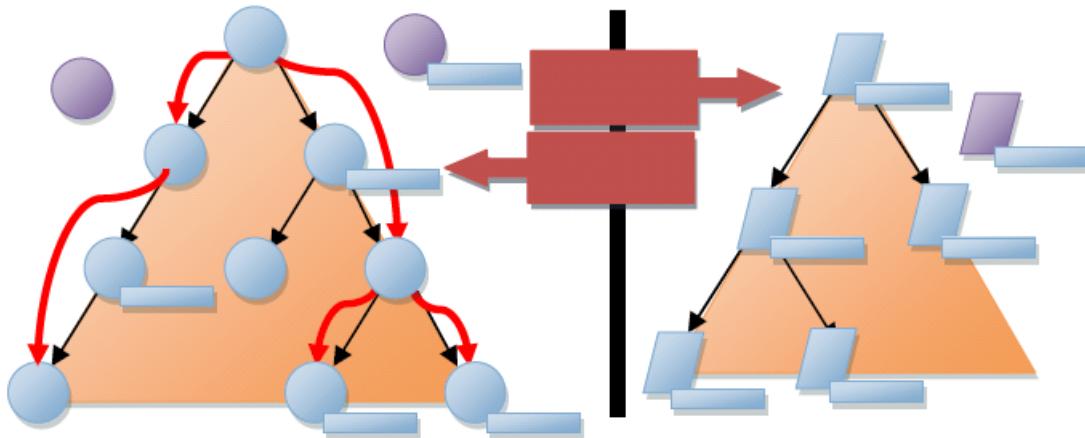
## System.Windows.Media.Visual

With a system defined, the next step is getting pixels drawn to the screen. The [Visual](#) class provides for building a tree of visual objects, each optionally containing drawing instructions and metadata about how to render those instructions (clipping, transformation, etc.). [Visual](#) is designed to be extremely lightweight and flexible, so most of the features have no public API exposure and rely heavily on protected callback functions.

[Visual](#) is really the entry point to the WPF composition system. [Visual](#) is the point of connection between these two subsystems, the managed API and the unmanaged milcore.

WPF displays data by traversing the unmanaged data structures managed by the milcore. These structures, called composition nodes, represent a hierarchical display tree with rendering instructions at each node. This tree, illustrated on the right hand side of the figure below, is only accessible through a messaging protocol.

When programming WPF, you create [Visual](#) elements, and derived types, which internally communicate to the composition tree through this messaging protocol. Each [Visual](#) in WPF may create one, none, or several composition nodes.



There is a very important architectural detail to notice here – the entire tree of visuals and drawing instructions is cached. In graphics terms, WPF uses a retained rendering system. This enables the system to repaint at high refresh rates without the composition system blocking on callbacks to user code. This helps prevent the appearance of an unresponsive application.

Another important detail that isn't really noticeable in the diagram is how the system actually performs composition.

In User32 and GDI, the system works on an immediate mode clipping system. When a component needs to be rendered, the system establishes a clipping bounds outside of which the component isn't allowed to touch the pixels, and then the component is asked to paint pixels in that box. This system works very well in memory constrained systems because when something changes you only have to touch the affected component – no two components ever contribute to the color of a single pixel.

WPF uses a "painter's algorithm" painting model. This means that instead of clipping each component, each component is asked to render from the back to the front of the display. This allows each component to paint over the previous component's display. The advantage of this model is that you can have complex, partially transparent shapes. With today's modern graphics hardware, this model is relatively fast (which wasn't the case when User32/GDI were created).

As mentioned previously, a core philosophy of WPF is to move to a more declarative, "property centric" model of programming. In the visual system, this shows up in a couple of interesting places.

First, if you think about the retained mode graphic system, this is really moving away from an imperative `DrawLine`/`DrawLine` type model, to a data oriented model – new `Line()`/new `Line()`. This move to data driven

rendering allows complex operations on the drawing instructions to be expressed using properties. The types deriving from [Drawing](#) are effectively the object model for rendering.

Second, if you evaluate the animation system, you'll see that it is almost completely declarative. Instead of requiring a developer to compute the next location, or next color, you can express animations as a set of properties on an animation object. These animations can then express the intent of the developer or designer (move this button from here to there in 5 seconds), and the system can determine the most efficient way to accomplish that.

## System.Windows.UIElement

[UIElement](#) defines core subsystems including Layout, Input, and Events.

Layout is a core concept in WPF. In many systems there is either a fixed set of layout models (HTML supports three models for layout; flow, absolute, and tables) or no model for layout (User32 really only supports absolute positioning). WPF started with the assumption that developers and designers wanted a flexible, extensible layout model, which could be driven by property values rather than imperative logic. At the [UIElement](#) level, the basic contract for layout is introduced – a two phase model with [Measure](#) and [Arrange](#) passes.

[Measure](#) allows a component to determine how much size it would like to take. This is a separate phase from [Arrange](#) because there are many situations where a parent element will ask a child to measure several times to determine its optimal position and size. The fact that parent elements ask child elements to measure demonstrates another key philosophy of WPF – size to content. All controls in WPF support the ability to size to the natural size of their content. This makes localization much easier, and allows for dynamic layout of elements as things resize. The [Arrange](#) phase allows a parent to position and determine the final size of each child.

A lot of time is often spent talking about the output side of WPF – [Visual](#) and related objects. However there is a tremendous amount of innovation on the input side as well. Probably the most fundamental change in the input model for WPF is the consistent model by which input events are routed through the system.

Input originates as a signal on a kernel mode device driver and gets routed to the correct process and thread through an intricate process involving the Windows kernel and User32. Once the User32 message corresponding to the input is routed to WPF, it is converted into a WPF raw input message and sent to the dispatcher. WPF allows for raw input events to be converted to multiple actual events, enabling features like "MouseEnter" to be implemented at a low level of the system with guaranteed delivery.

Each input event is converted to at least two events – a "preview" event and the actual event. All events in WPF have a notion of routing through the element tree. Events are said to "bubble" if they traverse from a target up the tree to the root, and are said to "tunnel" if they start at the root and traverse down to a target. Input preview events tunnel, enabling any element in the tree an opportunity to filter or take action on the event. The regular (non-preview) events then bubble from the target up to the root.

This split between the tunnel and bubble phase makes implementation of features like keyboard accelerators work in a consistent fashion in a composite world. In User32 you would implement keyboard accelerators by having a single global table containing all the accelerators you wanted to support (Ctrl+N mapping to "New"). In the dispatcher for your application you would call **TranslateAccelerator** which would sniff the input messages in User32 and determine if any matched a registered accelerator. In WPF this wouldn't work because the system is fully "composable" – any element can handle and use any keyboard accelerator. Having this two phase model for input allows components to implement their own "TranslateAccelerator".

To take this one step further, [UIElement](#) also introduces the notion of CommandBindings. The WPF command system allows developers to define functionality in terms of a command end point – something that implements  [ICommand](#). Command bindings enable an element to define a mapping between an input gesture (Ctrl+N) and a command (New). Both the input gestures and command definitions are extensible, and can be wired together at usage time. This makes it trivial, for example, to allow an end user to customize the key bindings that they want to use within an application.

To this point in the topic, "core" features of WPF – features implemented in the PresentationCore assembly, have been the focus. When building WPF, a clean separation between foundational pieces (like the contract for layout with **Measure** and **Arrange**) and framework pieces (like the implementation of a specific layout like **Grid**) was the desired outcome. The goal was to provide an extensibility point low in the stack that would allow external developers to create their own frameworks if needed.

## System.Windows.FrameworkElement

[FrameworkElement](#) can be looked at in two different ways. It introduces a set of policies and customizations on the subsystems introduced in lower layers of WPF. It also introduces a set of new subsystems.

The primary policy introduced by [FrameworkElement](#) is around application layout. [FrameworkElement](#) builds on the basic layout contract introduced by [UIElement](#) and adds the notion of a layout "slot" that makes it easier for layout authors to have a consistent set of property driven layout semantics. Properties like [HorizontalAlignment](#), [VerticalAlignment](#), [MinWidth](#), and [Margin](#) (to name a few) give all components derived from [FrameworkElement](#) consistent behavior inside of layout containers.

[FrameworkElement](#) also provides easier API exposure to many features found in the core layers of WPF. For example, [FrameworkElement](#) provides direct access to animation through the [BeginStoryboard](#) method. A [Storyboard](#) provides a way to script multiple animations against a set of properties.

The two most critical things that [FrameworkElement](#) introduces are data binding and styles.

The data binding subsystem in WPF should be relatively familiar to anyone that has used Windows Forms or ASP.NET for creating an application user interface (UI). In each of these systems, there is a simple way to express that you want one or more properties from a given element to be bound to a piece of data. WPF has full support for property binding, transformation, and list binding.

One of the most interesting features of data binding in WPF is the introduction of data templates. Data templates allow you to declaratively specify how a piece of data should be visualized. Instead of creating a custom user interface that can be bound to data, you can instead turn the problem around and let the data determine the display that will be created.

Styling is really a lightweight form of data binding. Using styling you can bind a set of properties from a shared definition to one or more instances of an element. Styles get applied to an element either by explicit reference (by setting the [Style](#) property) or implicitly by associating a style with the CLR type of the element.

## System.Windows.Controls.Control

Control's most significant feature is templating. If you think about WPF's composition system as a retained mode rendering system, templating allows a control to describe its rendering in a parameterized, declarative manner. A [ControlTemplate](#) is really nothing more than a script to create a set of child elements, with bindings to properties offered by the control.

[Control](#) provides a set of stock properties, [Foreground](#), [Background](#), [Padding](#), to name a few, which template authors can then use to customize the display of a control. The implementation of a control provides a data model and interaction model. The interaction model defines a set of commands (like Close for a window) and bindings to input gestures (like clicking the red X in the upper corner of the window). The data model provides a set of properties to either customize the interaction model or customize the display (determined by the template).

This split between the data model (properties), interaction model (commands and events), and display model (templates) enables complete customization of a control's look and behavior.

A common aspect of the data model of controls is the content model. If you look at a control like [Button](#), you will see that it has a property named "Content" of type [Object](#). In Windows Forms and ASP.NET, this property would typically be a string – however that limits the type of content you can put in a button. Content for a button can

either be a simple string, a complex data object, or an entire element tree. In the case of a data object, the data template is used to construct a display.

## Summary

WPF is designed to allow you to create dynamic, data driven presentation systems. Every part of the system is designed to create objects through property sets that drive behavior. Data binding is a fundamental part of the system, and is integrated at every layer.

Traditional applications create a display and then bind to some data. In WPF, everything about the control, every aspect of the display, is generated by some type of data binding. The text found inside a button is displayed by creating a composed control inside of the button and binding its display to the button's content property.

When you begin developing WPF based applications, it should feel very familiar. You can set properties, use objects, and data bind in much the same way that you can using Windows Forms or ASP.NET. With a deeper investigation into the architecture of WPF, you'll find that the possibility exists for creating much richer applications that fundamentally treat data as the core driver of the application.

## See also

- [Visual](#)
- [UIElement](#)
- [ICommand](#)
- [FrameworkElement](#)
- [DispatcherObject](#)
- [CommandBinding](#)
- [Control](#)
- [Data Binding Overview](#)
- [Layout](#)
- [Animation Overview](#)

# XAML in WPF

11/3/2019 • 2 minutes to read • [Edit Online](#)

Extensible Application Markup Language (XAML) is a markup language for declarative application programming. Windows Presentation Foundation (WPF) implements a XAML processor implementation and provides XAML language support. The WPF types are implemented such that they can provide the required type backing for a XAML representation. In general, you can create the majority of your WPF application UI in XAML markup.

## In This Section

- [XAML Overview \(WPF\)](#)
- [XAML Syntax In Detail](#)
- [Code-Behind and XAML in WPF](#)
- [XAML and Custom Classes for WPF](#)
- [Markup Extensions and WPF XAML](#)
- [XAML Namespaces and Namespace Mapping for WPF XAML](#)
- [WPF XAML Namescopes](#)
- [Inline Styles and Templates](#)
- [White-space Processing in XAML](#)
- [TypeConverters and XAML](#)
- [XML Character Entities and XAML](#)
- [XAML Namespace \(x:\) Language Features](#)
- [WPF XAML Extensions](#)
- [Markup Compatibility \(mc:\) Language Features](#)

## Related Sections

- [WPF Architecture](#)
- [Base Elements](#)
- [Element Tree and Serialization](#)
- [Properties](#)
- [Events](#)
- [Input](#)
- [Resources](#)
- [Styling and Templating](#)
- [Threading Model](#)

2 minutes to read

# XAML Syntax In Detail

11/7/2019 • 26 minutes to read • [Edit Online](#)

This topic defines the terms that are used to describe the elements of XAML syntax. These terms are used frequently throughout the remainder of this documentation, both for WPF documentation specifically and for the other frameworks that use XAML or the basic XAML concepts enabled by the XAML language support at the System.Xaml level. This topic expands on the basic terminology introduced in the topic [XAML Overview \(WPF\)](#).

## The XAML Language Specification

The XAML syntax terminology defined here is also defined or referenced within the XAML language specification. XAML is a language based on XML and follows or expands upon XML structural rules. Some of the terminology is shared from or is based on the terminology commonly used when describing the XML language or the XML document object model.

For more information about the XAML language specification, download [\[MS-XAML\]](#) from the Microsoft Download Center.

## XAML and CLR

XAML is a markup language. The common language runtime (CLR), as implied by its name, enables runtime execution. XAML is not by itself one of the common languages that is directly consumed by the CLR runtime. Instead, you can think of XAML as supporting its own type system. The particular XAML parsing system that is used by WPF is built on the CLR and the CLR type system. XAML types are mapped to CLR types to instantiate a run time representation when the XAML for WPF is parsed. For this reason, the remainder of discussion of syntax in this document will include references to the CLR type system, even though the equivalent syntax discussions in the XAML language specification do not. (Per the XAML language specification level, XAML types could be mapped to any other type system, which does not have to be the CLR, but that would require the creation and use of a different XAML parser.)

### Members of Types and Class Inheritance

Properties and events as they appear as XAML members of a WPF type are often inherited from base types. For example, consider this example: `<Button Background="Blue" .../>`. The [Background](#) property is not an immediately declared property on the [Button](#) class, if you were to look at the class definition, reflection results, or the documentation. Instead, [Background](#) is inherited from the base [Control](#) class.

The class inheritance behavior of WPF XAML elements is a significant departure from a schema-enforced interpretation of XML markup. Class inheritance can become complex, particularly when intermediate base classes are abstract, or when interfaces are involved. This is one reason that the set of XAML elements and their permissible attributes is difficult to represent accurately and completely using the schema types that are typically used for XML programming, such as DTD or XSD format. Another reason is that extensibility and type-mapping features of the XAML language itself preclude completeness of any fixed representation of the permissible types and members.

## Object Element Syntax

*Object element syntax* is the XAML markup syntax that instantiates a CLR class or structure by declaring an XML element. This syntax resembles the element syntax of other markup languages such as HTML. Object element syntax begins with a left angle bracket (<), followed immediately by the type name of the class or structure being instantiated. Zero or more spaces can follow the type name, and zero or more attributes may also be declared on

the object element, with one or more spaces separating each attribute name="value" pair. Finally, one of the following must be true:

- The element and tag must be closed by a forward slash (/) followed immediately by a right angle bracket (>).
- The opening tag must be completed by a right angle bracket (>). Other object elements, property elements, or inner text, can follow the opening tag. Exactly what content may be contained here is typically constrained by the object model of the element. The equivalent closing tag for the object element must also exist, in proper nesting and balance with other opening and closing tag pairs.

XAML as implemented by .NET has a set of rules that map object elements into types, attributes into properties or events, and XAML namespaces to CLR namespaces plus assembly. For WPF and .NET, XAML object elements map to .NET types as defined in referenced assemblies, and the attributes map to members of those types. When you reference a CLR type in XAML, you have access to the inherited members of that type as well.

For example, the following example is object element syntax that instantiates a new instance of the [Button](#) class, and also specifies a [Name](#) attribute and a value for that attribute:

```
<Button Name="CheckoutButton"/>
```

The following example is object element syntax that also includes XAML content property syntax. The inner text contained within will be used to set the [TextBox](#) XAML content property, [Text](#).

```
<TextBox>This is a Text Box</TextBox>
```

## Content Models

A class might support a usage as a XAML object element in terms of the syntax, but that element will only function properly in an application or page when it is placed in an expected position of an overall content model or element tree. For example, a [MenuItem](#) should typically only be placed as a child of a [MenuBase](#) derived class such as [Menu](#). Content models for specific elements are documented as part of the remarks on the class pages for controls and other WPF classes that can be used as XAML elements.

## Properties of Object Elements

Properties in XAML are set by a variety of possible syntaxes. Which syntax can be used for a particular property will vary, based on the underlying type system characteristics of the property that you are setting.

By setting values of properties, you add features or characteristics to objects as they exist in the run time object graph. The initial state of the created object from a object element is based on the parameterless constructor behavior. Typically, your application will use something other than a completely default instance of any given object.

## Attribute Syntax (Properties)

Attribute syntax is the XAML markup syntax that sets a value for a property by declaring an attribute on an existing object element. The attribute name must match the CLR member name of the property of the class that backs the relevant object element. The attribute name is followed by an assignment operator (=). The attribute value must be a string enclosed within quotes.

#### **NOTE**

You can use alternating quotes to place a literal quotation mark within an attribute. For instance you can use single quotes as a means to declare a string that contains a double quote character within it. Whether you use single or double quotes, you should use a matching pair for opening and closing the attribute value string. There are also escape sequences or other techniques available for working around character restrictions imposed by any particular XAML syntax. See [XML Character Entities and XAML](#).

In order to be set through attribute syntax, a property must be public and must be writeable. The value of the property in the backing type system must be a value type, or must be a reference type that can be instantiated or referenced by a XAML processor when accessing the relevant backing type.

For WPF XAML events, the event that is referenced as the attribute name must be public and have a public delegate.

The property or event must be a member of the class or structure that is instantiated by the containing object element.

#### **Processing of Attribute Values**

The string value contained within the opening and closing quotation marks is processed by a XAML processor. For properties, the default processing behavior is determined by the type of the underlying CLR property.

The attribute value is filled by one of the following, using this processing order:

1. If the XAML processor encounters a curly brace, or an object element that derives from [MarkupExtension](#), then the referenced markup extension is evaluated first rather than processing the value as a string, and the object returned by the markup extension is used as the value. In many cases the object returned by a markup extension will be a reference to an existing object, or an expression that defers evaluation until run time, and is not a newly instantiated object.
2. If the property is declared with an attributed [TypeConverter](#), or the value type of that property is declared with an attributed [TypeConverter](#), the string value of the attribute is submitted to the type converter as a conversion input, and the converter will return a new object instance.
3. If there is no [TypeConverter](#), a direct conversion to the property type is attempted. This final level is a direct conversion at the parser-native value between XAML language primitive types, or a check for the names of named constants in an enumeration (the parser then accesses the matching values).

#### **Enumeration Attribute Values**

Enumerations in XAML are processed intrinsically by XAML parsers, and the members of an enumeration should be specified by specifying the string name of one of the enumeration's named constants.

For nonflag enumeration values, the native behavior is to process the string of an attribute value and resolve it to one of the enumeration values. You do not specify the enumeration in the format *Enumeration.Value*, as you do in code. Instead, you specify only *Value*, and *Enumeration* is inferred by the type of the property you are setting. If you specify an attribute in the *Enumeration.Value* form, it will not resolve correctly.

For flagwise enumerations, the behavior is based on the [Enum.Parse](#) method. You can specify multiple values for a flagwise enumeration by separating each value with a comma. However, you cannot combine enumeration values that are not flagwise. For instance, you cannot use the comma syntax to attempt to create a [Trigger](#) that acts on multiple conditions of a nonflag enumeration:

```

<!--This will not compile, because Visibility is not a flagwise enumeration.-->
...
<Trigger Property="Visibility" Value="Collapsed,Hidden">
  <Setter ... />
</Trigger>
...

```

Flagwise enumerations that support attributes that are settable in XAML are rare in WPF. However, one such enumeration is [StyleSimulations](#). You could, for instance, use the comma-delimited flagwise attribute syntax to modify the example provided in the Remarks for the [Glyphs](#) class: `StyleSimulations = "BoldSimulation"` could become `StyleSimulations = "BoldSimulation,ItalicSimulation"`. [KeyBinding.Modifiers](#) is another property where more than one enumeration value can be specified. However, this property happens to be a special case, because the [ModifierKeys](#) enumeration supports its own type converter. The type converter for modifiers uses a plus sign (+) as a delimiter rather than a comma (,). This conversion supports the more traditional syntax to represent key combinations in Microsoft Windows programming, such as "Ctrl+Alt".

### Properties and Event Member Name References

When specifying an attribute, you can reference any property or event that exists as a member of the CLR type you instantiated for the containing object element.

Or, you can reference an attached property or attached event, independent of the containing object element. (Attached properties are discussed in an upcoming section.)

You can also name any event from any object that is accessible through the default namespace by using a `typeName.event` partially qualified name; this syntax supports attaching handlers for routed events where the handler is intended to handle events routing from child elements, but the parent element does not also have that event in its members table. This syntax resembles an attached event syntax, but the event here is not a true attached event. Instead, you are referencing an event with a qualified name. For more information, see [Routed Events Overview](#).

For some scenarios, property names are sometimes provided as the value of an attribute, rather than the attribute name. That property name can also include qualifiers, such as the property specified in the form `ownerType.dependencyPropertyName`. This scenario is common when writing styles or templates in XAML. The processing rules for property names provided as an attribute value are different, and are governed by the type of the property being set or by the behaviors of particular WPF subsystems. For details, see [Styling and Templating](#).

Another usage for property names is when an attribute value describes a property-property relationship. This feature is used for data binding and for storyboard targets, and is enabled by the [PropertyPath](#) class and its type converter. For a more complete description of the lookup semantics, see [PropertyPath XAML Syntax](#).

## Property Element Syntax

*Property element syntax* is a syntax that diverges somewhat from the basic XML syntax rules for elements. In XML, the value of an attribute is a de facto string, with the only possible variation being which string encoding format is being used. In XAML, you can assign other object elements to be the value of a property. This capability is enabled by the property element syntax. Instead of the property being specified as an attribute within the element tag, the property is specified using an opening element tag in `elementTypeName.propertyName` form, the value of the property is specified within, and then the property element is closed.

Specifically, the syntax begins with a left angle bracket (<), followed immediately by the type name of the class or structure that the property element syntax is contained within. This is followed immediately by a single dot (.), then by the name of a property, then by a right angle bracket (>). As with attribute syntax, that property must exist within the declared public members of the specified type. The value to be assigned to the property is contained within the property element. Typically, the value is given as one or more object elements, because specifying objects as values is the scenario that property element syntax is intended to address. Finally, an equivalent closing

tag specifying the same *elementType**.propertyName* combination must be provided, in proper nesting and balance with other element tags.

For example, the following is property element syntax for the [ContextMenu](#) property of a [Button](#).

```
<Button>
  <Button.ContextMenu>
    <ContextMenu>
      <MenuItem Header="1">First item</MenuItem>
      <MenuItem Header="2">Second item</MenuItem>
    </ContextMenu>
  </Button.ContextMenu>
  Right-click me!</Button>
```

The value within a property element can also be given as inner text, in cases where the property type being specified is a primitive value type, such as [String](#), or an enumeration where a name is specified. These two usages are somewhat uncommon, because each of these cases could also use a simpler attribute syntax. One scenario for filling a property element with a string is for properties that are not the XAML content property but still are used for representation of UI text, and particular white-space elements such as linefeeds are required to appear in that UI text. Attribute syntax cannot preserve linefeeds, but property element syntax can, so long as significant whitespace preservation is active (for details, see [White space processing in XAML](#)). Another scenario is so that [x:Uid Directive](#) can be applied to the property element and thus mark the value within as a value that should be localized in the WPF output BAML or by other techniques.

A property element is not represented in the WPF logical tree. A property element is just a particular syntax for setting a property, and is not an element that has an instance or object backing it. (For details on the logical tree concept, see [Trees in WPF](#).)

For properties where both attribute and property element syntax are supported, the two syntaxes generally have the same result, although subtleties such as white-space handling can vary slightly between syntaxes.

## Collection Syntax

The XAML specification requires XAML processor implementations to identify properties where the value type is a collection. The general XAML processor implementation in .NET is based on managed code and the CLR, and it identifies collection types through one of the following:

- Type implements [IList](#).
- Type implements [IDictionary](#).
- Type derives from [Array](#) (for more information about arrays in XAML, see [x:Array Markup Extension](#).)

If the type of a property is a collection, then the inferred collection type does not need to be specified in the markup as an object element. Instead, the elements that are intended to become the items in the collection are specified as one or more child elements of the property element. Each such item is evaluated to an object during loading and added to the collection by calling the `Add` method of the implied collection. For example, the [Triggers](#) property of [Style](#) takes the specialized collection type [TriggerCollection](#), which implements [IList](#). It is not necessary to instantiate a [TriggerCollection](#) object element in the markup. Instead, you specify one or more [Trigger](#) items as elements within the `Style.Triggers` property element, where [Trigger](#) (or a derived class) is the type expected as the item type for the strongly typed and implicit [TriggerCollection](#).

```

<Style x:Key="SpecialButton" TargetType="{x:Type Button}">
  <Style.Triggers>
    <Trigger Property="Button.IsMouseOver" Value="true">
      <Setter Property = "Background" Value="Red"/>
    </Trigger>
    <Trigger Property="Button.IsPressed" Value="true">
      <Setter Property = "Foreground" Value="Green"/>
    </Trigger>
  </Style.Triggers>
</Style>

```

A property may be both a collection type and the XAML content property for that type and derived types, which is discussed in the next section of this topic.

An implicit collection element creates a member in the logical tree representation, even though it does not appear in the markup as an element. Usually the constructor of the parent type performs the instantiation for the collection that is one of its properties, and the initially empty collection becomes part of the object tree.

#### NOTE

The generic list and dictionary interfaces ([IList<T>](#) and [IDictionary< TKey, TValue >](#)) are not supported for collection detection. However, you can use the [List<T>](#) class as a base class, because it implements [IList](#) directly, or [Dictionary< TKey, TValue >](#) as a base class, because it implements [IDictionary](#) directly.

In the .NET Reference pages for collection types, this syntax with the deliberate omission of the object element for a collection is occasionally noted in the XAML syntax sections as [Implicit Collection Syntax](#).

With the exception of the root element, every object element in a XAML file that is nested as a child element of another element is really an element that is one or both of the following cases: a member of an implicit collection property of its parent element, or an element that specifies the value of the XAML content property for the parent element (XAML content properties will be discussed in an upcoming section). In other words, the relationship of parent elements and child elements in a markup page is really a single object at the root, and every object element beneath the root is either a single instance that provides a property value of the parent, or one of the items within a collection that is also a collection-type property value of the parent. This single-root concept is common with XML, and is frequently reinforced in the behavior of APIs that load XAML such as [Load](#).

The following example is a syntax with the object element for a collection ([GradientStopCollection](#)) specified explicitly.

```

<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStopCollection>
      <GradientStop Offset="0.0" Color="Red" />
      <GradientStop Offset="1.0" Color="Blue" />
    </GradientStopCollection>
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>

```

Note that it is not always possible to explicitly declare the collection. For instance, attempting to declare [TriggerCollection](#) explicitly in the previously shown [Triggers](#) example would fail. Explicitly declaring the collection requires that the collection class must support a parameterless constructor, and [TriggerCollection](#) does not have a parameterless constructor.

## XAML Content Properties

XAML content syntax is a syntax that is only enabled on classes that specify the [ContentPropertyAttribute](#) as part

of their class declaration. The [ContentPropertyAttribute](#) references the property name that is the content property for that type of element (including derived classes). When processed by a XAML processor, any child elements or inner text that are found between the opening and closing tags of the object element will be assigned to be the value of the XAML content property for that object. You are permitted to specify explicit property elements for the content property, but this usage is not generally shown in the XAML syntax sections in the .NET reference. The explicit/verbose technique has occasional value for markup clarity or as a matter of markup style, but usually the intent of a content property is to streamline the markup so that elements that are intuitively related as parent-child can be nested directly. Property element tags for other properties on an element are not assigned as "content" per a strict XAML language definition; they are processed previously in the XAML parser's processing order and are not considered to be "content".

### XAML Content Property Values Must Be Contiguous

The value of a XAML content property must be given either entirely before or entirely after any other property elements on that object element. This is true whether the value of a XAML content property is specified as a string, or as one or more objects. For example, the following markup does not parse:

```
<Button>I am a
<Button.Background>Blue</Button.Background>
blue button</Button>
```

This is illegal essentially because if this syntax were made explicit by using property element syntax for the content property, then the content property would be set twice:

```
<Button>
<Button.Content>I am a </Button.Content>
<Button.Background>Blue</Button.Background>
<Button.Content> blue button</Button.Content>
</Button>
```

A similarly illegal example is if the content property is a collection, and child elements are interspersed with property elements:

```
<StackPanel>
<Button>This example</Button>
<StackPanel.Resources>
<SolidColorBrush x:Key="BlueBrush" Color="Blue"/>
</StackPanel.Resources>
<Button>... is illegal XAML</Button>
</StackPanel>
```

## Content Properties and Collection Syntax Combined

In order to accept more than a single object element as content, the type of the content property must specifically be a collection type. Similar to property element syntax for collection types, a XAML processor must identify types that are collection types. If an element has a XAML content property and the type of the XAML content property is a collection, then the implied collection type does not need to be specified in the markup as an object element and the XAML content property does not need to be specified as a property element. Therefore the apparent content model in the markup can now have more than one child element assigned as the content. The following is content syntax for a [Panel](#) derived class. All [Panel](#) derived classes establish the XAML content property to be [Children](#), which requires a value of type [UIElementCollection](#).

```

<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <StackPanel>
        <Button>Button 1</Button>
        <Button>Button 2</Button>
        <Button>Button 3</Button>
    </StackPanel>
</Page>

```

Note that neither the property element for [Children](#) nor the element for the [UIElementCollection](#) is required in the markup. This is a design feature of XAML so that recursively contained elements that define a UI are more intuitively represented as a tree of nested elements with immediate parent-child element relationships, without intervening property element tags or collection objects. In fact, [UIElementCollection](#) cannot be specified explicitly in markup as an object element, by design. Because its only intended use is as an implicit collection, [UIElementCollection](#) does not expose a public parameterless constructor and thus cannot be instantiated as an object element.

### Mixing Property Elements and Object Elements in an Object with a Content Property

The XAML specification declares that a XAML processor can enforce that object elements that are used to fill the XAML content property within an object element must be contiguous, and must not be mixed. This restriction against mixing property elements and content is enforced by the WPF XAML processors.

You can have a child object element as the first immediate markup within an object element. Then you can introduce property elements. Or, you can specify one or more property elements, then content, then more property elements. But once a property element follows content, you cannot introduce any further content, you can only add property elements.

This content / property element order requirement does not apply to inner text used as content. However, it is still a good markup style to keep inner text contiguous, because significant white space will be difficult to detect visually in the markup if property elements are interspersed with inner text.

## XAML Namespaces

None of the preceding syntax examples specified a XAML namespace other than the default XAML namespace. In typical WPF applications, the default XAML namespace is specified to be the WPF namespace. You can specify XAML namespaces other than the default XAML namespace and still use similar syntax. But then, anywhere where a class is named that is not accessible within the default XAML namespace, that class name must be preceded with the prefix of the XAML namespace as mapped to the corresponding CLR namespace. For example, `<custom:Example/>` is object element syntax to instantiate an instance of the `Example` class, where the CLR namespace containing that class (and possibly the external assembly information that contains backing types) was previously mapped to the `custom` prefix.

For more information about XAML namespaces, see [XAML Namespaces and Namespace Mapping for WPF XAML](#).

## Markup Extensions

XAML defines a markup extension programming entity that enables an escape from the normal XAML processor handling of string attribute values or object elements, and defers the processing to a backing class. The character that identifies a markup extension to a XAML processor when using attribute syntax is the opening curly brace ({), followed by any character other than a closing curly brace (}). The first string following the opening curly brace must reference the class that provides the particular extension behavior, where the reference may omit the substring "Extension" if that substring is part of the true class name. Thereafter, a single space may appear, and

then each succeeding character is used as input by the extension implementation, up until the closing curly brace is encountered.

The .NET XAML implementation uses the [MarkupExtension](#) abstract class as the basis for all of the markup extensions supported by WPF as well as other frameworks or technologies. The markup extensions that WPF specifically implements are often intended to provide a means to reference other existing objects, or to make deferred references to objects that will be evaluated at run time. For example, a simple WPF data binding is accomplished by specifying the `{Binding}` markup extension in place of the value that a particular property would ordinarily take. Many of the WPF markup extensions enable an attribute syntax for properties where an attribute syntax would not otherwise be possible. For example, a [Style](#) object is a relatively complex type that contains a nested series of objects and properties. Styles in WPF are typically defined as a resource in a [ResourceDictionary](#), and then referenced through one of the two WPF markup extensions that request a resource. The markup extension defers the evaluation of the property value to a resource lookup and enables providing the value of the [Style](#) property, taking type [Style](#), in attribute syntax as in the following example:

```
<Button Style="{StaticResource MyStyle}">My button</Button>
```

Here, `StaticResource` identifies the [StaticResourceExtension](#) class providing the markup extension implementation. The next string `MyStyle` is used as the input for the non-default [StaticResourceExtension](#) constructor, where the parameter as taken from the extension string declares the requested [ResourceKey](#). `MyStyle` is expected to be the `x:Key` value of a [Style](#) defined as a resource. The [StaticResource Markup Extension](#) usage requests that the resource be used to provide the [Style](#) property value through static resource lookup logic at load time.

For more information about markup extensions, see [Markup Extensions and WPF XAML](#). For a reference of markup extensions and other XAML programming features enabled in the general .NET XAML implementation, see [XAML Namespace \(x\) Language Features](#). For WPF-specific markup extensions, see [WPF XAML Extensions](#).

## Attached Properties

Attached properties are a programming concept introduced in XAML whereby properties can be owned and defined by a particular type, but set as attributes or property elements on any element. The primary scenario that attached properties are intended for is to enable child elements in a markup structure to report information to a parent element without requiring an extensively shared object model across all elements. Conversely, attached properties can be used by parent elements to report information to child elements. For more information on the purpose of attached properties and how to create your own attached properties, see [Attached Properties Overview](#).

Attached properties use a syntax that superficially resembles property element syntax, in that you also specify a `typeName.propertyName` combination. There are two important differences:

- You can use the `typeName.propertyName` combination even when setting an attached property through attribute syntax. Attached properties are the only case where qualifying the property name is a requirement in an attribute syntax.
- You can also use property element syntax for attached properties. However, for typical property element syntax, the `typeName` you specify is the object element that contains the property element. If you are referring to an attached property, then the `typeName` is the class that defines the attached property, not the containing object element.

## Attached Events

Attached events are another programming concept introduced in XAML where events can be defined by a specific type, but handlers may be attached on any object element. In the WOF implementation, often the type that defines an attached event is a static type that defines a service, and sometimes those attached events are exposed

by a routed event alias in types that expose the service. Handlers for attached events are specified through attribute syntax. As with attached events, the attribute syntax is expanded for attached events to allow a `typeName.eventName` usage, where `typeName` is the class that provides `Add` and `Remove` event handler accessors for the attached event infrastructure, and `eventName` is the event name.

## Anatomy of a XAML Root Element

The following table shows a typical XAML root element broken down, showing the specific attributes of a root element:

<code>&lt;Page</code>	Opening object element of the root element
<code>xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"</code>	The default (WPF) XAML namespace
<code>xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"</code>	The XAML language XAML namespace
<code>x:Class="ExampleNamespace.ExampleCode"</code>	The partial class declaration that connects markup to any code-behind defined for the partial class
<code>&gt;</code>	End of object element for the root. Object is not closed yet because the element contains child elements

## Optional and Nonrecommended XAML Usages

The following sections describe XAML usages that are technically supported by XAML processors, but that produce verbosity or other aesthetic issues that interfere with XAML files remaining human-readable when you develop applications that contain XAML sources.

### Optional Property Element Usages

Optional property element usages include explicitly writing out element content properties that the XAML processor considers implicit. For example, when you declare the contents of a `Menu`, you could choose to explicitly declare the `Items` collection of the `Menu` as a `<Menu.Items>` property element tag, and place each `MenuItem` within `<Menu.Items>`, rather than using the implicit XAML processor behavior that all child elements of a `Menu` must be a `MenuItem` and are placed in the `Items` collection. Sometimes the optional usages can help to visually clarify the object structure as represented in the markup. Or sometimes an explicit property element usage can avoid markup that is technically functional but visually confusing, such as nested markup extensions within an attribute value.

### Full `typeName.memberName` Qualified Attributes

The `typeName.memberName` form for an attribute actually works more universally than just the routed event case. But in other situations that form is superfluous and you should avoid it, if only for reasons of markup style and readability. In the following example, each of the three references to the `Background` attribute are completely equivalent:

```
<Button Background="Blue">Background</Button>
<Button Button.Background="Blue">Button.Background</Button>
<Button Control.Background="Blue">Control.Background</Button>
```

`Button.Background` works because the qualified lookup for that property on `Button` is successful (`Background` was inherited from `Control`) and `Button` is the class of the object element or a base class. `Control.Background` works because the `Control` class actually defines `Background` and `Control` is a `Button` base class.

However, the following `typeName.memberName` form example does not work and is thus shown commented:

```
<!--<Button Label.Background="Blue">Does not work</Button> -->
```

`Label` is another derived class of `Control`, and if you had specified `Label.Background` within a `Label` object element, this usage would have worked. However, because `Label` is not the class or base class of `Button`, the specified XAML processor behavior is to then process `Label.Background` as an attached property. `Label.Background` is not an available attached property, and this usage fails.

### `baseTypeName.memberName` Property Elements

In an analogous way to how the `typeName.memberName` form works for attribute syntax, a `baseTypeName.memberName` syntax works for property element syntax. For instance, the following syntax works:

```
<Button>Control.Background PE
  <Control.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="LimeGreen" Offset="1.0" />
    </LinearGradientBrush>
  </Control.Background>
</Button>
```

Here, the property element was given as `Control.Background` even though the property element was contained in `Button`.

But just like `typeName.memberName` form for attributes, `baseTypeName.memberName` is poor style in markup, and you should avoid it.

## See also

- [XAML Overview \(WPF\)](#)
- [XAML Namespace \(x:\) Language Features](#)
- [WPF XAML Extensions](#)
- [Dependency Properties Overview](#)
- [TypeConverters and XAML](#)
- [XAML and Custom Classes for WPF](#)

# Code-Behind and XAML in WPF

11/7/2019 • 3 minutes to read • [Edit Online](#)

Code-behind is a term used to describe the code that is joined with markup-defined objects, when a XAML page is markup-compiled. This topic describes requirements for code-behind as well as an alternative inline code mechanism for code in XAML.

This topic contains the following sections:

- [Prerequisites](#)
- [Code-Behind and the XAML Language](#)
- [Code-behind, Event Handler, and Partial Class Requirements in WPF](#)
- [x:Code](#)
- [Inline Code Limitations](#)

## Prerequisites

This topic assumes that you have read the [XAML Overview \(WPF\)](#) and have some basic knowledge of the CLR and object-oriented programming.

## Code-Behind and the XAML Language

The XAML language includes language-level features that make it possible to associate code files with markup files, from the markup file side. Specifically, the XAML language defines the language features [x:Class Directive](#), [x:Subclass Directive](#), and [x:ClassModifier Directive](#). Exactly how the code should be produced, and how to integrate markup and code, is not part of what the XAML language specifies. It is left up to frameworks such as WPF to determine how to integrate the code, how to use XAML in the application and programming models, and the build actions or other support that all this requires.

## Code-behind, Event Handler, and Partial Class Requirements in WPF

- The partial class must derive from the type that backs the root element.
- Note that under the default behavior of the markup compile build actions, you can leave the derivation blank in the partial class definition on the code-behind side. The compiled result will assume the page root's backing type to be the basis for the partial class, even if it not specified. However, relying on this behavior is not a best practice.
- The event handlers you write in the code-behind must be instance methods and cannot be static methods. These methods must be defined by the partial class within the CLR namespace identified by `x:class`. You cannot qualify the name of an event handler to instruct a XAML processor to look for an event handler for event wiring in a different class scope.
- The handler must match the delegate for the appropriate event in the backing type system.
- For the Microsoft Visual Basic language specifically, you can use the language-specific `Handles` keyword to associate handlers with instances and events in the handler declaration, instead of attaching handlers with attributes in XAML. However, this technique does have some limitations because the `Handles` keyword cannot support all of the specific features of the WPF event system, such as certain routed event scenarios

or attached events. For details, see [Visual Basic and WPF Event Handling](#).

## x:Code

x:Code is a directive element defined in XAML. An `x:Code` directive element can contain inline programming code. The code that is defined inline can interact with the XAML on the same page. The following example illustrates inline C# code. Notice that the code is inside the `x:Code` element and that the code must be surrounded by `<![CDATA[ ... ]]>` to escape the contents for XML, so that a XAML processor (interpreting either the XAML schema or the WPF schema) will not try to interpret the contents literally as XML.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="MyNamespace.MyCanvasCodeInline"
>
    <Button Name="button1" Click="Clicked">Click Me!</Button>
    <x:Code><![CDATA[
        void Clicked(object sender, RoutedEventArgs e)
        {
            button1.Content = "Hello World";
        }
    ]]></x:Code>
</Page>
```

## Inline Code Limitations

You should consider avoiding or limiting the use of inline code. In terms of architecture and coding philosophy, maintaining a separation between markup and code-behind keeps the designer and developer roles much more distinct. On a more technical level, the code that you write for inline code can be awkward to write, because you are always writing into the XAML generated partial class, and can only use the default XML namespace mappings.

Because you cannot add `using` statements, you must fully qualify many of the API calls that you make. The default WPF mappings include most but not all CLR namespaces that are present in the WPF assemblies; you will have to fully qualify calls to types and members contained within the other CLR namespaces. You also cannot define anything beyond the partial class in the inline code, and all user code entities you reference must exist as a member or variable within the generated partial class. Other language specific programming features, such as macros or `#ifdef` against global variables or build variables, are also not available. For more information, see [x:Code Intrinsic XAML Type](#).

## See also

- [XAML Overview \(WPF\)](#)
- [x:Code Intrinsic XAML Type](#)
- [Building a WPF Application](#)
- [XAML Syntax In Detail](#)

# XAML and Custom Classes for WPF

11/3/2019 • 11 minutes to read • [Edit Online](#)

XAML as implemented in common language runtime (CLR) frameworks supports the ability to define a custom class or structure in any common language runtime (CLR) language, and then access that class using XAML markup. You can use a mixture of Windows Presentation Foundation (WPF)-defined types and your custom types within the same markup file, typically by mapping the custom types to a XAML namespace prefix. This topic discusses the requirements that a custom class must satisfy to be usable as a XAML element.

## Custom Classes in Applications or Assemblies

Custom classes that are used in XAML can be defined in two distinct ways: within the code-behind or other code that produces the primary Windows Presentation Foundation (WPF) application, or as a class in a separate assembly, such as an executable or DLL used as a class library. Each of these approaches has particular advantages and disadvantages.

- The advantage of creating a class library is that any such custom classes can be shared across many different possible applications. A separate library also makes versioning issues of applications easier to control, and simplifies creating a class where the intended class usage is as a root element on a XAML page.
- The advantage of defining the custom classes in the application is that this technique is relatively lightweight and minimizes the deployment and testing issues encountered when you introduce separate assemblies beyond the main application executable.
- Whether defined in the same or different assembly, custom classes need to be mapped between CLR namespace and XML namespace in order to be used in XAML as elements. See [XAML Namespaces and Namespace Mapping for WPF XAML](#).

## Requirements for a Custom Class as a XAML Element

In order to be able to be instantiated as an object element, your class must meet the following requirements:

- Your custom class must be public and support a default (parameterless) public constructor. (See following section for notes regarding structures.)
- Your custom class must not be a nested class. Nested classes and the "dot" in their general CLR usage syntax interfere with other WPF and/or XAML features such as attached properties.

In addition to enabling object element syntax, your object definition also enables property element syntax for any other public properties that take that object as the value type. This is because the object can now be instantiated as an object element and can fill the property element value of such a property.

### Structures

Structures that you define as custom types are always able to be constructed in XAML in WPF .This is because the CLR compilers implicitly create a parameterless constructor for a structure that initializes all property values to their defaults. In some cases, the default construction behavior and/or object element usage for a structure is not desirable. This might be because the structure is intended to fill values and function conceptually as a union, where the values contained might have mutually exclusive interpretations and thus none of its properties are settable. A WPF example of such a structure is [GridLength](#). Generally, such structures should implement a type converter such that the values can be expressed in attribute form, using string conventions that create the different interpretations or modes of the structure's values. The structure should also expose similar behavior for code construction through a non-parameterless constructor.

# Requirements for Properties of a Custom Class as XAML Attributes

Properties must reference a by-value type (such as a primitive), or use a class for type that has either a parameterless constructor or a dedicated type converter that a XAML processor can access. In the CLR XAML implementation, XAML processors either find such converters through native support for language primitives, or through application of [TypeConverterAttribute](#) to a type or member in backing type definitions

Alternatively, the property may reference an abstract class type, or an interface. For abstract classes or interfaces, the expectation for XAML parsing is that the property value must be filled with practical class instances that implement the interface, or instances of types that derive from the abstract class.

Properties can be declared on an abstract class, but can only be set on practical classes that derive from the abstract class. This is because creating the object element for the class at all requires a public parameterless constructor on the class.

## TypeConverter Enabled Attribute Syntax

If you provide a dedicated, attributed type converter at the class level, the applied type conversion enables attribute syntax for any property that needs to instantiate that type. A type converter does not enable object element usage of the type; only the presence of a parameterless constructor for that type enables object element usage. Therefore, properties that are type-converter enabled are generally speaking not usable in property syntax, unless the type itself also supports object element syntax. The exception to this is that you can specify a property element syntax, but have the property element contain a string. That usage is really essentially equivalent to an attribute syntax usage, and such a usage is not common unless there is a need for more robust white-space handling of the attribute value. For example, the following is a property element usage that takes a string, and the attribute usage equivalent:

```
<Button>Hallo!
<Button.Language>
  de-DE
</Button.Language>
</Button>
```

```
<Button Language="de-DE">Hallo!</Button>
```

Examples of properties where attribute syntax is allowed but property element syntax that contains an object element is disallowed through XAML are various properties that take the [Cursor](#) type. The [Cursor](#) class has a dedicated type converter [CursorConverter](#), but does not expose a parameterless constructor, so the [Cursor](#) property can only be set through attribute syntax even though the actual [Cursor](#) type is a reference type.

## Per-Property Type Converters

Alternatively, the property itself may declare a type converter at the property level. This enables a "mini language" that instantiates objects of the type of the property inline, by processing incoming string values of the attribute as input for a [ConvertFrom](#) operation based on the appropriate type. Typically this is done to provide a convenience accessor, and not as the sole means to enable setting a property in XAML. However, it is also possible to use type converters for attributes where you want to use existing CLR types that do not supply either a parameterless constructor or an attributed type converter. Examples from the WPF API are certain properties that take the [CultureInfo](#) type. In this case, WPF used the existing Microsoft .NET Framework [CultureInfo](#) type to better address compatibility and migration scenarios that were used in earlier versions of frameworks, but the [CultureInfo](#) type did not support the necessary constructors or type-level type conversion to be usable as a XAML property value directly.

Whenever you expose a property that has a XAML usage, particularly if you are a control author, you should strongly consider backing that property with a dependency property. This is particularly true if you use the existing Windows Presentation Foundation (WPF) implementation of the XAML processor, because you can

improve performance by using [DependencyProperty](#) backing. A dependency property will expose property system features for your property that users will come to expect for a XAML accessible property. This includes features such as animation, data binding, and style support. For more information, see [Custom Dependency Properties](#) and [XAML Loading and Dependency Properties](#).

### Writing and Attributing a Type Converter

You occasionally will need to write a custom [TypeConverter](#) derived class to provide type conversion for your property type. For instructions on how to derive from and create a type converter that can support XAML usages, and how to apply the [TypeConverterAttribute](#), see [TypeConverters and XAML](#).

## Requirements for XAML Event Handler Attribute Syntax on Events of a Custom Class

To be usable as a CLR event, the event must be exposed as a public event on a class that supports a parameterless constructor, or on an abstract class where the event can be accessed on derived classes. In order to be used conveniently as a routed event, your CLR event should implement explicit `add` and `remove` methods, which add and remove handlers for the CLR event signature and forward those handlers to the [AddHandler](#) and [RemoveHandler](#) methods. These methods add or remove the handlers to the routed event handler store on the instance that the event is attached to.

#### NOTE

It is possible to register handlers directly for routed events using [AddHandler](#), and to deliberately not define a CLR event that exposes the routed event. This is not generally recommended because the event will not enable XAML attribute syntax for attaching handlers, and your resulting class will offer a less transparent XAML view of that type's capabilities.

## Writing Collection Properties

Properties that take a collection type have a XAML syntax that enables you to specify objects that are added to the collection. This syntax has two notable features.

- The object that is the collection object does not need to be specified in object element syntax. The presence of that collection type is implicit whenever you specify a property in XAML that takes a collection type.
- Child elements of the collection property in markup are processed to become members of the collection. Ordinarily, the code access to the members of a collection is performed through list/dictionary methods such as `Add`, or through an indexer. But XAML syntax does not support methods or indexers (exception: XAML 2009 can support methods, but using XAML 2009 restricts the possible WPF usages; see [XAML 2009 Language Features](#)). Collections are obviously a very common requirement for building a tree of elements, and you need some way to populate these collections in declarative XAML. Therefore, child elements of a collection property are processed by adding them to the collection that is the collection property type value.

The .NET Framework XAML Services implementation and thus the WPF XAML processor uses the following definition for what constitutes a collection property. The property type of the property must implement one of the following:

- Implements [IList](#).
- Implements [IDictionary](#) or the generic equivalent ([IDictionary< TKey, TValue >](#)).
- Derives from [Array](#) (for more information about arrays in XAML, see [x:Array Markup Extension](#).)
- Implements [IAddChild](#) (an interface defined by WPF).

Each of these types in CLR has an `Add` method, which is used by the XAML processor to add items to the underlying collection when creating the object graph.

#### NOTE

The generic `List` and `Dictionary` interfaces (`IList<T>` and `IDictionary< TKey, TValue >`) are not supported for collection detection by the WPF XAML processor. However, you can use the `List<T>` class as a base class, because it implements `IList` directly, or `Dictionary< TKey, TValue >` as a base class, because it implements `IDictionary` directly.

When you declare a property that takes a collection, be cautious about how that property value is initialized in new instances of the type. If you are not implementing the property as a dependency property, then having the property use a backing field that calls the collection type constructor is adequate. If your property is a dependency property, then you may need to initialize the collection property as part of the default type constructor. This is because a dependency property takes its default value from metadata, and you typically do not want the initial value of a collection property to be a static, shared collection. There should be a collection instance per each containing type instance. For more information, see [Custom Dependency Properties](#).

You can implement a custom collection type for your collection property. Because of implicit collection property treatment, the custom collection type does not need to provide a parameterless constructor in order to be used in XAML implicitly. However, you can optionally provide a parameterless constructor for the collection type. This can be a worthwhile practice. Unless you do provide a parameterless constructor, you cannot explicitly declare the collection as an object element. Some markup authors might prefer to see the explicit collection as a matter of markup style. Also, a parameterless constructor can simplify the initialization requirements when you create new objects that use your collection type as a property value.

## Declaring XAML Content Properties

The XAML language defines the concept of a XAML content property. Each class that is usable in object syntax can have exactly one XAML content property. To declare a property to be the XAML content property for your class, apply the `ContentPropertyAttribute` as part of the class definition. Specify the name of the intended XAML content property as the `Name` in the attribute. The property is specified as a string by name, not as a reflection construct such as  `PropertyInfo`.

You can specify a collection property to be the XAML content property. This results in a usage for that property whereby the object element can have one or more child elements, without any intervening collection object elements or property element tags. These elements are then treated as the value for the XAML content property and added to the backing collection instance.

Some existing XAML content properties use the property type of `object`. This enables a XAML content property that can take primitive values such as a `String` as well as taking a single reference object value. If you follow this model, your type is responsible for type determination as well as the handling of possible types. The typical reason for an `Object` content type is to support both a simple means of adding object content as a string (which receives a default presentation treatment), or an advanced means of adding object content that specifies a non-default presentation or additional data.

## Serializing XAML

For certain scenarios, such as if you are a control author, you may also want to assure that any object representation that can be instantiated in XAML can also be serialized back to equivalent XAML markup. Serialization requirements are not described in this topic. See [Control Authoring Overview](#) and [Element Tree and Serialization](#).

## See also

- [XAML Overview \(WPF\)](#)
- [Custom Dependency Properties](#)
- [Control Authoring Overview](#)
- [Base Elements Overview](#)
- [XAML Loading and Dependency Properties](#)

# Markup Extensions and WPF XAML

11/3/2019 • 10 minutes to read • [Edit Online](#)

This topic introduces the concept of markup extensions for XAML, including their syntax rules, purpose, and the class object model that underlies them. Markup extensions are a general feature of the XAML language and of the .NET implementation of XAML services. This topic specifically details markup extensions for use in WPF XAML.

## XAML Processors and Markup Extensions

Generally speaking, a XAML parser can either interpret an attribute value as a literal string that can be converted to a primitive, or convert it to an object by some means. One such means is by referencing a type converter; this is documented in the topic [TypeConverters and XAML](#). However, there are scenarios where different behavior is required. For example, a XAML processor can be instructed that a value of an attribute should not result in a new object in the object graph. Instead, the attribute should result in an object graph that makes a reference to an already constructed object in another part of the graph, or a static object. Another scenario is that a XAML processor can be instructed to use a syntax that provides non-default arguments to the constructor of an object. These are the types of scenarios where a markup extension can provide the solution.

## Basic Markup Extension Syntax

A markup extension can be implemented to provide values for properties in an attribute usage, properties in a property element usage, or both.

When used to provide an attribute value, the syntax that distinguishes a markup extension sequence to a XAML processor is the presence of the opening and closing curly braces ({ and }). The type of markup extension is then identified by the string token immediately following the opening curly brace.

When used in property element syntax, a markup extension is visually the same as any other element used to provide a property element value: a XAML element declaration that references the markup extension class as an element, enclosed within angle brackets (<>).

## XAML-Defined Markup Extensions

Several markup extensions exist that are not specific to the WPF implementation of XAML, but are instead implementations of intrinsics or features of XAML as a language. These markup extensions are implemented in the System.Xaml assembly as part of the general .NET Framework XAML services, and are within the XAML language XAML namespace. In terms of common markup usage, these markup extensions are typically identifiable by the `x:` prefix in the usage. The [MarkupExtension](#) base class (also defined in System.Xaml) provides the pattern that all markup extensions should use in order to be supported in XAML readers and XAML writers, including in WPF XAML.

- `x:Type` supplies the [Type](#) object for the named type. This facility is used most frequently in styles and templates. For details, see [x:Type Markup Extension](#).
- `x:Static` produces static values. The values come from value-type code entities that are not directly the type of a target property's value, but can be evaluated to that type. For details, see [x:Static Markup Extension](#).
- `x:Null` specifies `null` as a value for a property and can be used either for attributes or property element values. For details, see [x:Null Markup Extension](#).

- `x:Array` provides support for creation of general arrays in XAML syntax, for cases where the collection support provided by WPF base elements and control models is deliberately not used. For details, see [x:Array Markup Extension](#).

#### NOTE

The `x:` prefix is used for the typical XAML namespace mapping of the XAML language intrinsics, in the root element of a XAML file or production. For example, the Visual Studio templates for WPF applications initiate a XAML file using this `x:` mapping. You could choose a different prefix token in your own XAML namespace mapping, but this documentation will assume the default `x:` mapping as a means of identifying those entities that are a defined part of the XAML namespace for the XAML language, as opposed to the WPF default namespace or other XAML namespaces not related to a specific framework.

## WPF-Specific Markup Extensions

The most common markup extensions used in WPF programming are those that support resource references (`StaticResource` and `DynamicResource`), and those that support data binding (`Binding`).

- `StaticResource` provides a value for a property by substituting the value of an already defined resource. A `StaticResource` evaluation is ultimately made at XAML load time and does not have access to the object graph at run time. For details, see [StaticResource Markup Extension](#).
- `DynamicResource` provides a value for a property by deferring that value to be a run-time reference to a resource. A dynamic resource reference forces a new lookup each time that such a resource is accessed and has access to the object graph at run time. In order to get this access, `DynamicResource` concept is supported by dependency properties in the WPF property system, and evaluated expressions. Therefore you can only use `DynamicResource` for a dependency property target. For details, see [DynamicResource Markup Extension](#).
- `Binding` provides a data bound value for a property, using the data context that applies to the parent object at run time. This markup extension is relatively complex, because it enables a substantial inline syntax for specifying a data binding. For details, see [Binding Markup Extension](#).
- `RelativeSource` provides source information for a `Binding` that can navigate several possible relationships in the run-time object tree. This provides specialized sourcing for bindings that are created in multi-use templates or created in code without full knowledge of the surrounding object tree. For details, see [RelativeSource MarkupExtension](#).
- `TemplateBinding` enables a control template to use values for templated properties that come from object-model-defined properties of the class that will use the template. In other words, the property within the template definition can access a context that only exists once the template is applied. For details, see [TemplateBinding Markup Extension](#). For more information on the practical use of `TemplateBinding`, see [Styling with ControlTemplates Sample](#).
- `ColorConvertedBitmap` supports a relatively advanced imaging scenario. For details, see [ColorConvertedBitmap Markup Extension](#).
- `ComponentResourceKey` and `ThemeDictionary` support aspects of resource lookup, particularly for resources and themes that are packaged with custom controls. For more information, see [ComponentResourceKey Markup Extension](#), [ThemeDictionary Markup Extension](#), or [Control Authoring Overview](#).

## \*Extension Classes

For both the general XAML language and WPF-specific markup extensions, the behavior of each markup

extension is identified to a XAML processor through a `*Extension` class that derives from `MarkupExtension`, and provides an implementation of the `ProvideValue` method. This method on each extension provides the object that is returned when the markup extension is evaluated. The returned object is typically evaluated based on the various string tokens that are passed to the markup extension.

For example, the `StaticResourceExtension` class provides the surface implementation of actual resource lookup so that its `ProvideValue` implementation returns the object that is requested, with the input of that particular implementation being a string that is used to look up the resource by its `x:key`. Much of this implementation detail is unimportant if you are using an existing markup extension.

Some markup extensions do not use string token arguments. This is either because they return a static or consistent value, or because context for what value should be returned is available through one of the services passed through the `serviceProvider` parameter.

The `*Extension` naming pattern is for convenience and consistency. It is not necessary in order for a XAML processor to identify that class as support for a markup extension. So long as your codebase includes `System.Xaml` and uses .NET Framework XAML Services implementations, all that is necessary to be recognized as a XAML markup extension is to derive from `MarkupExtension` and to support a construction syntax. WPF defines markup extension-enabling classes that do not follow the `*Extension` naming pattern, for example `Binding`. Typically the reason for this is that the class supports scenarios beyond pure markup extension support. In the case of `Binding`, that class supports run-time access to methods and properties of the object for scenarios that have nothing to do with XAML.

### Extension Class Interpretation of Initialization Text

The string tokens following the markup extension name and still within the braces are interpreted by a XAML processor in one of the following ways:

- A comma always represents the separator or delimiter of individual tokens.
- If the individual separated tokens do not contain any equals signs, each token is treated as a constructor argument. Each constructor parameter must be given as the type expected by that signature, and in the proper order expected by that signature.

#### NOTE

A XAML processor must call the constructor that matches the argument count of the number of pairs. For this reason, if you are implementing a custom markup extension, do not provide multiple constructors with the same argument count. The behavior for how a XAML processor behaves if more than one markup extension constructor path with the same parameter count exists is not defined, but you should anticipate that a XAML processor is permitted to throw an exception on usage if this situation exists in the markup extension type definitions.

- If the individual separated tokens contain equals signs, then a XAML processor first calls the parameterless constructor for the markup extension. Then, each `name=value` pair is interpreted as a property name that exists on the markup extension, and a value to assign to that property.
- If there is a parallel result between the constructor behavior and the property setting behavior in a markup extension, it does not matter which behavior you use. It is more common usage to use the `property = value` pairs for markup extensions that have more than one settable property, if only because it makes your markup more intentional and you are less likely to accidentally transpose constructor parameters. (When you specify `property=value` pairs, those properties may be in any order.) Also, there is no guarantee that a markup extension supplies a constructor parameter that sets every one of its settable properties. For example, `Binding` is a markup extension, with many properties that are settable through the extension in `property = value` form, but `Binding` only supports two constructors: a parameterless constructor, and one that sets an initial path.

- A literal comma cannot be passed to a markup extension without escapement.

## Escape Sequences and Markup Extensions

Attribute handling in a XAML processor uses the curly braces as indicators of a markup extension sequence. It is also possible to produce a literal curly brace character attribute value if necessary, by entering an escape sequence using an empty curly brace pair followed by the literal curly brace. See [{} Escape Sequence - Markup Extension](#).

## Nesting Markup Extensions in XAML Usage

Nesting of multiple markup extensions is supported, and each markup extension will be evaluated deepest first. For example, consider the following usage:

```
<Setter Property="Background"
Value="{DynamicResource {x:Static SystemColors.ControlBrushKey}}" />
```

In this usage, the `x:Static` statement is evaluated first and returns a string. That string is then used as the argument for `DynamicResource`.

## Markup Extensions and Property Element Syntax

When used as an object element that fills a property element value, a markup extension class is visually indistinguishable from a typical type-backed object element that can be used in XAML. The practical difference between a typical object element and a markup extension is that the markup extension is either evaluated to a typed value or deferred as an expression. Therefore the mechanisms for any possible type errors of property values for the markup extension will be different, similar to how a late-bound property is treated in other programming models. An ordinary object element will be evaluated for type match against the target property it is setting when the XAML is parsed.

Most markup extensions, when used in object element syntax to fill a property element, would not have content or any further property element syntax within. Thus you would close the object element tag, and provide no child elements. Whenever any object element is encountered by a XAML processor, the constructor for that class is called, which instantiates the object created from the parsed element. A markup extension class is no different: if you want your markup extension to be usable in object element syntax, you must provide a parameterless constructor. Some existing markup extensions have at least one required property value that must be specified for effective initialization. If so, that property value is typically given as a property attribute on the object element. In the [XAML Namespace \(x\) Language Features](#) and [WPF XAML Extensions](#) reference pages, markup extensions that have required properties (and the names of required properties) will be noted. Reference pages will also note if either object element syntax or attribute syntax is disallowed for particular markup extensions. A notable case is [x:Array Markup Extension](#), which cannot support attribute syntax because the contents of that array must be specified within the tagging as content. The array contents are handled as general objects, therefore no default type converter for the attribute is feasible. Also, [x:Array Markup Extension](#) requires a `type` parameter.

## See also

- [XAML Overview \(WPF\)](#)
- [XAML Namespace \(x\) Language Features](#)
- [WPF XAML Extensions](#)
- [StaticResource Markup Extension](#)
- [Binding Markup Extension](#)

- [DynamicResource Markup Extension](#)
- [x:Type Markup Extension](#)

# XAML Namespaces and Namespace Mapping for WPF XAML

11/12/2019 • 7 minutes to read • [Edit Online](#)

This topic further explains the presence and purpose of the two XAML namespace mappings as often found in the root tag of a WPF XAML file. It also describes how to produce similar mappings for using elements that are defined in your own code, and/or within separate assemblies.

## What is a XAML Namespace?

A XAML namespace is really an extension of the concept of an XML namespace. The techniques of specifying a XAML namespace rely on the XML namespace syntax, the convention of using URIs as namespace identifiers, using prefixes to provide a means to reference multiple namespaces from the same markup source, and so on. The primary concept that is added to the XAML definition of the XML namespace is that a XAML namespace implies both a scope of uniqueness for the markup usages, and also influences how markup entities are potentially backed by specific CLR namespaces and referenced assemblies. This latter consideration is also influenced by the concept of a XAML schema context. But for purposes of how WPF works with XAML namespaces, you can generally think of XAML namespaces in terms of a default XAML namespace, the XAML language namespace, and any further XAML namespaces as mapped by your XAML markup directly to specific backing CLR namespaces and referenced assemblies.

## The WPF and XAML Namespace Declarations

Within the namespace declarations in the root tag of many XAML files, you will see that there are typically two XML namespace declarations. The first declaration maps the overall WPF client / framework XAML namespace as the default:

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

The second declaration maps a separate XAML namespace, mapping it (typically) to the `x:` prefix.

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

The relationship between these declarations is that the `x:` prefix mapping supports the intrinsics that are part of the XAML language definition, and WPF is one implementation that uses XAML as a language and defines a vocabulary of its objects for XAML. Because the WPF vocabulary's usages will be far more common than the XAML intrinsics usages, the WPF vocabulary is mapped as the default.

The `x:` prefix convention for mapping the XAML language intrinsics support is followed by project templates, sample code, and the documentation of language features within this SDK. The XAML namespace defines many commonly-used features that are necessary even for basic WPF applications. For instance, in order to join any code-behind to a XAML file through a partial class, you must name that class as the `x:Class` attribute in the root element of the relevant XAML file. Or, any element as defined in a XAML page that you wish to access as a keyed resource should have the `x:Key` attribute set on the element in question. For more information on these and other aspects of XAML see [XAML Overview \(WPF\)](#) or [XAML Syntax In Detail](#).

## Mapping to Custom Classes and Assemblies

You can map XML namespaces to assemblies using a series of tokens within an `xmlns` prefix declaration, similar to how the standard WPF and XAML-intrinsics XAML namespaces are mapped to prefixes.

The syntax takes the following possible named tokens and following values:

`clr-namespace`: The CLR namespace declared within the assembly that contains the public types to expose as elements.

`assembly=` The assembly that contains some or all of the referenced CLR namespace. This value is typically just the name of the assembly, not the path, and does not include the extension (such as .dll or .exe). The path to that assembly must be established as a project reference in the project file that contains the XAML you are trying to map. In order to incorporate versioning and strong-name signing, the `assembly` value can be a string as defined by [AssemblyName](#), rather than the simple string name.

Note that the character separating the `clr-namespace` token from its value is a colon (:) whereas the character separating the `assembly` token from its value is an equals sign (=). The character to use between these two tokens is a semicolon. Also, do not include any white space anywhere in the declaration.

## A Basic Custom Mapping Example

The following code defines an example custom class:

```
namespace SDKSample {  
    public class ExampleClass : ContentControl {  
        public ExampleClass() {  
            ...  
        }  
    }  
}
```

```
Namespace SDKSample  
    Public Class ExampleClass  
        Inherits ContentControl  
        ...  
        Public Sub New()  
        End Sub  
    End Class  
End Namespace
```

This custom class is then compiled into a library, which per the project settings (not shown) is named `SDKSampleLibrary`.

In order to reference this custom class, you also need to include it as a reference for your current project, which you would typically do using the Solution Explorer UI in Visual Studio.

Now that you have a library containing a class, and a reference to it in project settings, you can add the following prefix mapping as part of your root element in XAML:

```
xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary"
```

To put it all together, the following is XAML that includes the custom mapping along with the typical default and x: mappings in the root tag, then uses a prefixed reference to instantiate `ExampleClass` in that UI:

```
<Page x:Class="WPFAplication1.MainPage"  
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
      xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary">  
    ...  
    <custom:ExampleClass/>  
    ...  
</Page>
```

## Mapping to Current Assemblies

`assembly` can be omitted if the `clr-namespace` referenced is being defined within the same assembly as the application code that is referencing the custom classes. Or, an equivalent syntax for this case is to specify `assembly=`, with no string token following the equals sign.

Custom classes cannot be used as the root element of a page if defined in the same assembly. Partial classes do not need to be mapped; only classes that are not the partial class of a page in your application need to be mapped if you intend to reference them as elements in XAML.

## Mapping CLR Namespaces to XML Namespaces in an Assembly

WPF defines a CLR attribute that is consumed by XAML processors in order to map multiple CLR namespaces to a single XAML namespace. This attribute, [XmlAttribute](#), is placed at the assembly level in the source code that produces the assembly. The WPF assembly source code uses this attribute to map the various common namespaces, such as [System.Windows](#) and [System.Windows.Controls](#), to the <http://schemas.microsoft.com/winfx/2006/xaml/presentation> namespace.

The [XmlAttribute](#) takes two parameters: the XML/XAML namespace name, and the CLR namespace name. More than one [XmlAttribute](#) can exist to map multiple CLR namespaces to the same XML namespace. Once mapped, members of those namespaces can also be referenced without full qualification if desired by providing the appropriate `using` statement in the partial-class code-behind page. For more details, see [XmlAttribute](#).

## Designer Namespaces and Other Prefixes From XAML Templates

If you are working with development environments and/or design tools for WPF XAML, you may notice that there are other defined XAML namespaces / prefixes within the XAML markup.

WPF Designer for Visual Studio uses a designer namespace that is typically mapped to the prefix `d:`. More recent project templates for WPF might pre-map this XAML namespace to support interchange of the XAML between WPF Designer for Visual Studio and other design environments. This design XAML namespace is used to perpetuate design state while roundtripping XAML-based UI in the designer. It is also used for features such as `d:IsDataSource`, which enable runtime data sources in a designer.

Another prefix you might see mapped is `mc:`. `mc:` is for markup compatibility, and is leveraging a markup compatibility pattern that is not necessarily XAML-specific. To some extent, the markup compatibility features can be used to exchange XAML between frameworks or across other boundaries of backing implementation, work between XAML schema contexts, provide compatibility for limited modes in designers, and so on. For more information on markup compatibility concepts and how they relate to WPF, see [Markup Compatibility \(mc\) Language Features](#).

## WPF and Assembly Loading

The XAML schema context for WPF integrates with the WPF application model, which in turn uses the CLR-defined concept of [AppDomain](#). The following sequence describes how XAML schema context interprets how to either load assemblies or find types at run time or design time, based on the WPF use of [AppDomain](#) and other factors.

1. Iterate through the [AppDomain](#), looking for an already-loaded assembly that matches all aspects of the name, starting from the most recently loaded assembly.
2. If the name is qualified, call [Assembly.Load\(String\)](#) on the qualified name.
3. If the short name + public key token of a qualified name matches the assembly that the markup was loaded from, return that assembly.

4. Use the short name + public key token to call [Assembly.Load\(String\)](#).

5. If the name is unqualified, call [Assembly.LoadWithPartialName](#).

Loose XAML does not use Step 3; there is no loaded-from assembly.

Compiled XAML for WPF (generated via XamlBuildTask) does not use the already-loaded assemblies from [AppDomain](#) (Step 1). Also, the name should never be unqualified from XamlBuildTask output, so Step 5 does not apply.

Compiled BAML (generated via PresentationBuildTask) uses all steps, although BAML also should not contain unqualified assembly names.

## See also

- [Understanding XML Namespaces](#)
- [XAML Overview \(WPF\)](#)

# WPF XAML Namescopes

8/22/2019 • 7 minutes to read • [Edit Online](#)

XAML namespaces are a concept that identifies objects that are defined in XAML. The names in a XAML namespace can be used to establish relationships between the XAML-defined names of objects and their instance equivalents in an object tree. Typically, XAML namespaces in WPF managed code are created when loading the individual XAML page roots for a XAML application. XAML namespaces as the programming object are defined by the [INamespace](#) interface and are also implemented by the practical class [NameScope](#).

## Namescopes in Loaded XAML Applications

In a broader programming or computer science context, programming concepts often include the principle of a unique identifier or name that can be used to access an object. For systems that use identifiers or names, the namespace defines the boundaries within which a process or technique will search if an object of that name is requested, or the boundaries wherein uniqueness of identifying names is enforced. These general principles are true for XAML namespaces. In WPF, XAML namespaces are created on the root element for a XAML page when the page is loaded. Each name specified within the XAML page starting at the page root is added to a pertinent XAML namespace.

In WPF XAML, elements that are common root elements (such as [Page](#), and [Window](#)) always control a XAML namespace. If an element such as [FrameworkElement](#) or [FrameworkContentElement](#) is the root element of the page in markup, a XAML processor adds a [Page](#) root implicitly so that the [Page](#) can provide a working XAML namespace.

### NOTE

WPF build actions create a XAML namespace for a XAML production even if no `Name` or `x:Name` attributes are defined on any elements in the XAML markup.

If you try to use the same name twice in any XAML namespace, an exception is raised. For WPF XAML that has code-behind and is part of a compiled application, the exception is raised at build time by WPF build actions, when creating the generated class for the page during the initial markup compile. For XAML that is not markup-compiled by any build action, exceptions related to XAML namespace issues might be raised when the XAML is loaded. XAML designers might also anticipate XAML namespace issues at design time.

### Adding Objects to Runtime Object Trees

The moment that XAML is parsed represents the moment in time that a WPF XAML namespace is created and defined. If you add an object to an object tree at a point in time after the XAML that produced that tree was parsed, a `Name` or `x:Name` value on the new object does not automatically update the information in a XAML namespace. To add a name for an object into a WPF XAML namespace after XAML is loaded, you must call the appropriate implementation of [RegisterName](#) on the object that defines the XAML namespace, which is typically the XAML page root. If the name is not registered, the added object cannot be referenced by name through methods such as [FindName](#), and you cannot use that name for animation targeting.

The most common scenario for application developers is that you will use [RegisterName](#) to register names into the XAML namespace on the current root of the page. [RegisterName](#) is part of an important scenario for storyboards that target objects for animations. For more information, see [Storyboards Overview](#).

If you call [RegisterName](#) on an object other than the object that defines the XAML namespace, the name is still registered to the XAML namespace that the calling object is held within, as if you had called [RegisterName](#) on the

XAML namespace defining object.

## XAML Namespaces in Code

You can create and then use XAML namespaces in code. The APIs and the concepts involved in XAML namespace creation are the same even for a pure code usage, because the XAML processor for WPF uses these APIs and concepts when it processes XAML itself. The concepts and API exist mainly for the purpose of being able to find objects by name within an object tree that is typically defined partially or entirely in XAML.

For applications that are created programmatically, and not from loaded XAML, the object that defines a XAML namespace must implement [INamespace](#), or be a [FrameworkElement](#) or [FrameworkContentElement](#) derived class, in order to support creation of a XAML namespace on its instances.

Also, for any element that is not loaded and processed by a XAML processor, the XAML namespace for the object is not created or initialized by default. You must explicitly create a new XAML namespace for any object that you intend to register names into subsequently. To create a XAML namespace, you call the static [SetNameScope](#) method. Specify the object that will own it as the `dependencyObject` parameter, and a new [NameScope](#) constructor call as the `value` parameter.

If the object provided as `dependencyObject` for [SetNameScope](#) is not a [INamespace](#) implementation, [FrameworkElement](#) or [FrameworkContentElement](#), calling [RegisterName](#) on any child elements will have no effect. If you fail to create the new XAML namespace explicitly, then calls to [RegisterName](#) will raise an exception.

For an example of using XAML namespace APIs in code, see [Define a Name Scope](#).

## XAML Namescopes in Styles and Templates

Styles and templates in WPF provide the ability to reuse and reapply content in a straightforward way. However, styles and templates might also include elements with XAML names defined at the template level. That same template might be used multiple times in a page. For this reason, styles and templates both define their own XAML namespaces, independent of whatever location in an object tree where the style or template is applied.

Consider the following example:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <Page.Resources>
        <ControlTemplate x:Key="MyButtonTemplate" TargetType="{x:Type Button}">
            <Border BorderBrush="Red" Name="TheBorder" BorderThickness="2">
                <ContentPresenter/>
            </Border>
        </ControlTemplate>
    </Page.Resources>
    <StackPanel>
        <Button Template="{StaticResource MyButtonTemplate}">My first button</Button>
        <Button Template="{StaticResource MyButtonTemplate}">My second button</Button>
    </StackPanel>
</Page>
```

Here, the same template is applied to two different buttons. If templates did not have discrete XAML namespaces, the `TheBorder` name used in the template would cause a name collision in the XAML namespace. Each instantiation of the template has its own XAML namespace, so in this example each instantiated template's XAML namespace would contain exactly one name.

Styles also define their own XAML namespace, mostly so that parts of storyboards can have particular names assigned. These names enable control specific behaviors that will target elements of that name, even if the template was re-defined as part of control customization.

Because of the separate XAML namespaces, finding named elements in a template is more challenging than finding a non-templated named element in a page. You first need to determine the applied template, by getting the [Template](#) property value of the control where the template is applied. Then, you call the template version of [FindName](#), passing the control where the template was applied as the second parameter.

If you are a control author and you are generating a convention where a particular named element in an applied template is the target for a behavior that is defined by the control itself, you can use the [GetTemplateChild](#) method from your control implementation code. The [GetTemplateChild](#) method is protected, so only the control author has access to it.

If you are working from within a template, and need to get to the XAML namespace where the template is applied, get the value of [TemplatedParent](#), and then call [FindName](#) there. An example of working within the template would be if you are writing the event handler implementation where the event will be raised from an element in an applied template.

## XAML Namescopes and Name-related APIs

[FrameworkElement](#) has [FindName](#), [RegisterName](#) and [UnregisterName](#) methods. If the object you call these methods on owns a XAML namespace, the methods call into the methods of the relevant XAML namespace. Otherwise, the parent element is checked to see if it owns a XAML namespace, and this process continues recursively until a XAML namespace is found (because of the XAML processor behavior, there is guaranteed to be a XAML namespace at the root). [FrameworkContentElement](#) has analogous behaviors, with the exception that no [FrameworkContentElement](#) will ever own a XAML namespace. The methods exist on [FrameworkContentElement](#) so that the calls can be forwarded eventually to a [FrameworkElement](#) parent element.

[SetNameScope](#) is used to map a new XAML namespace to an existing object. You can call [SetNameScope](#) more than once in order to reset or clear the XAML namespace, but that is not a common usage. Also, [GetNameScope](#) is not typically used from code.

### XAML Namespace Implementations

The following classes implement [INamescope](#) directly:

- [NameScope](#)
- [Style](#)
- [ResourceDictionary](#)
- [FrameworkTemplate](#)

[ResourceDictionary](#) does not use XAML names or namespaces ; it uses keys instead, because it is a dictionary implementation. The only reason that [ResourceDictionary](#) implements [INamescope](#) is so it can raise exceptions to user code that help clarify the distinction between a true XAML namespace and how a [ResourceDictionary](#) handles keys, and also to assure that XAML namespaces are not applied to a [ResourceDictionary](#) by parent elements.

[FrameworkTemplate](#) and [Style](#) implement [INamescope](#) through explicit interface definitions. The explicit implementations allow these XAML namespaces to behave conventionally when they are accessed through the [INamescope](#) interface, which is how XAML namespaces are communicated by WPF internal processes. But the explicit interface definitions are not part of the conventional API surface of [FrameworkTemplate](#) and [Style](#), because you seldom need to call the [INamescope](#) methods on [FrameworkTemplate](#) and [Style](#) directly, and instead would use other API such as [GetTemplateChild](#).

The following classes define their own XAML namespace, by using the [System.Windows.NameScope](#) helper class and connecting to its XAML namespace implementation through the [NameScope.NameScope](#) attached property:

- [FrameworkElement](#)
- [FrameworkContentElement](#)

## See also

- [XAML Namespaces and Namespace Mapping for WPF XAML](#)
- [x:Name Directive](#)

# Inline Styles and Templates

11/3/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides [Style](#) objects and template objects ([FrameworkTemplate](#) subclasses) as a way to define the visual appearance of an element in resources, so that they can be used multiple times. For this reason, attributes in XAML that take the types [Style](#) and [FrameworkTemplate](#) almost always make resource references to existing styles and templates rather than define new ones inline.

## Limitations of Inline Styles and Templates

In Extensible Application Markup Language (XAML), style and template properties can technically be set in one of two ways. You can use attribute syntax to reference a style that was defined within a resource, for example `< object Style="{StaticResource myResourceKey}" .../>`. Or you can use property element syntax to define a style inline, for instance:

```
< object >  
< object .style>  
< Style .../>  
</ object .style>  
</ object >
```

The attribute usage is much more common. A style that is defined inline and not defined in resources is necessarily scoped to the containing element only, and cannot be re-used as easily because it has no resource key. In general a resource-defined style is more versatile and useful, and is more in keeping with the general Windows Presentation Foundation (WPF) programming model principle of separating program logic in code from design in markup.

Usually there is no reason to set a style or template inline, even if you only intend to use that style or template in that location. Most elements that can take a style or template also support a content property and a content model. If you are only using whatever logical tree you create through styling or templating once, it would be even easier to just fill that content property with the equivalent child elements in direct markup. This would bypass the style and template mechanisms altogether.

Other syntaxes enabled by markup extensions that return an object are also possible for styles and templates. Two such extensions that have possible scenarios include [TemplateBinding](#) and [Binding](#).

## See also

- [Styling and Templating](#)

# TypeConverters and XAML

11/3/2019 • 9 minutes to read • [Edit Online](#)

This topic introduces the purpose of type conversion from string as a general XAML language feature. In the .NET Framework, the [TypeConverter](#) class serves a particular purpose as part of the implementation for a managed custom class that can be used as a property value in XAML attribute usage. If you write a custom class, and you want instances of your class to be usable as XAML settable attribute values, you might need to apply a [TypeConverterAttribute](#) to your class, write a custom [TypeConverter](#) class, or both.

## Type Conversion Concepts

### XAML and String Values

When you set an attribute value in a XAML file, the initial type of that value is a string in pure text. Even other primitives such as [Double](#) are initially text strings to a XAML processor.

A XAML processor needs two pieces of information in order to process an attribute value. The first piece of information is the value type of the property that is being set. Any string that defines an attribute value and that is processed in XAML must ultimately be converted or resolved to a value of that type. If the value is a primitive that is understood by the XAML parser (such as a numeric value), a direct conversion of the string is attempted. If the value is an enumeration, the string is used to check for a name match to a named constant in that enumeration. If the value is neither a parser-understood primitive nor an enumeration, then the type in question must be able to provide an instance of the type, or a value, based on a converted string. This is done by indicating a type converter class. The type converter is effectively a helper class for providing values of another class, both for the XAML scenario and also potentially for code calls in .NET code.

### Using Existing Type Conversion Behavior in XAML

Depending on your familiarity with the underlying XAML concepts, you may already be using type conversion behavior in basic application XAML without realizing it. For instance, WPF defines literally hundreds of properties that take a value of type [Point](#). A [Point](#) is a value that describes a coordinate in a two-dimensional coordinate space, and it really just has two important properties: [X](#) and [Y](#). When you specify a point in XAML, you specify it as a string with a delimiter (typically a comma) between the [X](#) and [Y](#) values you provide. For example:

```
<LinearGradientBrush StartPoint="0,0" EndPoint="1,1"/>.
```

Even this simple type of [Point](#) and its simple usage in XAML involve a type converter. In this case that is the class [PointConverter](#).

The type converter for [Point](#) defined at the class level streamlines the markup usages of all properties that take [Point](#). Without a type converter here, you would need the following much more verbose markup for the same example shown previously:

```
<LinearGradientBrush>
  <LinearGradientBrush.StartPoint>
    <Point X="0" Y="0"/>
  </LinearGradientBrush.StartPoint>
  <LinearGradientBrush.EndPoint>
    <Point X="1" Y="1"/>
  </LinearGradientBrush.EndPoint>
</LinearGradientBrush>
```

Whether to use the type conversion string or a more verbose equivalent syntax is generally a coding style choice. Your XAML tooling workflow might also influence how values are set. Some XAML tools tend to emit the most

verbose form of the markup because it is easier to round-trip to designer views or its own serialization mechanism.

Existing type converters can generally be discovered on WPF and .NET Framework types by checking a class (or property) for the presence of an applied [TypeConverterAttribute](#). This attribute will name the class that is the supporting type converter for values of that type, for XAML purposes as well as potentially other purposes.

### Type Converters and Markup Extensions

Markup extensions and type converters fill orthogonal roles in terms of XAML processor behavior and the scenarios that they are applied to. Although context is available for markup extension usages, type conversion behavior of properties where a markup extension provides a value is generally not checked in the markup extension implementations. In other words, even if a markup extension returns a text string as its `ProvideValue` output, type conversion behavior on that string as applied to a specific property or property value type is not invoked. Generally, the purpose of a markup extension is to process a string and return an object without any type converter involved.

One common situation where a markup extension is necessary rather than a type converter is to make a reference to an object that already exists. At best, a stateless type converter could only generate a new instance, which might not be desirable. For more information on markup extensions, see [Markup Extensions and WPF XAML](#).

### Native Type Converters

In the WPF and .NET Framework implementation of the XAML parser, there are certain types that have native type conversion handling, yet are not types that might conventionally be thought of as primitives. An example of such a type is [DateTime](#). The reason for this is based on how the .NET Framework architecture works: the type [DateTime](#) is defined in mscorelib, the most basic library in .NET. [DateTime](#) is not permitted to be attributed with an attribute that comes from another assembly that introduces a dependency ([TypeConverterAttribute](#) is from System) so the usual type converter discovery mechanism by attributing cannot be supported. Instead, the XAML parser has a list of types that need such native processing and processes these similarly to how the true primitives are processed. (In the case of [DateTime](#) this involves a call to [Parse](#).)

## Implementing a Type Converter

### TypeConverter

In the [Point](#) example given previously, the class [PointConverter](#) was mentioned. For .NET implementations of XAML, all type converters that are used for XAML purposes are classes that derive from the base class [TypeConverter](#). The [TypeConverter](#) class existed in versions of .NET Framework that precede the existence of XAML; one of its original usages was to provide string conversion for property dialogs in visual designers. For XAML, the role of [TypeConverter](#) is expanded to include being the base class for to-string and from-string conversions that enable parsing a string attribute value, and possibly processing a run-time value of a particular object property back into a string for serialization as an attribute.

[TypeConverter](#) defines four members that are relevant for converting to and from strings for XAML processing purposes:

- [CanConvertTo](#)
- [CanConvertFrom](#)
- [ConvertTo](#)
- [ConvertFrom](#)

Of these, the most important method is [ConvertFrom](#). This method converts the input string to the required object type. Strictly speaking, the [ConvertFrom](#) method could be implemented to convert a much wider range of types into the converter's intended destination type, and thus serve purposes that extend beyond XAML such as supporting run-time conversions, but for XAML purposes it is only the code path that can process a [String](#) input

that matters.

The next most important method is [ConvertTo](#). If an application is converted to a markup representation (for instance, if it is saved to XAML as a file), [ConvertTo](#) is responsible for producing a markup representation. In this case, the code path that matters for XAML is when you pass a `destinationType` of [String](#).

[CanConvertTo](#) and [CanConvertFrom](#) are support methods that are used when a service queries the capabilities of the [TypeConverter](#) implementation. You must implement these methods to return `true` for type-specific cases that the equivalent conversion methods of your converter support. For XAML purposes, this generally means the [String](#) type.

## Culture Information and Type Converters for XAML

Each [TypeConverter](#) implementation can have its own interpretation of what constitutes a valid string for a conversion, and can also use or ignore the type description passed as parameters. There is an important consideration with regard to culture and XAML type conversion. Using localizable strings as attribute values is entirely supported by XAML. But using that localizable string as type converter input with specific culture requirements is not supported, because type converters for XAML attribute values involve a necessarily fixed-language parsing behavior, using `en-US` culture. For more information on the design reasons for this restriction, you should consult the XAML language specification ([\[MS-XAML\]](#)).

As an example where culture can be an issue, some cultures use a comma as their decimal point delimiter for numbers. This will collide with the behavior that many of the WPF XAML type converters have, which is to use a comma as a delimiter (based on historical precedents such as the common X,Y form, or comma delimited lists). Even passing a culture in the surrounding XAML (setting `Language` or `xml:lang` to the `s1-SI` culture, an example of a culture that uses a comma for decimal in this way) does not solve the issue.

## Implementing ConvertFrom

To be usable as a [TypeConverter](#) implementation that supports XAML, the [ConvertFrom](#) method for that converter must accept a string as the `value` parameter. If the string was in valid format, and can be converted by the [TypeConverter](#) implementation, then the returned object must support a cast to the type expected by the property. Otherwise, the [ConvertFrom](#) implementation must return `null`.

Each [TypeConverter](#) implementation can have its own interpretation of what constitutes a valid string for a conversion, and can also use or ignore the type description or culture contexts passed as parameters. However, the WPF XAML processing might not pass values to the type description context in all cases, and also might not pass culture based on `xml:lang`.

### NOTE

Do not use the curly brace characters, particularly {}, as a possible element of your string format. These characters are reserved as the entry and exit for a markup extension sequence.

## Implementing ConvertTo

[ConvertTo](#) is potentially used for serialization support. Serialization support through [ConvertTo](#) for your custom type and its type converter is not an absolute requirement. However, if you are implementing a control, or using serialization as part of the features or design of your class, you should implement [ConvertTo](#).

To be usable as a [TypeConverter](#) implementation that supports XAML, the [ConvertTo](#) method for that converter must accept an instance of the type (or a value) being supported as the `value` parameter. When the `destinationType` parameter is the type [String](#), then the returned object must be able to be cast as [String](#). The returned string must represent a serialized value of `value`. Ideally, the serialization format you choose should be capable of generating the same value if that string were passed to the [ConvertFrom](#) implementation of the same converter, without significant loss of information.

If the value cannot be serialized, or the converter does not support serialization, the [ConvertTo](#) implementation must return `null`, and is permitted to throw an exception in this case. But if you do throw exceptions, you should report the inability to use that conversion as part of your [CanConvertTo](#) implementation so that the best practice of checking with [CanConvertTo](#) first to avoid exceptions is supported.

If `destinationType` parameter is not of type [String](#), you can choose your own converter handling. Typically, you would revert to base implementation handling, which in the basemost [ConvertTo](#) raises a specific exception.

### Implementing [CanConvertTo](#)

Your [CanConvertTo](#) implementation should return `true` for `destinationType` of type [String](#), and otherwise defer to the base implementation.

### Implementing [CanConvertFrom](#)

Your [CanConvertFrom](#) implementation should return `true` for `sourceType` of type [String](#), and otherwise defer to the base implementation.

## Applying the [TypeConverterAttribute](#)

In order for your custom type converter to be used as the acting type converter for a custom class by a XAML processor, you must apply the [TypeConverterAttribute](#) to your class definition. The [ConverterTypeName](#) that you specify through the attribute must be the type name of your custom type converter. With this attribute applied, when a XAML processor handles values where the property type uses your custom class type, it can input strings and return object instances.

You can also provide a type converter on a per-property basis. Instead of applying a [TypeConverterAttribute](#) to the class definition, apply it to a property definition (the main definition, not the `get` / `set` implementations within it). The type of the property must match the type that is processed by your custom type converter. With this attribute applied, when a XAML processor handles values of that property, it can process input strings and return object instances. The per-property type converter technique is particularly useful if you choose to use a property type from Microsoft .NET Framework or from some other library where you cannot control the class definition and cannot apply a [TypeConverterAttribute](#) there.

## See also

- [TypeConverter](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)
- [XAML Syntax In Detail](#)

# WPF XAML Extensions

3/5/2019 • 2 minutes to read • [Edit Online](#)

## In This Section

- [Binding Markup Extension](#)
- [ColorConvertedBitmap Markup Extension](#)
- [ComponentResourceKey Markup Extension](#)
- [DynamicResource Markup Extension](#)
- [RelativeSource MarkupExtension](#)
- [StaticResource Markup Extension](#)
- [TemplateBinding Markup Extension](#)
- [ThemeDictionary Markup Extension](#)
- [PropertyPath XAML Syntax](#)
- [PresentationOptions:Freeze Attribute](#)

# Binding Markup Extension

11/3/2019 • 7 minutes to read • [Edit Online](#)

Defers a property value to be a data-bound value, creating an intermediate expression object and interpreting the data context that applies to the element and its binding at run time.

## Binding Expression Usage

```
<object property="{Binding}" .../>
-or-
<object property="{Binding bindProp1=value1[, bindPropN=valueN]*}" ...
/>
-or-
<object property="{Binding path}" .../>
-or
<object property="{Binding path[, bindPropN=valueN]*}" .../>
```

## Syntax Notes

In these syntaxes, the `[]` and `*` are not literals. They are part of a notation to indicate that zero or more `bindProp = value` pairs can be used, with a `,` separator between them and preceding `bindProp = value` pairs.

Any of the properties listed in the "Binding Properties That Can Be Set with the Binding Extension" section could instead be set using attributes of a `Binding` object element. However, that is not truly the markup extension usage of `Binding`, it is just the general XAML processing of attributes that set properties of the CLR `Binding` class. In other words, `<Binding bindProp1 = "value1" [ bindPropN = "valueN" ]* />` is an equivalent syntax for attributes of `Binding` object element usage instead of a `Binding` expression usage. To learn about the XAML attribute usage of specific properties of `Binding`, see the "XAML Attribute Usage" section of the relevant property of `Binding` in the .NET Framework Class Library.

## XAML Values

<code>bindProp1, bindPropN</code>	The name of the <code>Binding</code> or <code>BindingBase</code> property to set. Not all <code>Binding</code> properties can be set with the <code>Binding</code> extension, and some properties are settable within a <code>Binding</code> expression only by using further nested markup extensions. See "Binding Properties That Can Be Set with the Binding Extension" section.
<code>value1, valueN</code>	The value to set the property to. The handling of the attribute value is ultimately specific to the type and logic of the specific <code>Binding</code> property being set.
<code>path</code>	The path string that sets the implicit <code>Binding.Path</code> property. See also <a href="#">PropertyPath XAML Syntax</a> .

## Unqualified {Binding}

The `{Binding}` usage shown in "Binding Expression Usage" creates a `Binding` object with default values, which

includes an initial `Binding.Path` of `null`. This is still useful in many scenarios, because the created `Binding` might be relying on key data binding properties such as `Binding.Path` and `Binding.Source` being set in the run-time data context. For more information on the concept of data context, see [Data Binding](#).

## Implicit Path

The `Binding` markup extension uses `Binding.Path` as a conceptual "default property", where `Path=` does not need to appear in the expression. If you specify a `Binding` expression with an implicit path, the implicit path must appear first in the expression, prior to any other `bindProp = value` pairs where the `Binding` property is specified by name. For example: `{Binding PathString}`, where `PathString` is a string that is evaluated to be the value of `Binding.Path` in the `Binding` created by the markup extension usage. You can append an implicit path with other named properties after the comma separator, for example, `{Binding LastName, Mode=TwoWay}`.

## Binding Properties That Can Be Set with the Binding Extension

The syntax shown in this topic uses the generic `bindProp = value` approximation, because there are many read/write properties of `BindingBase` or `Binding` that can be set through the `Binding` markup extension / expression syntax. They can be set in any order, with the exception of an implicit `Binding.Path`. (You do have the option to explicitly specify `Path=`, in which case it can be set in any order). Basically, you can set zero or more of the properties in the list below, using `bindProp = value` pairs separated by commas.

Several of these property values require object types that do not support a native type conversion from a text syntax in XAML, and thus require markup extensions in order to be set as an attribute value. Check the XAML Attribute Usage section in the .NET Framework Class Library for each property for more information; the string you use for XAML attribute syntax with or without further markup extension usage is basically the same as the value you specify in a `Binding` expression, with the exception that you do not place quotation marks around each `bindProp = value` in the `Binding` expression.

- **BindingGroupName:** a string that identifies a possible binding group. This is a relatively advanced binding concept; see reference page for [BindingGroupName](#).
- **BindsDirectlyToSource:** Boolean, can be either `true` or `false`. The default is `false`.
- **Converter:** can be set as a `bindProp = value` string in the expression, but to do so requires an object reference for the value, such as a [StaticResource Markup Extension](#). The value in this case is an instance of a custom converter class.
- **ConverterCulture:** settable in the expression as a standards-based identifier; see the reference topic for [ConverterCulture](#).
- **ConverterParameter:** can be set as a `bindProp = value` string in the expression, but this is dependent on the type of the parameter being passed. If passing a reference type for the value, this usage requires an object reference such as a nested [StaticResource Markup Extension](#).
- **ElementName:** mutually exclusive versus [RelativeSource](#) and [Source](#); each of these binding properties represents a particular binding methodology. See [Data Binding Overview](#).
- **FallbackValue:** can be set as a `bindProp = value` string in the expression, but this is dependent on the type of the value being passed. If passing a reference type, requires an object reference such as a nested [StaticResource Markup Extension](#).
- **IsAsync:** Boolean, can be either `true` or `false`. The default is `false`.
- **Mode:** `value` is a constant name from the `BindingMode` enumeration. For example, `{Binding Mode=OneWay}`.
- **NotifyOnSourceUpdated:** Boolean, can be either `true` or `false`. The default is `false`.

- **NotifyOnTargetUpdated**: Boolean, can be either `true` or `false`. The default is `false`.
- **NotifyOnValidationError**: Boolean, can be either `true` or `false`. The default is `false`.
- **Path**: a string that describes a path into a data object or a general object model. The format provides several different conventions for traversing an object model that cannot be adequately described in this topic. See [PropertyPath XAML Syntax](#).
- **RelativeSource**: mutually exclusive versus with [ElementName](#) and [Source](#); each of these binding properties represents a particular binding methodology. See [Data Binding Overview](#). Requires a nested [RelativeSource MarkupExtension](#) usage to specify the value.
- **Source**: mutually exclusive versus [RelativeSource](#) and [ElementName](#); each of these binding properties represents a particular binding methodology. See [Data Binding Overview](#). Requires a nested extension usage, typically a [StaticResource Markup Extension](#) that refers to an object data source from a keyed resource dictionary.
- **StringFormat**: a string that describes a string format convention for the bound data. This is a relatively advanced binding concept; see reference page for [StringFormat](#).
- **TargetNullValue**: can be set as a `bindProp = value` string in the expression, but this is dependent on the type of the parameter being passed. If passing a reference type for the value, requires an object reference such as a nested [StaticResource Markup Extension](#).
- **UpdateSourceTrigger**: *value* is a constant name from the [UpdateSourceTrigger](#) enumeration. For example, `{Binding UpdateSourceTrigger=LostFocus}`. Specific controls potentially have different default values for this binding property. See [UpdateSourceTrigger](#).
- **ValidatesOnDataErrors**: Boolean, can be either `true` or `false`. The default is `false`. See Remarks.
- **ValidatesOnExceptions**: Boolean, can be either `true` or `false`. The default is `false`. See Remarks.
- **XPath**: a string that describes a path into the XMLDOM of an XML data source. See [Bind to XML Data Using an XMLDataProvider and XPath Queries](#).

The following are properties of [Binding](#) that cannot be set using the `Binding` markup extension/`{Binding}` expression form.

- **UpdateSourceExceptionFilter**: this property expects a reference to a callback implementation. Callbacks/methods other than event handlers cannot be referenced in XAML syntax.
- **ValidationRules**: the property takes a generic collection of [ValidationRule](#) objects. This could be expressed as a property element in a [Binding](#) object element, but has no readily available attribute-parsing technique for usage in a `Binding` expression. See reference topic for [ValidationRules](#).
- **XmlNamespaceManager**

## Remarks

### IMPORTANT

In terms of dependency property precedence, a `Binding` expression is equivalent to a locally set value. If you set a local value for a property that previously had a `Binding` expression, the `Binding` is completely removed. For details, see [Dependency Property Value Precedence](#).

Describing data binding at a basic level is not covered in this topic. See [Data Binding Overview](#).

#### NOTE

[MultiBinding](#) and [PriorityBinding](#) do not support a XAML extension syntax. You would instead use property elements. See reference topics for [MultiBinding](#) and [PriorityBinding](#).

Boolean values for XAML are case insensitive. For example you could specify either

`{Binding NotifyOnValidationError=true}` or `{Binding NotifyOnValidationError=True}`.

Bindings that involve data validation are typically specified by an explicit `Binding` element rather than as a `{Binding ...}` expression, and setting [ValidatesOnDataErrors](#) or [ValidatesOnExceptions](#) in an expression is uncommon. This is because the companion property [ValidationRules](#) cannot be readily set in the expression form. For more information, see [Implement Binding Validation](#).

`Binding` is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than type converters attributed on certain types or properties. All markup extensions in XAML use the `{` and `}` characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the string contents. For more information, see [Markup Extensions and WPF XAML](#).

`Binding` is an atypical markup extension in that the `Binding` class that implements the extension functionality for WPF's XAML implementation also implements several other methods and properties that are not related to XAML. The other members are intended to make `Binding` a more versatile and self-contained class that can address many data binding scenarios in addition to functioning as a XAML markup extension.

## See also

- [Binding](#)
- [Data Binding Overview](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)

# ColorConvertedBitmap Markup Extension

10/18/2019 • 2 minutes to read • [Edit Online](#)

Provides a way to specify a bitmap source that does not have an embedded profile. Color contexts / profiles are specified by URI, as is the image source URI.

## XAML Attribute Usage

```
<object property="{ColorConvertedBitmap imageSource sourceIIC destinationIIC}" .../>
```

## XAML Values

<code>imageSource</code>	The URI of the nonprofiled bitmap.
<code>sourceIIC</code>	The URI of the source profile configuration.
<code>destinationIIC</code>	The URI of the destination profile configuration

## Remarks

This markup extension is intended to fill a related set of image-source property values such as [UriSource](#).

Attribute syntax is the most common syntax used with this markup extension. `ColorConvertedBitmap` (or `ColorConvertedBitmapExtension`) cannot be used in property element syntax, because the values can only be set as values on the initial constructor, which is the string following the extension identifier.

`ColorConvertedBitmap` is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than just putting type converters on certain types or properties. All markup extensions in XAML use the { and } characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the attribute. For more information, see [Markup Extensions and WPF XAML](#).

## See also

- [UriSource](#)
- [Markup Extensions and WPF XAML](#)
- [Imaging Overview](#)

# ComponentResourceKey Markup Extension

11/3/2019 • 3 minutes to read • [Edit Online](#)

Defines and references keys for resources that are loaded from external assemblies. This enables a resource lookup to specify a target type in an assembly, rather than an explicit resource dictionary in an assembly or on a class.

## XAML Attribute Usage (setting key, compact)

```
<object x:Key="{ComponentResourceKey {x>Type targetTypeName}, targetID}" .../>
```

## XAML Attribute Usage (setting key, verbose)

```
<object x:Key="{ComponentResourceKey TypeInTargetAssembly={x>Type targetTypeName}, ResourceID=targetID}" ...>
```

## XAML Attribute Usage (requesting resource, compact)

```
<object property="{DynamicResource {ComponentResourceKey {x>Type targetTypeName}, targetID}}" .../>
```

## XAML Attribute Usage (requesting resource, verbose)

```
<object property="{DynamicResource {ComponentResourceKey TypeInTargetAssembly={x>Type targetTypeName}, ResourceID=targetID}}" .../>
```

## XAML Values

<code>targetTypeNames</code>	The name of the public common language runtime (CLR) type that is defined in the resource assembly.
<code>targetID</code>	The key for the resource. When resources are looked up, <code>targetID</code> will be analogous to the <a href="#">x:Key Directive</a> of the resource.

## Remarks

As seen in the usages above, a `{ ComponentResourceKey }` markup extension usage is found in two places:

- The definition of a key within a theme resource dictionary, as provided by a control author.
- Accessing a theme resource from the assembly, when you are retemplating the control but want to use property values that come from resources provided by the control's themes.

For referencing component resources that come from themes, it is generally recommended that you use

`{DynamicResource}` rather than `{StaticResource}`. This is shown in the usages. `{DynamicResource}` is recommended because the theme itself can be changed by the user. If you want the component resource that most closely matches the control author's intent for supporting a theme, you should enable your component resource reference to be dynamic also.

The `TypeInTargetAssembly` identifies a type that exists in the target assembly where the resource is actually defined. A `ComponentResourceKey` can be defined and used independently of knowing exactly where the `TypeInTargetAssembly` is defined, but eventually must resolve the type through referenced assemblies.

A common usage for `ComponentResourceKey` is to define keys that are then exposed as members of a class. For this usage, you use the `ComponentResourceKey` class constructor, not the markup extension. For more information, see `ComponentResourceKey`, or the "Defining and Referencing Keys for Theme Resources" section of the topic [Control Authoring Overview](#).

For both establishing keys and referencing keyed resources, attribute syntax is commonly used for the `ComponentResourceKey` markup extension.

The compact syntax shown relies on the `ComponentResourceKey.ComponentResourceKey` constructor signature and positional parameter usage of a markup extension. The order in which the `targetTypeName` and `targetID` are given is important. The verbose syntax relies on the `ComponentResourceKey.ComponentResourceKey` parameterless constructor, and then sets the `TypeInTargetAssembly` and `Resourceld` in a way that is analogous to a true attribute syntax on an object element. In the verbose syntax, the order in which the properties are set is not important. The relationship and mechanisms of these two alternatives (compact and verbose) is described in more detail in the topic [Markup Extensions and WPF XAML](#).

Technically, the value for `targetID` can be any object, it does not have to be a string. However, the most common usage in WPF is to align the `targetID` value with forms that are strings, and where such strings are valid in the [XamlName Grammar](#).

`ComponentResourceKey` can be used in object element syntax. In this case, specifying the value of both the `TypeInTargetAssembly` and `Resourceld` properties is required to properly initialize the extension.

In the WPF XAML reader implementation, the handling for this markup extension is defined by the `ComponentResourceKey` class.

`ComponentResourceKey` is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than just putting type converters on certain types or properties. All markup extensions in XAML use the { and } characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the attribute. For more information, see [Markup Extensions and WPF XAML](#).

## See also

- [ComponentResourceKey](#)
- [ControlTemplate](#)
- [Control Authoring Overview](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)

# DateTime XAML Syntax

11/3/2019 • 3 minutes to read • [Edit Online](#)

Some controls, such as [Calendar](#) and [DatePicker](#), have properties that use the [DateTime](#) type. Although you typically specify an initial date or time for these controls in the code-behind at run time, you can specify an initial date or time in XAML. The WPF XAML parser handles parsing of [DateTime](#) values using a built-in XAML text syntax. This topic describes the specifics of the [DateTime](#) XAML text syntax.

## When To Use DateTime XAML Syntax

Setting dates in XAML is not always necessary and may not even be desirable. For example, you could use the [DateTime.Now](#) property to initialize a date at run time, or you could do all your date adjustments for a calendar in the code-behind based on user input. However, there are scenarios where you may want to hard-code dates into a [Calendar](#) and [DatePicker](#) in a control template. The [DateTime](#) XAML syntax must be used for these scenarios.

### DateTime XAML Syntax is a Native Behavior

[DateTime](#) is a class that is defined in the base class libraries of the CLR. Because of how the base class libraries relate to the rest of the CLR, it is not possible to apply [TypeConverterAttribute](#) to the class and use a type converter to process strings from XAML and convert them to [DateTime](#) in the run time object model. There is no [DateTimeConverter](#) class that provides the conversion behavior; the conversion behavior described in this topic is native to the WPF XAML parser.

## Format Strings for DateTime XAML Syntax

You can specify the format of a [DateTime](#) with a format string. Format strings formalize the text syntax that can be used to create a value. [DateTime](#) values for the existing WPF controls generally only use the date components of [DateTime](#) and not the time components.

When specifying a [DateTime](#) in XAML, you can use any of the format strings interchangeably.

You can also use formats and format strings that are not specifically shown in this topic. Technically, the XAML for any [DateTime](#) value that is specified and then parsed by the WPF XAML parser uses an internal call to [DateTime.Parse](#), therefore you could use any string accepted by [DateTime.Parse](#) for your XAML input. For more information, see [DateTime.Parse](#).

#### IMPORTANT

The [DateTime](#) XAML syntax always uses [en-us](#) as the [CultureInfo](#) for its native conversion. This is not influenced by [Language](#) value or [xml:lang](#) value in the XAML, because XAML attribute-level type conversion acts without that context. Do not attempt to interpolate the format strings shown here due to cultural variations, such as the order in which day and month appear. The format strings shown here are the exact format strings used when parsing the XAML regardless of other culture settings.

The following sections describe some of the common [DateTime](#) format strings.

### Short Date Pattern ("d")

The following shows the short date format for a [DateTime](#) in XAML:

M/d/YYYY

This is the simplest form that specifies all necessary information for typical usages by WPF controls, and cannot be

influenced by accidental time zone offsets versus a time component, and is therefore recommended over the other formats.

For example, to specify the date of June 1, 2010, use the following string:

```
3/1/2010
```

For more information, see [DateTimeFormatInfo.ShortDatePattern](#).

### Sortable DateTime Pattern ("s")

The following shows the sortable [DateTime](#) pattern in XAML:

```
yyyy'-'MM'-'dd'T'HH':'mm':'ss
```

For example, to specify the date of June 1, 2010, use the following string (time components are all entered as 0):

```
2010-06-01T000:00:00
```

### RFC1123 Pattern ("r")

The RFC1123 pattern is useful because it could be a string input from other date generators that also use the RFC1123 pattern for culture invariant reasons. The following shows the RFC1123 [DateTime](#) pattern in XAML:

```
ddd, dd MMM yyyy HH':'mm':'ss 'UTC'
```

For example, to specify the date of June 1, 2010, use the following string (time components are all entered as 0):

```
Mon, 01 Jun 2010 00:00:00 UTC
```

### Other Formats and Patterns

As stated previously, a [DateTime](#) in XAML can be specified as any string that is acceptable as input for [DateTime.Parse](#). This includes other formalized formats (for example [UniversalSortableDateTimePattern](#)), and formats that are not formalized as a particular [DateTimeFormatInfo](#) form. For example, the form `YYYY/mm/dd` is acceptable as input for [DateTime.Parse](#). This topic does not attempt to describe all possible formats that work, and instead recommends the short date pattern as a standard practice.

## See also

- [XAML Overview \(WPF\)](#)

# DynamicResource Markup Extension

11/3/2019 • 3 minutes to read • [Edit Online](#)

Provides a value for any XAML property attribute by deferring that value to be a reference to a defined resource. Lookup behavior for that resource is analogous to run-time lookup.

## XAML Attribute Usage

```
<object property="{DynamicResource key}" .../>
```

## XAML Property Element Usage

```
<object>
  <object.property>
    <DynamicResource ResourceKey="key" .../>
  </object.property>
</object>
```

## XAML Values

key

The key for the requested resource. This key was initially assigned by the [x:Key Directive](#) if a resource was created in markup, or was provided as the `key` parameter when calling [ResourceDictionary.Add](#) if the resource was created in code.

## Remarks

A `DynamicResource` will create a temporary expression during the initial compilation and thus defer lookup for resources until the requested resource value is actually required in order to construct an object. This may potentially be after the XAML page is loaded. The resource value will be found based on key search against all active resource dictionaries starting from the current page scope, and is substituted for the placeholder expression from compilation.

### IMPORTANT

In terms of dependency property precedence, a `DynamicResource` expression is equivalent to the position where the dynamic resource reference is applied. If you set a local value for a property that previously had a `DynamicResource` expression as the local value, the `DynamicResource` is completely removed. For details, see [Dependency Property Value Precedence](#).

Certain resource access scenarios are particularly appropriate for `DynamicResource` as opposed to a [StaticResource Markup Extension](#). See [XAML Resources](#) for a discussion about the relative merits and performance implications of `DynamicResource` and `StaticResource`.

The specified `ResourceKey` should correspond to an existing resource determined by [x:Key Directive](#) at some level in your page, application, the available control themes and external resources, or system resources, and the

resource lookup will happen in that order. For more information about resource lookup for static and dynamic resources, see [XAML Resources](#).

A resource key may be any string defined in the [XamlName Grammar](#). A resource key may also be other object types, such as a [Type](#). A [Type](#) key is fundamental to how controls can be styled by themes. For more information, see [Control Authoring Overview](#).

APIs for lookup of resource values, such as [FindResource](#), follow the same resource lookup logic as used by [DynamicResource](#).

The alternative declarative means of referencing a resource is as a [StaticResource Markup Extension](#).

Attribute syntax is the most common syntax used with this markup extension. The string token provided after the [DynamicResource](#) identifier string is assigned as the [ResourceKey](#) value of the underlying [DynamicResourceExtension](#) extension class.

[DynamicResource](#) can be used in object element syntax. In this case, specifying the value of the [ResourceKey](#) property is required.

[DynamicResource](#) can also be used in a verbose attribute usage that specifies the [ResourceKey](#) property as a property=value pair:

```
<object property="{DynamicResource ResourceKey=key}" .../>
```

The verbose usage is often useful for extensions that have more than one settable property, or if some properties are optional. Because [DynamicResource](#) has only one settable property, which is required, this verbose usage is not typical.

In the WPF XAML processor implementation, the handling for this markup extension is defined by the [DynamicResourceExtension](#) class.

[DynamicResource](#) is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than just putting type converters on certain types or properties. All markup extensions in XAML use the { and } characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the attribute. For more information, see [Markup Extensions and WPF XAML](#).

## See also

- [XAML Resources](#)
- [Resources and Code](#)
- [x:Key Directive](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)
- [StaticResource Markup Extension](#)
- [Markup Extensions and WPF XAML](#)

# RelativeSource MarkupExtension

11/3/2019 • 4 minutes to read • [Edit Online](#)

Specifies properties of a [RelativeSource](#) binding source, to be used within a [Binding Markup Extension](#), or when setting the [RelativeSource](#) property of a [Binding](#) element established in XAML.

## XAML Attribute Usage

```
<Binding RelativeSource="{RelativeSource modeEnumValue}" .../>
```

## XAML Attribute Usage (nested within Binding extension)

```
<object property="{Binding RelativeSource={RelativeSource modeEnumValue} ...}" .../>
```

## XAML Object Element Usage

```
<Binding>
  <Binding.RelativeSource>
    <RelativeSource Mode="modeEnumValue"/>
  </Binding.RelativeSource>
</Binding>
```

-or-

```
<Binding>
  <Binding.RelativeSource>
    <RelativeSource
      Mode="FindAncestor"
      AncestorType="{x:Type typeName}"
      AncestorLevel="intLevel"
    />
  </Binding.RelativeSource>
</Binding>
```

## XAML Values

`modeEnumValue`

One of the following:

- The string token `Self` ; corresponds to a [RelativeSource](#) as created with its [Mode](#) property set to `Self`.
- The string token `TemplatedParent` ; corresponds to a [RelativeSource](#) as created with its [Mode](#) property set to `TemplatedParent`.
- The string token `PreviousData` ; corresponds to a [RelativeSource](#) as created with its [Mode](#) property set to `PreviousData`.
- See below for information on `FindAncestor` mode.

<code>FindAncestor</code>	The string token <code>FindAncestor</code> . Using this token enters a mode whereby a <code>RelativeSource</code> specifies an ancestor type and optionally an ancestor level. This corresponds to a <code>RelativeSource</code> as created with its <code>Mode</code> property set to <code>FindAncestor</code> .
<code>typeName</code>	Required for <code>FindAncestor</code> mode. The name of a type, which fills the <code>AncestorType</code> property.
<code>intLevel</code>	Optional for <code>FindAncestor</code> mode. An ancestor level (evaluated towards the parent direction in the logical tree).

## Remarks

`{RelativeSource TemplatedParent}` binding usages are a key technique that addresses a larger concept of the separation of a control's UI and a control's logic. This enables binding from within the template definition to the templated parent (the run time object instance where the template is applied). For this case, the [TemplateBinding Markup Extension](#) is in fact a shorthand for the following binding expression:

`{Binding RelativeSource={RelativeSource TemplatedParent}} . TemplateBinding` or

`{RelativeSource TemplatedParent}` usages are both only relevant within the XAML that defines a template. For more information, see [TemplateBinding Markup Extension](#).

`{RelativeSource FindAncestor}` is mainly used in control templates or predictable self-contained UI compositions, for cases where a control is always expected to be in a visual tree of a certain ancestor type. For example, items of an items control might use `FindAncestor` usages to bind to properties of their items control parent ancestor. Or, elements that are part of control composition in a template can use `FindAncestor` bindings to the parent elements in that same composition structure.

In the object element syntax for `FindAncestor` mode shown in the XAML Syntax sections, the second object element syntax is used specifically for `FindAncestor` mode. `FindAncestor` mode requires an `AncestorType` value. You must set `AncestorType` as an attribute using an [x:Type Markup Extension](#) reference to the type of ancestor to look for. The `AncestorType` value is used when the binding request is processed at run-time.

For `FindAncestor` mode, the optional property `AncestorLevel` can help disambiguate the ancestor lookup in cases where there is possibly more than one ancestor of that type existing in the element tree.

For more information on how to use the `FindAncestor` mode, see [RelativeSource](#).

`{RelativeSource Self}` is useful for scenarios where one property of an instance should depend on the value of another property of the same instance, and no general dependency property relationship (such as coercion) already exists between those two properties. Although it is rare that two properties exist on an object such that the values are literally identical (and are identically typed), you can also apply a `converter` parameter to a binding that has `{RelativeSource Self}`, and use the converter to convert between source and target types. Another scenario for `{RelativeSource Self}` is as part of a [MultiDataTrigger](#).

For example, the following XAML defines a `Rectangle` element such that no matter what value is entered for `Width`, the `Rectangle` is always a square:

```
<Rectangle Width="200" Height="{Binding RelativeSource={RelativeSource Self}, Path=Width}" .../>
```

`{RelativeSource PreviousData}` is useful either in data templates, or in cases where bindings are using a collection as the data source. You can use `{RelativeSource PreviousData}` to highlight relationships between adjacent data items in the collection. A related technique is to establish a [MultiBinding](#) between the current and previous items in the data source, and use a converter on that binding to determine the difference between the two items and

their properties.

In the following example, the first [TextBlock](#) in the items template displays the current number. The second [TextBlock](#) binding is a [MultiBinding](#) that nominally has two [Binding](#) constituents: the current record, and a binding that deliberately uses the previous data record by using `{RelativeSource PreviousData}`. Then, a converter on the [MultiBinding](#) calculates the difference and returns it to the binding.

```
<ListBox Name="fibolist">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding}" />
        <TextBlock>, difference = </TextBlock>
        <TextBlock>
          <TextBlock.Text>
            <MultiBinding Converter="{StaticResource DiffConverter}">
              <Binding/>
              <Binding RelativeSource="{RelativeSource PreviousData}" />
            </MultiBinding>
          </TextBlock.Text>
        </TextBlock>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
```

Describing data binding as a concept is not covered here, see [Data Binding Overview](#).

In the WPF XAML processor implementation, the handling for this markup extension is defined by the [RelativeSource](#) class.

`RelativeSource` is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than just putting type converters on certain types or properties. All markup extensions in XAML use the `{` and `}` characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the attribute. For more information, see [Markup Extensions and WPF XAML](#).

## See also

- [Binding](#)
- [Styling and Templating](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)
- [Data Binding Overview](#)
- [Binding Declarations Overview](#)
- [x>Type Markup Extension](#)

# StaticResource Markup Extension

11/3/2019 • 2 minutes to read • [Edit Online](#)

Provides a value for any XAML property attribute by looking up a reference to an already defined resource. Lookup behavior for that resource is analogous to load-time lookup, which will look for resources that were previously loaded from the markup of the current XAML page as well as other application sources, and will generate that resource value as the property value in the run-time objects.

## XAML Attribute Usage

```
<object property="{StaticResource key}" .../>
```

## XAML Object Element Usage

```
<object>
  <object.property>
    <StaticResource ResourceKey="key" .../>
  </object.property>
</object>
```

## XAML Values

key

The key for the requested resource. This key was initially assigned by the [x:Key Directive](#) if a resource was created in markup, or was provided as the `key` parameter when calling [ResourceDictionary.Add](#) if the resource was created in code.

## Remarks

### IMPORTANT

A `StaticResource` must not attempt to make a forward reference to a resource that is defined lexically further within the XAML file. Attempting to do so is not supported, and even if such a reference does not fail, attempting the forward reference will incur a load time performance penalty when the internal hash tables representing a [ResourceDictionary](#) are searched. For best results, adjust the composition of your resource dictionaries such that forward references can be avoided. If you cannot avoid a forward reference, use [DynamicResource Markup Extension](#) instead.

The specified [ResourceKey](#) should correspond to an existing resource, identified with an [x:Key Directive](#) at some level in your page, application, the available control themes and external resources, or system resources. The resource lookup occurs in that order. For more information about resource lookup behavior for static and dynamic resources, see [XAML Resources](#).

A resource key can be any string defined in the [XamlName Grammar](#). A resource key can also be other object types, such as a [Type](#). A [Type](#) key is fundamental to how controls can be styled by themes, through an implicit style key. For more information, see [Control Authoring Overview](#).

The alternative declarative means of referencing a resource is as a [DynamicResource Markup Extension](#).

Attribute syntax is the most common syntax used with this markup extension. The string token provided after the `StaticResource` identifier string is assigned as the [ResourceKey](#) value of the underlying [StaticResourceExtension](#) extension class.

`StaticResource` can be used in object element syntax. In this case, specifying the value of the [ResourceKey](#) property is required.

`StaticResource` can also be used in a verbose attribute usage that specifies the [ResourceKey](#) property as a `property=value` pair:

```
<object property="{StaticResource ResourceKey=key}" .../>
```

The verbose usage is often useful for extensions that have more than one settable property, or if some properties are optional. Because `StaticResource` has only one settable property, which is required, this verbose usage is not typical.

In the WPF XAML processor implementation, the handling for this markup extension is defined by the [StaticResourceExtension](#) class.

`StaticResource` is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than just putting type converters on certain types or properties. All markup extensions in XAML use the { and } characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the attribute. For more information, see [Markup Extensions and WPF XAML](#).

## See also

- [Styling and Templating](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)
- [XAML Resources](#)
- [Resources and Code](#)

# TemplateBinding Markup Extension

11/12/2019 • 2 minutes to read • [Edit Online](#)

Links the value of a property in a control template to be the value of another property on the templated control.

## XAML Attribute Usage

```
<object property="{TemplateBinding sourceProperty}" .../>
```

## XAML Attribute Usage (for Setter property in template or style)

```
<Setter Property="propertyName" Value="{TemplateBinding sourceProperty}" .../>
```

## XAML Values

<code>propertyName</code>	<code>DependencyProperty.Name</code> of the property being set in the setter syntax.
<code>sourceProperty</code>	<p>Another dependency property that exists on the type being templated, specified by its <code>DependencyProperty.Name</code>.</p> <p>- or -</p> <p>A "dotted-down" property name that is defined by a different type than the target type being templated. This is actually a <code>PropertyPath</code>. See <a href="#">PropertyPath XAML Syntax</a>.</p>

## Remarks

A `TemplateBinding` is an optimized form of a `Binding` for template scenarios, analogous to a `Binding` constructed with `{Binding RelativeSource={RelativeSource TemplatedParent}, Mode=OneWay}`. A `TemplateBinding` is always a one-way binding, even if properties involved default to two-way binding. Both properties involved must be dependency properties. In order to achieve two-way binding to a templated parent use the following binding statement instead

```
{Binding RelativeSource={RelativeSource TemplatedParent}, Mode=TwoWay, Path=MyDependencyProperty}.
```

`RelativeSource` is another markup extension that is sometimes used in conjunction with or instead of `TemplateBinding` in order to perform relative property binding within a template.

Describing control templates as a concept is not covered here; for more information, see [Control Styles and Templates](#).

Attribute syntax is the most common syntax used with this markup extension. The string token provided after the `TemplateBinding` identifier string is assigned as the `Property` value of the underlying `TemplateBindingExtension` extension class.

Object element syntax is possible, but it is not shown because it has no realistic application. `TemplateBinding` is used to fill values within setters, using evaluated expressions, and using object element syntax for

`TemplateBinding` to fill `<Setter.Property>` property element syntax is unnecessarily verbose.

`TemplateBinding` can also be used in a verbose attribute usage that specifies the [Property](#) property as a property=value pair:

```
<object property="{TemplateBinding Property=sourceProperty}" .../>
```

The verbose usage is often useful for extensions that have more than one settable property, or if some properties are optional. Because `TemplateBinding` has only one settable property, which is required, this verbose usage is not typical.

In the WPF XAML processor implementation, the handling for this markup extension is defined by the [TemplateBindingExtension](#) class.

`TemplateBinding` is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than just putting type converters on certain types or properties. All markup extensions in XAML use the `{` and `}` characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the attribute. For more information, see [Markup Extensions and WPF XAML](#).

## See also

- [Style](#)
- [ControlTemplate](#)
- [Styling and Templating](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)
- [RelativeSource MarkupExtension](#)
- [Binding Markup Extension](#)

# ThemeDictionary Markup Extension

11/3/2019 • 2 minutes to read • [Edit Online](#)

Provides a way for custom control authors or applications that integrate third-party controls to load theme-specific resource dictionaries to use in styling the control.

## XAML Attribute Usage

```
<object property="{ThemeDictionary assemblyUri}" .../>
```

## XAML Object Element Usage

```
<object>
  <object.property>
    <ThemeDictionary AssemblyName="assemblyUri"/>
  <object.property>
<object>
```

## XAML Values

assemblyUri

The uniform resource identifier (URI) of the assembly that contains theme information. Typically, this is a pack URI that references an assembly in the larger package. Assembly resources and pack URIs simplify deployment issues. For more information see [Pack URLs in WPF](#).

## Remarks

This extension is intended to fill only one specific property value: a value for [ResourceDictionary.Source](#).

By using this extension, you can specify a single resources-only assembly that contains some styles to use only when the Windows Aero theme is applied to the user's system, other styles only when the Luna theme is active, and so on. By using this extension, the contents of a control-specific resource dictionary can be automatically invalidated and reloaded to be specific for another theme when required.

The `assemblyUri` string ([AssemblyName](#) property value) forms the basis of a naming convention that identifies which dictionary applies for a particular theme. The [ProvideValue](#) logic for `ThemeDictionary` completes the convention by generating a uniform resource identifier (URI) that points to a particular theme dictionary variant, as contained within a precompiled resource assembly. Describing this convention, or theme interactions with general control styling and page/application level styling as a concept, is not covered fully here. The basic scenario for using `ThemeDictionary` is to specify the [Source](#) property of a `ResourceDictionary` declared at the application level. When you provide a URI for the assembly through a `ThemeDictionary` extension rather than as a direct URI, the extension logic will provide invalidation logic that applies whenever the system theme changes.

Attribute syntax is the most common syntax used with this markup extension. The string token provided after the `ThemeDictionary` identifier string is assigned as the [AssemblyName](#) value of the underlying `ThemeDictionaryExtension` extension class.

`ThemeDictionary` may also be used in object element syntax. In this case, specifying the value of the `AssemblyName` property is required.

`ThemeDictionary` can also be used in a verbose attribute usage that specifies the `Member` property as a `property=value` pair:

```
<object property="{ThemeDictionary AssemblyName=assemblyUri}" .../>
```

The verbose usage is often useful for extensions that have more than one settable property, or if some properties are optional. Because `ThemeDictionary` has only one settable property, which is required, this verbose usage is not typical.

In the WPF XAML processor implementation, the handling for this markup extension is defined by the [ThemeDictionaryExtension](#) class.

`ThemeDictionary` is a markup extension. Markup extensions are typically implemented when there is a requirement to escape attribute values to be other than literal values or handler names, and the requirement is more global than just putting type converters on certain types or properties. All markup extensions in XAML use the { and } characters in their attribute syntax, which is the convention by which a XAML processor recognizes that a markup extension must process the attribute. For more information, see [Markup Extensions and WPF XAML](#).

## See also

- [Styling and Templating](#)
- [XAML Overview \(WPF\)](#)
- [Markup Extensions and WPF XAML](#)
- [WPF Application Resource, Content, and Data Files](#)

# PropertyPath XAML Syntax

11/7/2019 • 10 minutes to read • [Edit Online](#)

The [PropertyPath](#) object supports a complex inline XAML syntax for setting various properties that take the [PropertyPath](#) type as their value. This topic documents the [PropertyPath](#) syntax as applied to binding and animation syntaxes.

## Where PropertyPath Is Used

[PropertyPath](#) is a common object that is used in several Windows Presentation Foundation (WPF) features. Despite using the common [PropertyPath](#) to convey property path information, the usages for each feature area where [PropertyPath](#) is used as a type vary. Therefore, it is more practical to document the syntaxes on a per-feature basis.

Primarily, WPF uses [PropertyPath](#) to describe object-model paths for traversing the properties of an object data source, and to describe the target path for targeted animations.

Some style and template properties such as [Setter.Property](#) take a qualified property name that superficially resembles a [PropertyPath](#). But this is not a true [PropertyPath](#); instead it is a qualified *owner:property* string format usage that is enabled by the WPF XAML processor in combination with the type converter for [DependencyProperty](#).

## PropertyPath for Objects in Data Binding

Data binding is a WPF feature whereby you can bind to the target value of any dependency property. However, the source of such a data binding need not be a dependency property; it can be any property type that is recognized by the applicable data provider. Property paths are particularly used for the [ObjectDataProvider](#), which is used for obtaining binding sources from common language runtime (CLR) objects and their properties.

Note that data binding to XML does not use [PropertyPath](#), because it does not use [Path](#) in the [Binding](#). Instead, you use [XPath](#) and specify valid XPath syntax into the XML Document Object Model (DOM) of the data. [XPath](#) is also specified as a string, but is not documented here; see [Bind to XML Data Using an XMLDataProvider and XPath Queries](#).

A key to understanding property paths in data binding is that you can target the binding to an individual property value, or you can instead bind to target properties that take lists or collections. If you are binding collections, for instance binding a [ListBox](#) that will expand depending on how many data items are in the collection, then your property path should reference the collection object, not individual collection items. The data binding engine will match the collection used as the data source to the type of the binding target automatically, resulting in behavior such as populating a [ListBox](#) with an items array.

### Single Property on the Immediate Object as Data Context

```
<Binding Path="propertyName" .../>
```

*propertyName* must resolve to be the name of a property that is in the current [DataContext](#) for a [Path](#) usage. If your binding updates the source, that property must be read/write and the source object must be mutable.

### Single Indexer on the Immediate Object as Data Context

```
<Binding Path="[key]" .../>
```

`key` must be either the typed index to a dictionary or hash table, or the integer index of an array. Also, the value of the key must be a type that is directly bindable to the property where it is applied. For instance, a hash table that contains string keys and string values can be used this way to bind to Text for a [TextBox](#). Or, if the key points to a collection or subindex, you could use this syntax to bind to a target collection property. Otherwise, you need to reference a specific property, through a syntax such as `<Binding Path="[key].propertyName" .../>`.

You can specify the type of the index if necessary. For details on this aspect of an indexed property path, see [Binding.Path](#).

## Multiple Property (Indirect Property Targeting)

```
<Binding Path="propertyName.propertyName2" .../>
```

`propertyName` must resolve to be the name of a property that is the current [DataContext](#). The path properties `propertyName` and `propertyName2` can be any properties that exist in a relationship, where `propertyName2` is a property that exists on the type that is the value of `propertyName`.

## Single Property, Attached or Otherwise Type-Qualified

```
<object property="(ownerType.propertyName)" .../>
```

The parentheses indicate that this property in a [PropertyPath](#) should be constructed using a partial qualification. It can use an XML namespace to find the type with an appropriate mapping. The `ownerType` searches types that a XAML processor has access to, through the [XmlnsDefinitionAttribute](#) declarations in each assembly. Most applications have the default XML namespace mapped to the

<http://schemas.microsoft.com/winfx/2006/xaml/presentation> namespace, so a prefix is usually only necessary for custom types or types otherwise outside that namespace. `propertyName` must resolve to be the name of a property existing on the `ownerType`. This syntax is generally used for one of the following cases:

- The path is specified in XAML that is in a style or template that does not have a specified Target Type. A qualified usage is generally not valid for cases other than this, because in non-style, non-template cases, the property exists on an instance, not a type.
- The property is an attached property.
- You are binding to a static property.

For use as storyboard target, the property specified as `propertyName` must be a [DependencyProperty](#).

## Source Traversal (Binding to Hierarchies of Collections)

```
<object Path="propertyName/propertyNameX" .../>
```

The / in this syntax is used to navigate within a hierarchical data source object, and multiple steps into the hierarchy with successive / characters are supported. The source traversal accounts for the current record pointer position, which is determined by synchronizing the data with the UI of its view. For details on binding with hierarchical data source objects, and the concept of current record pointer in data binding, see [Use the Master-Detail Pattern with Hierarchical Data](#) or [Data Binding Overview](#).

## NOTE

Superficially, this syntax resembles XPath. A true XPath expression for binding to an XML data source is not used as a [Path](#) value and should instead be used for the mutually exclusive [XPath](#) property.

## Collection Views

To reference a named collection view, prefix the collection view name with the hash character (`#`).

## Current Record Pointer

To reference the current record pointer for a collection view or master detail data binding scenario, start the path string with a forward slash (`/`). Any path past the forward slash is traversed starting from the current record pointer.

## Multiple Indexers

```
<object Path="[index1,index2...]" .../>
```

or

```
<object Path="propertyName[index,index2...]" .../>
```

If a given object supports multiple indexers, those indexers can be specified in order, similar to an array referencing syntax. The object in question can be either the current context or the value of a property that contains a multiple index object.

By default, the indexer values are typed by using the characteristics of the underlying object. You can specify the type of the index if necessary. For details on typing the indexers, see [Binding.Path](#).

## Mixing Syntaxes

Each of the syntaxes shown above can be interspersed. For instance, the following is an example that creates a property path to the color at a particular x,y of a `ColorGrid` property that contains a pixel grid array of `SolidColorBrush` objects:

```
<Rectangle Fill="{Binding ColorGrid[20,30].SolidColorBrushResult}" .../>
```

## Escapes for Property Path Strings

For certain business objects, you might encounter a case where the property path string requires an escape sequence in order to parse correctly. The need to escape should be rare, because many of these characters have similar naming-interaction issues in languages that would typically be used to define the business object.

- Inside indexers ([ ]), the caret character (^) escapes the next character.
- You must 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 ">".
- You must escape (using backslash \) characters that are special to the WPF XAML parser behavior for processing a markup extension.
  - 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.

#### NOTE

Technically, these escapes work for a storyboard property path also, but you are usually traversing object models for existing WPF objects, and escaping should be unnecessary.

## PropertyPath for Animation Targets

The target property of an animation must be a dependency property that takes either a [Freezable](#) or a primitive type. However, the targeted property on a type and the eventual animated property can exist on different objects. For animations, a property path is used to define the connection between the named animation target object's property and the intended target animation property, by traversing object-property relationships in the property values.

### General Object-Property Considerations for Animations

For more information on animation concepts in general, see [Storyboards Overview](#) and [Animation Overview](#).

The value type or the property being animated must be either a [Freezable](#) type or a primitive. The property that starts the path must resolve to be the name of a dependency property that exists on the specified [TargetName](#) type.

In order to support cloning for animating a [Freezable](#) that is already frozen, the object specified by [TargetName](#) must be a [FrameworkElement](#) or [FrameworkContentElement](#) derived class.

### Single Property on the Target Object

```
<animation Storyboard.TargetProperty="propertyName" .../>
```

`propertyName` must resolve to be the name of a dependency property that exists on the specified [TargetName](#) type.

### Indirect Property Targeting

```
<animation Storyboard.TargetProperty="propertyName.propertyName2" .../>
```

`propertyName` must be a property that is either a [Freezable](#) value type or a primitive, which exists on the specified [TargetName](#) type.

`propertyName2` must be the name of a dependency property that exists on the object that is the value of `propertyName`. In other words, `propertyName2` must exist as a dependency property on the type that is the `propertyName` [.PropertyType](#).

Indirect targeting of animations is necessary because of applied styles and templates. In order to target an animation, you need a [TargetName](#) on a target object, and that name is established by [x:Name](#) or [Name](#). Although template and style elements also can have names, those names are only valid within the namescope of the style and template. (If templates and styles did share namespaces with application markup, names couldn't be unique. The styles and templates are literally shared between instances and would perpetuate duplicate names.) Thus, if the individual properties of an element that you might wish to animate came from a style or template, you need to start with a named element instance that is not from a style template, and then target into the style or template visual tree to arrive at the property you wish to animate.

For instance, the [Background](#) property of a [Panel](#) is a complete [Brush](#) (actually a [SolidColorBrush](#)) that came from a theme template. To animate a [Brush](#) completely, there would need to be a [BrushAnimation](#) (probably one for

every [Brush](#) type) and there is no such type. To animate a Brush, you instead animate properties of a particular Brush type. You need to get from [SolidColorBrush](#) to its [Color](#) to apply a [ColorAnimation](#) there. The property path for this example would be `Background.Color`.

## Attached Properties

```
<animation Storyboard.TargetProperty="(ownerType.propertyName)" .../>
```

The parentheses indicate that this property in a [PropertyPath](#) should be constructed using a partial qualification. It can use an XML namespace to find the type. The `ownerType` searches types that a XAML processor has access to, through the [XmlAttributeDefinition](#) declarations in each assembly. Most applications have the default XML namespace mapped to the `http://schemas.microsoft.com/winfx/2006/xaml/presentation` namespace, so a prefix is usually only necessary for custom types or types otherwise outside that namespace. `propertyName` must resolve to be the name of a property existing on the `ownerType`. The property specified as `propertyName` must be a [DependencyProperty](#). (All WPF attached properties are implemented as dependency properties, so this issue is only of concern for custom attached properties.)

## Indexers

```
<animation Storyboard.TargetProperty="propertyName.propertyName2[index].propertyName3" .../>
```

Most dependency properties or [Freezable](#) types do not support an indexer. Therefore, the only usage for an indexer in an animation path is at an intermediate position between the property that starts the chain on the named target and the eventual animated property. In the provided syntax, that is `propertyName2`. For instance, an indexer usage might be necessary if the intermediate property is a collection such as [TransformGroup](#), in a property path such as `RenderTransform.Children[1].Angle`.

## PropertyPath in Code

Code usage for [PropertyPath](#), including how to construct a [PropertyPath](#), is documented in the reference topic for [PropertyPath](#).

In general, [PropertyPath](#) is designed to use two different constructors, one for the binding usages and simplest animation usages, and one for the complex animation usages. Use the [PropertyPath\(Object\)](#) signature for binding usages, where the object is a string. Use the [PropertyPath\(Object\)](#) signature for one-step animation paths, where the object is a [DependencyProperty](#). Use the [PropertyPath\(String, Object\[\]\)](#) signature for complex animations. This latter constructor uses a token string for the first parameter and an array of objects that fill positions in the token string to define a property path relationship.

## See also

- [PropertyPath](#)
- [Data Binding Overview](#)
- [Storyboards Overview](#)

# PresentationOptions:Freeze Attribute

9/14/2019 • 2 minutes to read • [Edit Online](#)

Sets the `IsFrozen` state to `true` on the containing `Freezable` element. Default behavior for a `Freezable` without the `PresentationOptions:Freeze` attribute specified is that `IsFrozen` is `false` at load time, and dependent on general `Freezable` behavior at runtime.

## XAML Attribute Usage

```
<object
    xmlns:PresentationOptions="http://schemas.microsoft.com/winfx/2006/xaml/presentation/options"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="PresentationOptions">
    <freezableElement PresentationOptions:Freeze="true"/>
</object>
```

## XAML Values

<code>PresentationOptions</code>	An XML namespace prefix, which can be any valid prefix string, per the XML 1.0 specification. The prefix <code>PresentationOptions</code> is used for identification purposes in this documentation.
<code>freezableElement</code>	An element that instantiates any derived class of <code>Freezable</code> .

## Remarks

The `Freeze` attribute is the only attribute or other programming element defined in the `http://schemas.microsoft.com/winfx/2006/xaml/presentation/options` XML namespace. The `Freeze` attribute exists in this special namespace specifically so that it can be designated as ignorable, using `mc:Ignorable Attribute` as part of the root element declarations. The reason that `Freeze` must be able to be ignorable is because not all XAML processor implementations are able to freeze a `Freezable` at load time; this capability is not part of the XAML specification.

The ability to process the `Freeze` attribute is specifically built in to the XAML processor that processes XAML for compiled applications. The attribute is not supported by any class, and the attribute syntax is not extensible or modifiable. If you are implementing your own XAML processor you can choose to parallel the freezing behavior of the WPF XAML processor when processing the `Freeze` attribute on `Freezable` elements at load time.

Any value for the `Freeze` attribute other than `true` (not case sensitive) generates a load time error. (Specifying the `Freeze` attribute as `false` is not an error, but that is already the default, so setting to `false` does nothing).

## See also

- [Freezable](#)
- [Freezable Objects Overview](#)
- [mc:Ignorable Attribute](#)

# Markup Compatibility (mc:) Language Features

3/5/2019 • 2 minutes to read • [Edit Online](#)

## In This Section

[mc:Ignorable Attribute](#)

[mc:ProcessContent Attribute](#)

# mc:Ignorable Attribute

11/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies which XML namespace prefixes encountered in a markup file may be ignored by a XAML processor. The `mc:Ignorable` attribute supports markup compatibility both for custom namespace mapping and for XAML versioning.

## XAML Attribute Usage (Single Prefix)

```
<object
    xmlns:ignorablePrefix="ignorableUri"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="ignorablePrefix"...
        <ignorablePrefix1:ThisElementCanBeIgnored/>
</object>
```

## XAML Attribute Usage (Two Prefixes)

```
<object
    xmlns:ignorablePrefix1="ignorableUri"
    xmlns:ignorablePrefix2="ignorableUri2"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="ignorablePrefix1 ignorablePrefix2"...
        <ignorablePrefix1:ThisElementCanBeIgnored/>
</object>
```

## XAML Values

<i>ignorablePrefix, ignorablePrefix1, etc.</i>	Any valid prefix string, per the XML 1.0 specification.
<i>ignorableUri</i>	Any valid URI for designating a namespace, per the XML 1.0 specification.
<i>ThisElementCanBeIgnored</i>	An element that can be ignored by Extensible Application Markup Language (XAML) processor implementations, if the underlying type cannot be resolved.

## Remarks

The `mc` XML namespace prefix is the recommended prefix convention to use when mapping the XAML compatibility namespace <http://schemas.openxmlformats.org/markup-compatibility/2006>.

Elements or attributes where the prefix portion of the element name are identified as `mc:Ignorable` will not raise errors when processed by a XAML processor. If that attribute could not be resolved to an underlying type or programming construct, then that element is ignored. Note however that ignored elements might still generate additional parsing errors for additional element requirements that are side effects of that element not being processed. For instance, a particular element content model might require exactly one child element, but if the specified child element was in an `mc:Ignorable` prefix, and the specified child element could not be resolved to a

type, then the XAML processor might raise an error.

`mc:Ignorable` only applies to namespace mappings to identifier strings. `mc:Ignorable` does not apply to namespace mappings into assemblies, which specify a CLR namespace and an assembly (or default to the current executable as the assembly).

If you are implementing a XAML processor, your processor implementation must not raise parsing or processing errors on type resolution for any element or attribute that is qualified by a prefix that is identified as `mc:Ignorable`. But your processor implementation can still raise exceptions that are a secondary result of an element failing to load or be processed, such as the one-child element example given earlier.

By default, a XAML processor will ignore content within an ignored element. However, you can specify an additional attribute, [mc:ProcessContent Attribute](#), to require continued processing of content within an ignored element by the next available parent element.

Multiple prefixes can be specified in the attribute, using one or more white-space characters as the separator, for example: `mc:Ignorable="ignore1 ignore2"`.

The `http://schemas.openxmlformats.org/markup-compatibility/2006` namespace defines other elements and attributes that are not documented within this area of the SDK. For more information, see [XML Markup Compatibility Specification](#).

## See also

- [XamlReader](#)
- [PresentationOptions:Freeze Attribute](#)
- [XAML Overview \(WPF\)](#)
- [Documents in WPF](#)

# mc:ProcessContent Attribute

11/3/2019 • 2 minutes to read • [Edit Online](#)

Specifies which XAML elements should still have content processed by relevant parent elements, even if the immediate parent element may be ignored by a XAML processor due to specifying [mc:Ignorable Attribute](#). The `mc:ProcessContent` attribute supports markup compatibility both for custom namespace mapping and for XAML versioning.

## XAML Attribute Usage

```
<object
  xmlns:ignorablePrefix="ignorableUri"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="ignorablePrefix"...
  mc:ProcessContent="ignorablePrefix:ThisElementCanBeIgnored"
>
  <ignorablePrefix:ThisElementCanBeIgnored>
    [content]
  </ignorablePrefix:ThisElementCanBeIgnored>
</object>
```

## XAML Values

<i>ignorablePrefix</i>	Any valid prefix string, per the XML 1.0 specification.
<i>ignorableUri</i>	Any valid URI for designating a namespace, per the XML 1.0 specification.
<i>ThisElementCanBeIgnored</i>	An element that can be ignored by Extensible Application Markup Language (XAML) processor implementations, if the underlying type cannot be resolved.
<i>[content]</i>	<i>ThisElementCanBeIgnored</i> is marked ignorable. If the processor ignores that element, <i>[content]</i> is processed by <i>object</i> .

## Remarks

By default, a XAML processor will ignore content within an ignored element. You can specify a specific element by `mc:ProcessContent`, and a XAML processor will continue to process the content within the ignored element. This would typically be used if the content is nested within several tags, at least one of which is ignorable and at least one of which is not ignorable.

Multiple prefixes may be specified in the attribute, using a space separator, for example:

```
mc:ProcessContent="ignore:Element1 ignore:Element2" .
```

The `http://schemas.openxmlformats.org/markup-compatibility/2006` namespace defines other elements and attributes that are not documented within this area of the SDK. For more information, see [XML Markup Compatibility Specification](#).

## See also

- [mc:Ignorable Attribute](#)
- [XAML Overview \(WPF\)](#)

# Base Elements

11/3/2019 • 2 minutes to read • [Edit Online](#)

Four key classes--[UIElement](#), [ContentElement](#), [FrameworkElement](#), and [FrameworkContentElement](#)--implement a substantial percentage of the common element functionality available in WPF programming. These four classes are referred to in this SDK as the base element classes.

## In This Section

[Base Elements Overview](#)

[Freezable Objects Overview](#)

[Alignment, Margins, and Padding Overview](#)

[How-to Topics](#)

## Reference

[UIElement](#)

[ContentElement](#)

[FrameworkElement](#)

[FrameworkContentElement](#)

## Related Sections

[WPF Architecture](#)

[XAML in WPF](#)

[Element Tree and Serialization](#)

[Properties](#)

[Events](#)

[Input](#)

[Resources](#)

[Styling and Templating](#)

[Threading Model](#)

# Base Elements Overview

11/7/2019 • 6 minutes to read • [Edit Online](#)

A high percentage of classes in Windows Presentation Foundation (WPF) are derived from four classes which are commonly referred to in the SDK documentation as the base element classes. These classes are [UIElement](#), [FrameworkElement](#), [ContentElement](#), and [FrameworkContentElement](#). The [DependencyObject](#) class is also related, because it is a common base class of both [UIElement](#) and [ContentElement](#).

## Base Element APIs in WPF Classes

Both [UIElement](#) and [ContentElement](#) are derived from [DependencyObject](#), through somewhat different pathways. The split at this level deals with how a [UIElement](#) or [ContentElement](#) are used in a user interface and what purpose they serve in an application. [UIElement](#) also has [Visual](#) in its class hierarchy, which is a class that exposes the lower-level graphics support underlying the Windows Presentation Foundation (WPF). [Visual](#) provides a rendering framework by defining independent rectangular screen regions. In practice, [UIElement](#) is for elements that will support a larger object model, are intended to render and layout into regions that can be described as rectangular screen regions, and where the content model is deliberately more open, to allow different combinations of elements. [ContentElement](#) does not derive from [Visual](#); its model is that a [ContentElement](#) would be consumed by something else, such as a reader or viewer that would then interpret the elements and produce the complete [Visual](#) for Windows Presentation Foundation (WPF) to consume. Certain [UIElement](#) classes are intended to be content hosts: they provide the hosting and rendering for one or more [ContentElement](#) classes ([DocumentViewer](#) is an example of such a class). [ContentElement](#) is used as base class for elements with somewhat smaller object models and that more address the text, information, or document content that might be hosted within a [UIElement](#).

### Framework-Level and Core-Level

[UIElement](#) serves as the base class for [FrameworkElement](#), and [ContentElement](#) serves as the base class for [FrameworkContentElement](#). The reason for this next level of classes is to support a WPF core level that is separate from a WPF framework level, with this division also existing in how the APIs are divided between the [PresentationCore](#) and [PresentationFramework](#) assemblies. The WPF framework level presents a more complete solution for basic application needs, including the implementation of the layout manager for presentation. The WPF core level provides a way to use much of WPF without taking the overhead of the additional assembly. The distinction between these levels very rarely matters for most typical application development scenarios, and in general you should think of the WPF APIs as a whole and not concern yourself with the difference between WPF framework level and WPF core level. You might need to know about the level distinctions if your application design chooses to replace substantial quantities of WPF framework level functionality, for instance if your overall solution already has its own implementations of user interface (UI) composition and layout.

## Choosing Which Element to Derive From

The most practical way to create a custom class that extends WPF is by deriving from one of the WPF classes where you get as much as possible of your desired functionality through the existing class hierarchy. This section lists the functionality that comes with three of the most important element classes to help you decide which class to inherit from.

If you are implementing a control, which is really one of the more common reasons for deriving from a WPF class, you probably want to derive from a class that is a practical control, a control family base class, or at least from the [Control](#) base class. For some guidance and practical examples, see [Control Authoring Overview](#).

If you are not creating a control and need to derive from a class that is higher in the hierarchy, the following

sections are intended as a guide for what characteristics are defined in each base element class.

If you create a class that derives from [DependencyObject](#), you inherit the following functionality:

- [GetValue](#) and [SetValue](#) support, and general property system support.
- Ability to use dependency properties and attached properties that are implemented as dependency properties.

If you create a class that derives from [UIElement](#), you inherit the following functionality in addition to that provided by [DependencyObject](#):

- Basic support for animated property values. For more information, see [Animation Overview](#).
- Basic input event support, and commanding support. For more information, see [Input Overview](#) and [Commanding Overview](#).
- Virtual methods that can be overridden to provide information to a layout system.

If you create a class that derives from [FrameworkElement](#), you inherit the following functionality in addition to that provided by [UIElement](#):

- Support for styling and storyboards. For more information, see [Style](#) and [Storyboards Overview](#).
- Support for data binding. For more information, see [Data Binding Overview](#).
- Support for dynamic resource references. For more information, see [XAML Resources](#).
- Property value inheritance support, and other flags in the metadata that help report conditions about properties to framework services such as data binding, styles, or the framework implementation of layout. For more information, see [Framework Property Metadata](#).
- The concept of the logical tree. For more information, see [Trees in WPF](#).
- Support for the practical WPF framework-level implementation of the layout system, including an [OnPropertyChanged](#) override that can detect changes to properties that influence layout.

If you create a class that derives from [ContentElement](#), you inherit the following functionality in addition to that provided by [DependencyObject](#):

- Support for animations. For more information, see [Animation Overview](#).
- Basic input event support, and commanding support. For more information, see [Input Overview](#) and [Commanding Overview](#).

If you create a class that derives from [FrameworkContentElement](#), you get the following functionality in addition to that provided by [ContentElement](#):

- Support for styling and storyboards. For more information, see [Style](#) and [Animation Overview](#).
- Support for data binding. For more information, see [Data Binding Overview](#).
- Support for dynamic resource references. For more information, see [XAML Resources](#).
- Property value inheritance support, and other flags in the metadata that help report conditions about properties to framework services like data binding, styles, or the framework implementation of layout. For more information, see [Framework Property Metadata](#).
- You do not inherit access to layout system modifications (such as [ArrangeOverride](#)). Layout system implementations are only available on [FrameworkElement](#). However, you inherit an [OnPropertyChanged](#) override that can detect changes to properties that influence layout and report these to any content hosts.

Content models are documented for a variety of classes. The content model for a class is one possible factor you should consider if you want to find an appropriate class to derive from. For more information, see [WPF Content Model](#).

## Other Base Classes

### **DispatcherObject**

[DispatcherObject](#) provides support for the WPF threading model and enables all objects created for WPF applications to be associated with a [Dispatcher](#). Even if you do not derive from [UIElement](#), [DependencyObject](#), or [Visual](#), you should consider deriving from [DispatcherObject](#) in order to get this threading model support. For more information, see [Threading Model](#).

### **Visual**

[Visual](#) implements the concept of a 2D object that generally requires visual presentation in a roughly rectangular region. The actual rendering of a [Visual](#) happens in other classes (it is not self-contained), but the [Visual](#) class provides a known type that is used by rendering processes at various levels. [Visual](#) implements hit testing, but it does not expose events that report hit-testing positives (these are in [UIElement](#)). For more information, see [Visual Layer Programming](#).

### **Freezable**

[Freezable](#) simulates immutability in a mutable object by providing the means to generate copies of the object when an immutable object is required or desired for performance reasons. The [Freezable](#) type provides a common basis for certain graphics elements such as geometries and brushes, as well as animations. Notably, a [Freezable](#) is not a [Visual](#); it can hold properties that become subproperties when the [Freezable](#) is applied to fill a property value of another object, and those subproperties might affect rendering. For more information, see [Freezable Objects Overview](#).

### **Animatable**

[Animatable](#) is a [Freezable](#) derived class that specifically adds the animation control layer and some utility members so that currently animated properties can be distinguished from nonanimated properties.

### **Control**

[Control](#) is the intended base class for the type of object that is variously termed a control or component, depending on the technology. In general, WPF control classes are classes that either directly represent a UI control or participate closely in control composition. The primary functionality that [Control](#) enables is control templating.

## See also

- [Control](#)
- [Dependency Properties Overview](#)
- [Control Authoring Overview](#)
- [WPF Architecture](#)

# Freezable Objects Overview

11/3/2019 • 9 minutes to read • [Edit Online](#)

This topic describes how to effectively use and create [Freezable](#) objects, which provide special features that can help improve application performance. Examples of freezable objects include brushes, pens, transformations, geometries, and animations.

## What Is a Freezable?

A [Freezable](#) is a special type of object that has two states: unfrozen and frozen. When unfrozen, a [Freezable](#) appears to behave like any other object. When frozen, a [Freezable](#) can no longer be modified.

A [Freezable](#) provides a [Changed](#) event to notify observers of any modifications to the object. Freezing a [Freezable](#) can improve its performance, because it no longer needs to spend resources on change notifications. A frozen [Freezable](#) can also be shared across threads, while an unfrozen [Freezable](#) cannot.

Although the [Freezable](#) class has many applications, most [Freezable](#) objects in Windows Presentation Foundation (WPF) are related to the graphics sub-system.

The [Freezable](#) class makes it easier to use certain graphics system objects and can help improve application performance. Examples of types that inherit from [Freezable](#) include the [Brush](#), [Transform](#), and [Geometry](#) classes. Because they contain unmanaged resources, the system must monitor these objects for modifications, and then update their corresponding unmanaged resources when there is a change to the original object. Even if you don't actually modify a graphics system object, the system must still spend some of its resources monitoring the object, in case you do change it.

For example, suppose you create a [SolidColorBrush](#) brush and use it to paint the background of a button.

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);
myButton.Background = myBrush;
```

```
Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)
myButton.Background = myBrush
```

When the button is rendered, the WPF graphics sub-system uses the information you provided to paint a group of pixels to create the appearance of a button. Although you used a solid color brush to describe how the button should be painted, your solid color brush doesn't actually do the painting. The graphics system generates fast, low-level objects for the button and the brush, and it is those objects that actually appear on the screen.

If you were to modify the brush, those low-level objects would have to be regenerated. The freezable class is what gives a brush the ability to find its corresponding generated, low-level objects and to update them when it changes. When this ability is enabled, the brush is said to be "unfrozen."

A freezable's [Freeze](#) method enables you to disable this self-updating ability. You can use this method to make the brush become "frozen," or unmodifiable.

#### NOTE

Not every [Freezable](#) object can be frozen. To avoid throwing an [InvalidOperationException](#), check the value of the [Freezable](#) object's [CanFreeze](#) property to determine whether it can be frozen before attempting to freeze it.

```
if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}
```

```
If myBrush.CanFreeze Then
    ' Makes the brush unmodifiable.
    myBrush.Freeze()
End If
```

When you no longer need to modify a freezable, freezing it provides performance benefits. If you were to freeze the brush in this example, the graphics system would no longer need to monitor it for changes. The graphics system can also make other optimizations, because it knows the brush won't change.

#### NOTE

For convenience, freezable objects remain unfrozen unless you explicitly freeze them.

## Using Freezables

Using an unfrozen freezable is like using any other type of object. In the following example, the color of a [SolidColorBrush](#) is changed from yellow to red after it's used to paint the background of a button. The graphics system works behind the scenes to automatically change the button from yellow to red the next time the screen is refreshed.

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);
myButton.Background = myBrush;

// Changes the button's background to red.
myBrush.Color = Colors.Red;
```

```
Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)
myButton.Background = myBrush

' Changes the button's background to red.
myBrush.Color = Colors.Red
```

### Freezing a Freezable

To make a [Freezable](#) unmodifiable, you call its [Freeze](#) method. When you freeze an object that contains freezable objects, those objects are frozen as well. For example, if you freeze a [PathGeometry](#), the figures and segments it contains would be frozen too.

A Freezable **can't** be frozen if any of the following are true:

- It has animated or data bound properties.
- It has properties set by a dynamic resource. (See the [XAML Resources](#) for more information about dynamic resources.)
- It contains [Freezable](#) sub-objects that can't be frozen.

If these conditions are false, and you don't intend to modify the [Freezable](#), then you should freeze it to gain the performance benefits described earlier.

Once you call a freezable's [Freeze](#) method, it can no longer be modified. Attempting to modify a frozen object causes an [InvalidOperationException](#) to be thrown. The following code throws an exception, because we attempt to modify the brush after it's been frozen.

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

try {

    // Throws an InvalidOperationException, because the brush is frozen.
    myBrush.Color = Colors.Red;
} catch(InvalidOperationException ex)
{
    MessageBox.Show("Invalid operation: " + ex.ToString());
}
```

```
Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)

If myBrush.CanFreeze Then
    ' Makes the brush unmodifiable.
    myBrush.Freeze()
End If

myButton.Background = myBrush

Try

    ' Throws an InvalidOperationException, because the brush is frozen.
    myBrush.Color = Colors.Red
Catch ex As InvalidOperationException
    MessageBox.Show("Invalid operation: " & ex.ToString())
End Try
```

To avoid throwing this exception, you can use the [IsFrozen](#) method to determine whether a [Freezable](#) is frozen.

```

Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

if (myBrush.IsFrozen) // Evaluates to true.
{
    // If the brush is frozen, create a clone and
    // modify the clone.
    SolidColorBrush myBrushClone = myBrush.Clone();
    myBrushClone.Color = Colors.Red;
    myButton.Background = myBrushClone;
}
else
{
    // If the brush is not frozen,
    // it can be modified directly.
    myBrush.Color = Colors.Red;
}

```

```

Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)

If myBrush.CanFreeze Then
    ' Makes the brush unmodifiable.
    myBrush.Freeze()
End If

myButton.Background = myBrush

If myBrush.IsFrozen Then ' Evaluates to true.
    ' If the brush is frozen, create a clone and
    ' modify the clone.
    Dim myBrushClone As SolidColorBrush = myBrush.Clone()
    myBrushClone.Color = Colors.Red
    myButton.Background = myBrushClone
Else
    ' If the brush is not frozen,
    ' it can be modified directly.
    myBrush.Color = Colors.Red
End If

```

In the preceding code example, a modifiable copy was made of a frozen object using the [Clone](#) method. The next section discusses cloning in more detail.

## NOTE

Because a frozen [Freezable](#) cannot be animated, the animation system will automatically create modifiable clones of frozen [Freezable](#) objects when you try to animate them with a [Storyboard](#). To eliminate the performance overhead caused by cloning, leave an object unfrozen if you intend to animate it. For more information about animating with storyboards, see the [Storyboards Overview](#).

## Freezing from Markup

To freeze a [Freezable](#) object declared in markup, you use the `PresentationOptions:Freeze` attribute. In the following example, a [SolidColorBrush](#) is declared as a page resource and frozen. It is then used to set the background of a button.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:PresentationOptions="http://schemas.microsoft.com/winfx/2006/xaml/presentation/options"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="PresentationOptions">

    <Page.Resources>

        <!-- This resource is frozen. -->
        <SolidColorBrush
            x:Key="MyBrush"
            PresentationOptions:Freeze="True"
            Color="Red" />
    </Page.Resources>

    <StackPanel>

        <Button Content="A Button"
            Background="{StaticResource MyBrush}">
        </Button>

    </StackPanel>
</Page>
```

To use the `Freeze` attribute, you must map to the presentation options namespace:

`http://schemas.microsoft.com/winfx/2006/xaml/presentation/options`. `PresentationOptions` is the recommended prefix for mapping this namespace:

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

Because not all XAML readers recognize this attribute, it's recommended that you use the [mc:Ignorable Attribute](#) to mark the `Presentation:Freeze` attribute as ignorable:

```
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="PresentationOptions"
```

For more information, see the [mc:Ignorable Attribute](#) page.

## "Unfreezing" a Freezable

Once frozen, a [Freezable](#) can never be modified or unfrozen; however, you can create an unfrozen clone using the [Clone](#) or [CloneCurrentValue](#) method.

In the following example, the button's background is set with a brush and that brush is then frozen. An unfrozen

copy is made of the brush using the [Clone](#) method. The clone is modified and used to change the button's background from yellow to red.

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

// Freezing a Freezable before it provides
// performance improvements if you don't
// intend on modifying it.
if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

// If you need to modify a frozen brush,
// the Clone method can be used to
// create a modifiable copy.
SolidColorBrush myBrushClone = myBrush.Clone();

// Changing myBrushClone does not change
// the color of myButton, because its
// background is still set by myBrush.
myBrushClone.Color = Colors.Red;

// Replacing myBrush with myBrushClone
// makes the button change to red.
myButton.Background = myBrushClone;
```

```
Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)

' Freezing a Freezable before it provides
' performance improvements if you don't
' intend on modifying it.
If myBrush.CanFreeze Then
    ' Makes the brush unmodifiable.
    myBrush.Freeze()
End If

myButton.Background = myBrush

' If you need to modify a frozen brush,
' the Clone method can be used to
' create a modifiable copy.
Dim myBrushClone As SolidColorBrush = myBrush.Clone()

' Changing myBrushClone does not change
' the color of myButton, because its
' background is still set by myBrush.
myBrushClone.Color = Colors.Red

' Replacing myBrush with myBrushClone
' makes the button change to red.
myButton.Background = myBrushClone
```

#### NOTE

Regardless of which clone method you use, animations are never copied to the new [Freezable](#).

The [Clone](#) and [CloneCurrentValue](#) methods produce deep copies of the freezable. If the freezable contains other frozen freezable objects, they are also cloned and made modifiable. For example, if you clone a frozen [PathGeometry](#) to make it modifiable, the figures and segments it contains are also copied and made modifiable.

## Creating Your Own Freezable Class

A class that derives from [Freezable](#) gains the following features.

- Special states: a read-only (frozen) and a writable state.
- Thread safety: a frozen [Freezable](#) can be shared across threads.
- Detailed change notification: Unlike other [DependencyObjects](#), Freezable objects provide change notifications when sub-property values change.
- Easy cloning: the Freezable class has already implemented several methods that produce deep clones.

A [Freezable](#) is a type of [DependencyObject](#), and therefore uses the dependency property system. Your class properties don't have to be dependency properties, but using dependency properties will reduce the amount of code you have to write, because the [Freezable](#) class was designed with dependency properties in mind. For more information about the dependency property system, see the [Dependency Properties Overview](#).

Every [Freezable](#) subclass must override the [CreateInstanceCore](#) method. If your class uses dependency properties for all its data, you're finished.

If your class contains non-dependency property data members, you must also override the following methods:

- [CloneCore](#)
- [CloneCurrentValueCore](#)
- [GetAsFrozenCore](#)
- [GetCurrentValueAsFrozenCore](#)
- [FreezeCore](#)

You must also observe the following rules for accessing and writing to data members that are not dependency properties:

- At the beginning of any API that reads non-dependency property data members, call the [ReadPreamble](#) method.
- At the beginning of any API that writes non-dependency property data members, call the [WritePreamble](#) method. (Once you've called [WritePreamble](#) in an API, you don't need to make an additional call to [ReadPreamble](#) if you also read non-dependency property data members.)
- Call the [WritePostscript](#) method before exiting methods that write to non-dependency property data members.

If your class contains non-dependency-property data members that are [DependencyObject](#) objects, you must also call the [OnFreezablePropertyChanged](#) method each time you change one of their values, even if you're setting the member to `null`.

**NOTE**

It's very important that you begin each [Freezable](#) method you override with a call to the base implementation.

For an example of a custom [Freezable](#) class, see the [Custom Animation Sample](#).

## See also

- [Freezable](#)
- [Custom Animation Sample](#)
- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)

# Alignment, Margins, and Padding Overview

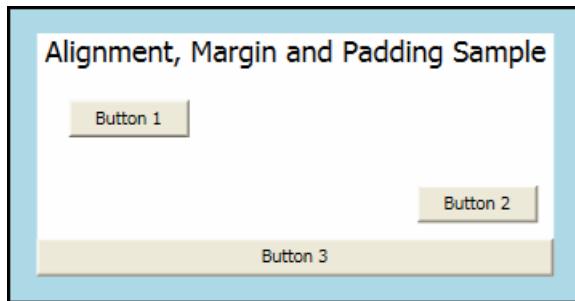
8/22/2019 • 19 minutes to read • [Edit Online](#)

The [FrameworkElement](#) class exposes several properties that are used to precisely position child elements. This topic discusses four of the most important properties: [HorizontalAlignment](#), [Margin](#), [Padding](#), and [VerticalAlignment](#). The effects of these properties are important to understand, because they provide the basis for controlling the position of elements in Windows Presentation Foundation (WPF) applications.

## Introduction to Element Positioning

There are numerous ways to position elements using WPF. However, achieving ideal layout goes beyond simply choosing the right [Panel](#) element. Fine control of positioning requires an understanding of the [HorizontalAlignment](#), [Margin](#), [Padding](#), and [VerticalAlignment](#) properties.

The following illustration shows a layout scenario that utilizes several positioning properties.



At first glance, the [Button](#) elements in this illustration may appear to be placed randomly. However, their positions are actually precisely controlled by using a combination of margins, alignments, and padding.

The following example describes how to create the layout in the preceding illustration. A [Border](#) element encapsulates a parent [StackPanel](#), with a [Padding](#) value of 15 device independent pixels. This accounts for the narrow [LightBlue](#) band that surrounds the child [StackPanel](#). Child elements of the [StackPanel](#) are used to illustrate each of the various positioning properties that are detailed in this topic. Three [Button](#) elements are used to demonstrate both the [Margin](#) and [HorizontalAlignment](#) properties.

```
// Create the application's main Window.  
mainWindow = new Window();  
mainWindow.Title = "Margins, Padding and Alignment Sample";  
  
// Add a Border  
myBorder = new Border();  
myBorder.Background = Brushes.LightBlue;  
myBorder.BorderBrush = Brushes.Black;  
myBorder.Padding = new Thickness(15);  
myBorder.BorderThickness = new Thickness(2);  
  
myStackPanel = new StackPanel();  
myStackPanel.Background = Brushes.White;  
myStackPanel.HorizontalAlignment = HorizontalAlignment.Center;  
myStackPanel.VerticalAlignment = VerticalAlignment.Top;  
  
TextBlock myTextBlock = new TextBlock();  
myTextBlock.Margin = new Thickness(5, 0, 5, 0);  
myTextBlock.FontSize = 18;  
myTextBlock.HorizontalAlignment = HorizontalAlignment.Center;  
myTextBlock.Text = "Alignment, Margin and Padding Sample";  
Button myButton1 = new Button();  
myButton1.HorizontalAlignment = HorizontalAlignment.Left;  
myButton1.Margin = new Thickness(20);  
myButton1.Content = "Button 1";  
Button myButton2 = new Button();  
myButton2.HorizontalAlignment = HorizontalAlignment.Right;  
myButton2.Margin = new Thickness(10);  
myButton2.Content = "Button 2";  
Button myButton3 = new Button();  
myButton3.HorizontalAlignment = HorizontalAlignment.Stretch;  
myButton3.Margin = new Thickness(0);  
myButton3.Content = "Button 3";  
  
// Add child elements to the parent StackPanel.  
myStackPanel.Children.Add(myTextBlock);  
myStackPanel.Children.Add(myButton1);  
myStackPanel.Children.Add(myButton2);  
myStackPanel.Children.Add(myButton3);  
  
// Add the StackPanel as the lone Child of the Border.  
myBorder.Child = myStackPanel;  
  
// Add the Border as the Content of the Parent Window Object.  
mainWindow.Content = myBorder;  
mainWindow.Show();
```

```

WindowTitle = "Margins, Padding and Alignment Sample"

'Add a Border.
Dim myBorder As New Border()
myBorder.Background = Brushes.LightBlue
myBorder.BorderBrush = Brushes.Black
myBorder.Padding = New Thickness(15)
myBorder.BorderThickness = New Thickness(2)

Dim myStackPanel As New StackPanel()
myStackPanel.Background = Brushes.White
myStackPanel.HorizontalAlignment = Windows.HorizontalAlignment.Center
myStackPanel.VerticalAlignment = Windows.VerticalAlignment.Top

Dim myTextBlock As New TextBlock()
myTextBlock.Margin = New Thickness(5, 0, 5, 0)
myTextBlock.FontSize = 18
myTextBlock.HorizontalAlignment = Windows.HorizontalAlignment.Center
myTextBlock.Text = "Alignment, Margin and Padding Sample"
Dim myButton1 As New Button()
myButton1.HorizontalAlignment = Windows.HorizontalAlignment.Left
myButton1.Margin = New Thickness(20)
myButton1.Content = "Button 1"
Dim myButton2 As New Button()
myButton2.HorizontalAlignment = Windows.HorizontalAlignment.Right
myButton2.Margin = New Thickness(10)
myButton2.Content = "Button 2"
Dim myButton3 As New Button()
myButton3.HorizontalAlignment = Windows.HorizontalAlignment.Stretch
myButton3.Margin = New Thickness(0)
myButton3.Content = "Button 3"

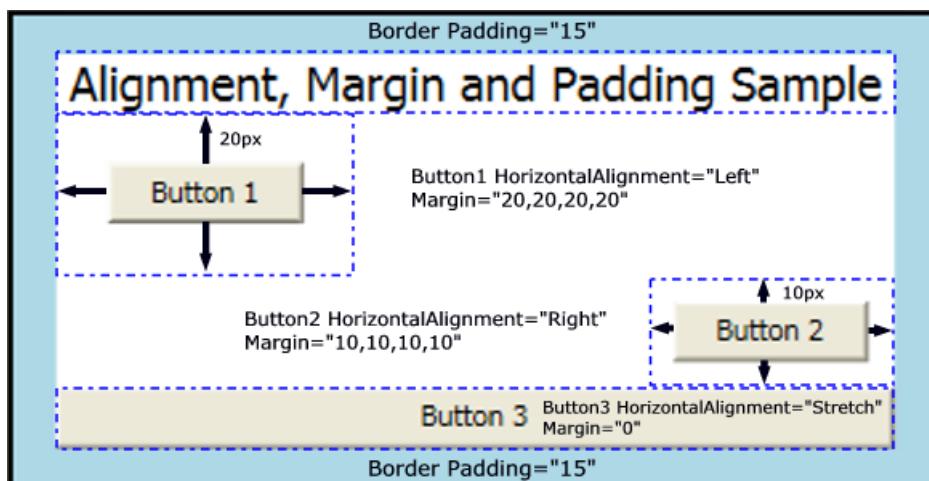
'Add child elements to the parent StackPanel.
myStackPanel.Children.Add(myTextBlock)
myStackPanel.Children.Add(myButton1)
myStackPanel.Children.Add(myButton2)
myStackPanel.Children.Add(myButton3)

'Add the StackPanel as the lone Child of the Border.
myBorder.Child = myStackPanel

' Add the Canvas as the lone Child of the Border
myBorder.Child = myStackPanel
Me.Content = myBorder

```

The following diagram provides a close-up view of the various positioning properties that are used in the preceding sample. Subsequent sections in this topic describe in greater detail how to use each positioning property.



# Understanding Alignment Properties

The [HorizontalAlignment](#) and [VerticalAlignment](#) properties describe how a child element should be positioned within a parent element's allocated layout space. By using these properties together, you can position child elements precisely. For example, child elements of a [DockPanel](#) can specify four different horizontal alignments: [Left](#), [Right](#), or [Center](#), or to [Stretch](#) to fill available space. Similar values are available for vertical positioning.

## NOTE

Explicitly-set [Height](#) and [Width](#) properties on an element take precedence over the [Stretch](#) property value. Attempting to set [Height](#), [Width](#), and a [HorizontalAlignment](#) value of [Stretch](#) results in the [Stretch](#) request being ignored.

## HorizontalAlignment Property

The [HorizontalAlignment](#) property declares the horizontal alignment characteristics to apply to child elements. The following table shows each of the possible values of the [HorizontalAlignment](#) property.

MEMBER	DESCRIPTION
<a href="#">Left</a>	Child elements are aligned to the left of the parent element's allocated layout space.
<a href="#">Center</a>	Child elements are aligned to the center of the parent element's allocated layout space.
<a href="#">Right</a>	Child elements are aligned to the right of the parent element's allocated layout space.
<a href="#">Stretch</a> (Default)	Child elements are stretched to fill the parent element's allocated layout space. Explicit <a href="#">Width</a> and <a href="#">Height</a> values take precedence.

The following example shows how to apply the [HorizontalAlignment](#) property to [Button](#) elements. Each attribute value is shown, to better illustrate the various rendering behaviors.

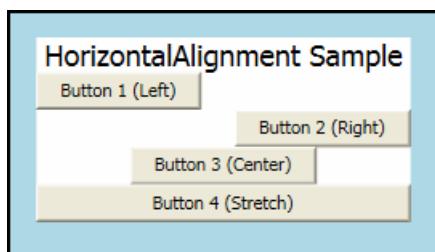
```
Button myButton1 = new Button();
myButton1.HorizontalAlignment = HorizontalAlignment.Left;
myButton1.Content = "Button 1 (Left)";
Button myButton2 = new Button();
myButton2.HorizontalAlignment = HorizontalAlignment.Right;
myButton2.Content = "Button 2 (Right)";
Button myButton3 = new Button();
myButton3.HorizontalAlignment = HorizontalAlignment.Center;
myButton3.Content = "Button 3 (Center)";
Button myButton4 = new Button();
myButton4.HorizontalAlignment = HorizontalAlignment.Stretch;
myButton4.Content = "Button 4 (Stretch);
```

```

Dim myButton1 As New Button()
myButton1.HorizontalAlignment = Windows.HorizontalAlignment.Left
myButton1.Margin = New Thickness(20)
myButton1.Content = "Button 1"
Dim myButton2 As New Button()
myButton2.HorizontalAlignment = Windows.HorizontalAlignment.Right
myButton2.Margin = New Thickness(10)
myButton2.Content = "Button 2"
Dim myButton3 As New Button()
myButton3.HorizontalAlignment = Windows.HorizontalAlignment.Center
myButton3.Margin = New Thickness(0)
myButton3.Content = "Button 3"
Dim myButton4 As New Button()
myButton4.HorizontalAlignment = Windows.HorizontalAlignment.Stretch
myButton4.Content = "Button 4 (Stretch)"

```

The preceding code yields a layout similar to the following image. The positioning effects of each [HorizontalAlignment](#) value are visible in the illustration.



## VerticalAlignment Property

The [VerticalAlignment](#) property describes the vertical alignment characteristics to apply to child elements. The following table shows each of the possible values for the [VerticalAlignment](#) property.

MEMBER	DESCRIPTION
<a href="#">Top</a>	Child elements are aligned to the top of the parent element's allocated layout space.
<a href="#">Center</a>	Child elements are aligned to the center of the parent element's allocated layout space.
<a href="#">Bottom</a>	Child elements are aligned to the bottom of the parent element's allocated layout space.
<a href="#">Stretch</a> (Default)	Child elements are stretched to fill the parent element's allocated layout space. Explicit <a href="#">Width</a> and <a href="#">Height</a> values take precedence.

The following example shows how to apply the [VerticalAlignment](#) property to [Button](#) elements. Each attribute value is shown, to better illustrate the various rendering behaviors. For purposes of this sample, a [Grid](#) element with visible gridlines is used as the parent, to better illustrate the layout behavior of each property value.

```
TextBlock myTextBlock = new TextBlock();
myTextBlock.FontSize = 18;
myTextBlock.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock.Text = "VerticalAlignment Sample";
Grid.SetRow(myTextBlock, 0);
Button myButton1 = new Button();
myButton1.VerticalAlignment = VerticalAlignment.Top;
myButton1.Content = "Button 1 (Top)";
Grid.SetRow(myButton1, 1);
Button myButton2 = new Button();
myButton2.VerticalAlignment = VerticalAlignment.Bottom;
myButton2.Content = "Button 2 (Bottom)";
Grid.SetRow(myButton2, 2);
Button myButton3 = new Button();
myButton3.VerticalAlignment = VerticalAlignment.Center;
myButton3.Content = "Button 3 (Center)";
Grid.SetRow(myButton3, 3);
Button myButton4 = new Button();
myButton4.VerticalAlignment = VerticalAlignment.Stretch;
myButton4.Content = "Button 4 (Stretch)";
Grid.SetRow(myButton4, 4);
```

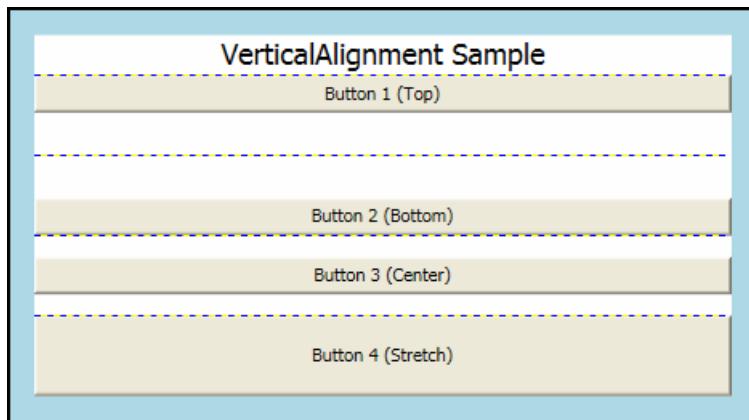
```
Dim myTextBlock As New TextBlock()
myTextBlock.FontSize = 18
myTextBlock.HorizontalAlignment = Windows.HorizontalAlignment.Center
myTextBlock.Text = "VerticalAlignment Sample"
Grid.SetRow(myTextBlock, 0)
Dim myButton1 As New Button()
myButton1.VerticalAlignment = Windows.VerticalAlignment.Top
myButton1.Content = "Button 1 (Top)"
Grid.SetRow(myButton1, 1)
Dim myButton2 As New Button()
myButton2.VerticalAlignment = Windows.VerticalAlignment.Bottom
myButton2.Content = "Button 2 (Bottom)"
Grid.SetRow(myButton2, 2)
Dim myButton3 As New Button()
myButton3.VerticalAlignment = Windows.VerticalAlignment.Center
myButton3.Content = "Button 3 (Center)"
Grid.SetRow(myButton3, 3)
Dim myButton4 As New Button()
myButton4.VerticalAlignment = Windows.VerticalAlignment.Stretch
myButton4.Content = "Button 4 (Stretch)"
Grid.SetRow(myButton4, 4)
```

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      WindowTitle="VerticalAlignment Sample">
    <Border Background="LightBlue" BorderBrush="Black" BorderThickness="2" Padding="15">
        <Grid Background="White" ShowGridLines="True">
            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
                <RowDefinition Height="50"/>
                <RowDefinition Height="50"/>
                <RowDefinition Height="50"/>
                <RowDefinition Height="50"/>
            </Grid.RowDefinitions>
            <TextBlock Grid.Row="0" Grid.Column="0" FontSize="18"
HorizontalAlignment="Center">VerticalAlignment Sample</TextBlock>
            <Button Grid.Row="1" Grid.Column="0" VerticalAlignment="Top">Button 1 (Top)</Button>
            <Button Grid.Row="2" Grid.Column="0" VerticalAlignment="Bottom">Button 2 (Bottom)</Button>
            <Button Grid.Row="3" Grid.Column="0" VerticalAlignment="Center">Button 3 (Center)</Button>
            <Button Grid.Row="4" Grid.Column="0" VerticalAlignment="Stretch">Button 4 (Stretch)</Button>
        </Grid>
    </Border>
</Page>

```

The preceding code yields a layout similar to the following image. The positioning effects of each **VerticalAlignment** value are visible in the illustration.



## Understanding Margin Properties

The **Margin** property describes the distance between an element and its child or peers. **Margin** values can be uniform, by using syntax like `Margin="20"`. With this syntax, a uniform **Margin** of 20 device independent pixels would be applied to the element. **Margin** values can also take the form of four distinct values, each value describing a distinct margin to apply to the left, top, right, and bottom (in that order), like `Margin="0,10,5,25"`. Proper use of the **Margin** property enables very fine control of an element's rendering position and the rendering position of its neighbor elements and children.

### NOTE

A non-zero margin applies space outside the element's **ActualWidth** and **ActualHeight**.

The following example shows how to apply uniform margins around a group of **Button** elements. The **Button** elements are spaced evenly with a ten-pixel margin buffer in each direction.

```
Button^ myButton7 = gcnew Button();
myButton7->Margin = Thickness(10);
myButton7->Content = "Button 7";
Button^ myButton8 = gcnew Button();
myButton8->Margin = Thickness(10);
myButton8->Content = "Button 8";
Button^ myButton9 = gcnew Button();
myButton9->Margin = Thickness(10);
myButton9->Content = "Button 9";
```

```
Button myButton7 = new Button();
myButton7.Margin = new Thickness(10);
myButton7.Content = "Button 7";
Button myButton8 = new Button();
myButton8.Margin = new Thickness(10);
myButton8.Content = "Button 8";
Button myButton9 = new Button();
myButton9.Margin = new Thickness(10);
myButton9.Content = "Button 9";
```

```
Dim myButton7 As New Button
myButton7.Margin = New Thickness(10)
myButton7.Content = "Button 7"
Dim myButton8 As New Button
myButton8.Margin = New Thickness(10)
myButton8.Content = "Button 8"
Dim myButton9 As New Button
myButton9.Margin = New Thickness(10)
myButton9.Content = "Button 9"
```

```
<Button Margin="10">Button 7</Button>
<Button Margin="10">Button 8</Button>
<Button Margin="10">Button 9</Button>
```

In many instances, a uniform margin is not appropriate. In these cases, non-uniform spacing can be applied. The following example shows how to apply non-uniform margin spacing to child elements. Margins are described in this order: left, top, right, bottom.

```
Button^ myButton1 = gcnew Button();
myButton1->Margin = Thickness(0, 10, 0, 10);
myButton1->Content = "Button 1";
Button^ myButton2 = gcnew Button();
myButton2->Margin = Thickness(0, 10, 0, 10);
myButton2->Content = "Button 2";
Button^ myButton3 = gcnew Button();
myButton3->Margin = Thickness(0, 10, 0, 10);
```

```
Button myButton1 = new Button();
myButton1.Margin = new Thickness(0, 10, 0, 10);
myButton1.Content = "Button 1";
Button myButton2 = new Button();
myButton2.Margin = new Thickness(0, 10, 0, 10);
myButton2.Content = "Button 2";
Button myButton3 = new Button();
myButton3.Margin = new Thickness(0, 10, 0, 10);
```

```
Dim myButton1 As New Button  
myButton1.Margin = New Thickness(0, 10, 0, 10)  
myButton1.Content = "Button 1"  
Dim myButton2 As New Button  
myButton2.Margin = New Thickness(0, 10, 0, 10)  
myButton2.Content = "Button 2"  
Dim myButton3 As New Button  
myButton3.Margin = New Thickness(0, 10, 0, 10)
```

```
<Button Margin="0,10,0,10">Button 1</Button>  
<Button Margin="0,10,0,10">Button 2</Button>  
<Button Margin="0,10,0,10">Button 3</Button>
```

## Understanding the Padding Property

Padding is similar to [Margin](#) in most respects. The Padding property is exposed on only on a few classes, primarily as a convenience: [Block](#), [Border](#), [Control](#), and [TextBlock](#) are samples of classes that expose a Padding property. The [Padding](#) property enlarges the effective size of a child element by the specified [Thickness](#) value.

The following example shows how to apply [Padding](#) to a parent [Border](#) element.

```
myBorder = gcnew Border();  
myBorder->Background = Brushes::LightBlue;  
myBorder->BorderBrush = Brushes::Black;  
myBorder->BorderThickness = Thickness(2);  
myBorder->CornerRadius = CornerRadius(45);  
myBorder->Padding = Thickness(25);
```

```
myBorder = new Border();  
myBorder.Background = Brushes.LightBlue;  
myBorder.BorderBrush = Brushes.Black;  
myBorder.BorderThickness = new Thickness(2);  
myBorder.CornerRadius = new CornerRadius(45);  
myBorder.Padding = new Thickness(25);
```

```
Dim myBorder As New Border  
myBorder.Background = Brushes.LightBlue  
myBorder.BorderBrush = Brushes.Black  
myBorder.BorderThickness = New Thickness(2)  
myBorder.CornerRadius = New CornerRadius(45)  
myBorder.Padding = New Thickness(25)
```

```
<Border Background="LightBlue"  
       BorderBrush="Black"  
       BorderThickness="2"  
       CornerRadius="45"  
       Padding="25">
```

## Using Alignment, Margins, and Padding in an Application

[HorizontalAlignment](#), [Margin](#), [Padding](#), and [VerticalAlignment](#) provide the positioning control necessary to create a complex user interface (UI). You can use the effects of each property to change child-element positioning, enabling flexibility in creating dynamic applications and user experiences.

The following example demonstrates each of the concepts that are detailed in this topic. Building on the infrastructure found in the first sample in this topic, this example adds a [Grid](#) element as a child of the [Border](#) in the first sample. [Padding](#) is applied to the parent [Border](#) element. The [Grid](#) is used to partition space between three child [StackPanel](#) elements. [Button](#) elements are again used to show the various effects of [Margin](#) and [HorizontalAlignment](#). [TextBlock](#) elements are added to each [ColumnDefinition](#) to better define the various properties applied to the [Button](#) elements in each column.

```
mainWindow = gcnew Window();

myBorder = gcnew Border();
myBorder->Background = Brushes::LightBlue;
myBorder->BorderBrush = Brushes::Black;
myBorder->BorderThickness = Thickness(2);
myBorder->CornerRadius = CornerRadius(45);
myBorder->Padding = Thickness(25);

// Define the Grid.
myGrid = gcnew Grid();
myGrid->Background = Brushes::White;
myGrid->ShowGridLines = true;

// Define the Columns.
ColumnDefinition^ myColDef1 = gcnew ColumnDefinition();
myColDef1->Width = GridLength(1, GridUnitType::Auto);
ColumnDefinition^ myColDef2 = gcnew ColumnDefinition();
myColDef2->Width = GridLength(1, GridUnitType::Star);
ColumnDefinition^ myColDef3 = gcnew ColumnDefinition();
myColDef3->Width = GridLength(1, GridUnitType::Auto);

// Add the ColumnDefinitions to the Grid.
myGrid->ColumnDefinitions->Add(myColDef1);
myGrid->ColumnDefinitions->Add(myColDef2);
myGrid->ColumnDefinitions->Add(myColDef3);

// Add the first child StackPanel.
StackPanel^ myStackPanel = gcnew StackPanel();
myStackPanel->HorizontalAlignment = HorizontalAlignment::Left;
myStackPanel->VerticalAlignment = VerticalAlignment::Top;
Grid::SetColumn(myStackPanel, 0);
Grid::SetRow(myStackPanel, 0);
TextBlock^ myTextBlock1 = gcnew TextBlock();
myTextBlock1->FontSize = 18;
myTextBlock1->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock1->Margin = Thickness(0, 0, 0, 15);
myTextBlock1->Text = "StackPanel 1";
Button^ myButton1 = gcnew Button();
myButton1->Margin = Thickness(0, 10, 0, 10);
myButton1->Content = "Button 1";
Button^ myButton2 = gcnew Button();
myButton2->Margin = Thickness(0, 10, 0, 10);
myButton2->Content = "Button 2";
Button^ myButton3 = gcnew Button();
myButton3->Margin = Thickness(0, 10, 0, 10);
TextBlock^ myTextBlock2 = gcnew TextBlock();
myTextBlock2->Text = "ColumnDefinition.Width = \"Auto\"";
TextBlock^ myTextBlock3 = gcnew TextBlock();
myTextBlock3->Text = "StackPanel.HorizontalAlignment = \"Left\"";
TextBlock^ myTextBlock4 = gcnew TextBlock();
myTextBlock4->Text = "StackPanel.VerticalAlignment = \"Top\"";
TextBlock^ myTextBlock5 = gcnew TextBlock();
myTextBlock5->Text = "StackPanel.Orientation = \"Vertical\"";
TextBlock^ myTextBlock6 = gcnew TextBlock();
myTextBlock6->Text = "Button.Margin = \"1,10,0,10\"";
myStackPanel->Children->Add(myTextBlock1);
myStackPanel->Children->Add(myButton1);
myStackPanel->Children->Add(myButton2);
```

```

myStackPanel->Children->Add(myButton3);
myStackPanel->Children->Add(myTextBlock2);
myStackPanel->Children->Add(myTextBlock3);
myStackPanel->Children->Add(myTextBlock4);
myStackPanel->Children->Add(myTextBlock5);
myStackPanel->Children->Add(myTextBlock6);

// Add the second child StackPanel.
StackPanel^ myStackPanel2 = gcnew StackPanel();
myStackPanel2->HorizontalAlignment = HorizontalAlignment::Stretch;
myStackPanel2->VerticalAlignment = VerticalAlignment::Top;
myStackPanel2->Orientation = Orientation::Vertical;
Grid::SetColumn(myStackPanel2, 1);
Grid::SetRow(myStackPanel2, 0);
TextBlock^ myTextBlock7 = gcnew TextBlock();
myTextBlock7->FontSize = 18;
myTextBlock7->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock7->Margin = Thickness(0, 0, 0, 15);
myTextBlock7->Text = "StackPanel 2";
Button^ myButton4 = gcnew Button();
myButton4->Margin = Thickness(10, 0, 10, 0);
myButton4->Content = "Button 4";
Button^ myButton5 = gcnew Button();
myButton5->Margin = Thickness(10, 0, 10, 0);
myButton5->Content = "Button 5";
Button^ myButton6 = gcnew Button();
myButton6->Margin = Thickness(10, 0, 10, 0);
myButton6->Content = "Button 6";
TextBlock^ myTextBlock8 = gcnew TextBlock();
myTextBlock8->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock8->Text = "ColumnDefinition.Width = \\"*\\\"";
TextBlock^ myTextBlock9 = gcnew TextBlock();
myTextBlock9->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock9->Text = "StackPanel.HorizontalAlignment = \"Stretch\"";
TextBlock^ myTextBlock10 = gcnew TextBlock();
myTextBlock10->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock10->Text = "StackPanel.VerticalAlignment = \"Top\"";
TextBlock^ myTextBlock11 = gcnew TextBlock();
myTextBlock11->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock11->Text = "StackPanel.Orientation = \"Horizontal\"";
TextBlock^ myTextBlock12 = gcnew TextBlock();
myTextBlock12->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock12->Text = "Button.Margin = \"10,0,10,0\"";
myStackPanel2->Children->Add(myTextBlock7);
myStackPanel2->Children->Add(myButton4);
myStackPanel2->Children->Add(myButton5);
myStackPanel2->Children->Add(myButton6);
myStackPanel2->Children->Add(myTextBlock8);
myStackPanel2->Children->Add(myTextBlock9);
myStackPanel2->Children->Add(myTextBlock10);
myStackPanel2->Children->Add(myTextBlock11);
myStackPanel2->Children->Add(myTextBlock12);

// Add the final child StackPanel.
StackPanel^ myStackPanel3 = gcnew StackPanel();
myStackPanel3->HorizontalAlignment = HorizontalAlignment::Left;
myStackPanel3->VerticalAlignment = VerticalAlignment::Top;
Grid::SetColumn(myStackPanel3, 2);
Grid::SetRow(myStackPanel3, 0);
TextBlock^ myTextBlock13 = gcnew TextBlock();
myTextBlock13->FontSize = 18;
myTextBlock13->HorizontalAlignment = HorizontalAlignment::Center;
myTextBlock13->Margin = Thickness(0, 0, 0, 15);
myTextBlock13->Text = "StackPanel 3";
Button^ myButton7 = gcnew Button();
myButton7->Margin = Thickness(10);
myButton7->Content = "Button 7";
Button^ myButton8 = gcnew Button();
myButton8->Margin = Thickness(10);

```

```

myButton8->Content = "Button 8";
Button^ myButton9 = gcnew Button();
myButton9->Margin = Thickness(10);
myButton9->Content = "Button 9";
TextBlock^ myTextBlock14 = gcnew TextBlock();
myTextBlock14->Text = "ColumnDefinition.Width = \"Auto\"";
TextBlock^ myTextBlock15 = gcnew TextBlock();
myTextBlock15->Text = "StackPanel.HorizontalAlignment = \"Left\"";
TextBlock^ myTextBlock16 = gcnew TextBlock();
myTextBlock16->Text = "StackPanel.VerticalAlignment = \"Top\"";
TextBlock^ myTextBlock17 = gcnew TextBlock();
myTextBlock17->Text = "StackPanel.Orientation = \"Vertical\"";
TextBlock^ myTextBlock18 = gcnew TextBlock();
myTextBlock18->Text = "Button.Margin = \"10\"";
myStackPanel3->Children->Add(myTextBlock13);
myStackPanel3->Children->Add(myButton7);
myStackPanel3->Children->Add(myButton8);
myStackPanel3->Children->Add(myButton9);
myStackPanel3->Children->Add(myTextBlock14);
myStackPanel3->Children->Add(myTextBlock15);
myStackPanel3->Children->Add(myTextBlock16);
myStackPanel3->Children->Add(myTextBlock17);
myStackPanel3->Children->Add(myTextBlock18);

// Add child content to the parent Grid.
myGrid->Children->Add(myStackPanel);
myGrid->Children->Add(myStackPanel2);
myGrid->Children->Add(myStackPanel3);

// Add the Grid as the lone child of the Border.
myBorder->Child = myGrid;

// Add the Border to the Window as Content and show the Window.
mainWindow->Content = myBorder;
mainWindow->Title = "Margin, Padding, and Alignment Sample";
mainWindow->Show();

```

```

mainWindow = new Window();

myBorder = new Border();
myBorder.Background = Brushes.LightBlue;
myBorder.BorderBrush = Brushes.Black;
myBorder.BorderThickness = new Thickness(2);
myBorder.CornerRadius = new CornerRadius(45);
myBorder.Padding = new Thickness(25);

// Define the Grid.
myGrid = new Grid();
myGrid.Background = Brushes.White;
myGrid.ShowGridLines = true;

// Define the Columns.
ColumnDefinition myColDef1 = new ColumnDefinition();
myColDef1.Width = new GridLength(1, GridUnitType.Auto);
ColumnDefinition myColDef2 = new ColumnDefinition();
myColDef2.Width = new GridLength(1, GridUnitType.Star);
ColumnDefinition myColDef3 = new ColumnDefinition();
myColDef3.Width = new GridLength(1, GridUnitType.Auto);

// Add the ColumnDefinitions to the Grid.
myGrid.ColumnDefinitions.Add(myColDef1);
myGrid.ColumnDefinitions.Add(myColDef2);
myGrid.ColumnDefinitions.Add(myColDef3);

// Add the first child StackPanel.
StackPanel myStackPanel = new StackPanel();
myStackPanel.HorizontalAlignment = HorizontalAlignment.Left;

```

```

myStackPanel.VerticalAlignment = VerticalAlignment.Top;
Grid.SetColumn(myStackPanel, 0);
Grid.SetRow(myStackPanel, 0);
TextBlock myTextBlock1 = new TextBlock();
myTextBlock1.FontSize = 18;
myTextBlock1.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock1.Margin = new Thickness(0, 0, 0, 15);
myTextBlock1.Text = "StackPanel 1";
Button myButton1 = new Button();
myButton1.Margin = new Thickness(0, 10, 0, 10);
myButton1.Content = "Button 1";
Button myButton2 = new Button();
myButton2.Margin = new Thickness(0, 10, 0, 10);
myButton2.Content = "Button 2";
Button myButton3 = new Button();
myButton3.Margin = new Thickness(0, 10, 0, 10);
TextBlock myTextBlock2 = new TextBlock();
myTextBlock2.Text = @"ColumnDefinition.Width = ""Auto""";
TextBlock myTextBlock3 = new TextBlock();
myTextBlock3.Text = @"StackPanel.HorizontalAlignment = ""Left""";
TextBlock myTextBlock4 = new TextBlock();
myTextBlock4.Text = @"StackPanel.VerticalAlignment = ""Top""";
TextBlock myTextBlock5 = new TextBlock();
myTextBlock5.Text = @"StackPanel.Orientation = ""Vertical""";
TextBlock myTextBlock6 = new TextBlock();
myTextBlock6.Text = @"Button.Margin = ""1,10,0,10""";
myStackPanel.Children.Add(myTextBlock1);
myStackPanel.Children.Add(myButton1);
myStackPanel.Children.Add(myButton2);
myStackPanel.Children.Add(myButton3);
myStackPanel.Children.Add(myTextBlock2);
myStackPanel.Children.Add(myTextBlock3);
myStackPanel.Children.Add(myTextBlock4);
myStackPanel.Children.Add(myTextBlock5);
myStackPanel.Children.Add(myTextBlock6);

// Add the second child StackPanel.
StackPanel myStackPanel2 = new StackPanel();
myStackPanel2.HorizontalAlignment = HorizontalAlignment.Stretch;
myStackPanel2.VerticalAlignment = VerticalAlignment.Top;
myStackPanel2.Orientation = Orientation.Vertical;
Grid.SetColumn(myStackPanel2, 1);
Grid.SetRow(myStackPanel2, 0);
TextBlock myTextBlock7 = new TextBlock();
myTextBlock7.FontSize = 18;
myTextBlock7.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock7.Margin = new Thickness(0, 0, 0, 15);
myTextBlock7.Text = "StackPanel 2";
Button myButton4 = new Button();
myButton4.Margin = new Thickness(10, 0, 10, 0);
myButton4.Content = "Button 4";
Button myButton5 = new Button();
myButton5.Margin = new Thickness(10, 0, 10, 0);
myButton5.Content = "Button 5";
Button myButton6 = new Button();
myButton6.Margin = new Thickness(10, 0, 10, 0);
myButton6.Content = "Button 6";
TextBlock myTextBlock8 = new TextBlock();
myTextBlock8.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock8.Text = @"ColumnDefinition.Width = ""*""";
TextBlock myTextBlock9 = new TextBlock();
myTextBlock9.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock9.Text = @"StackPanel.HorizontalAlignment = ""Stretch""";
TextBlock myTextBlock10 = new TextBlock();
myTextBlock10.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock10.Text = @"StackPanel.VerticalAlignment = ""Top""";
TextBlock myTextBlock11 = new TextBlock();
myTextBlock11.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock11.Text = @"StackPanel.Orientation = ""Horizontal""";

```

```

TextBlock myTextBlock12 = new TextBlock();
myTextBlock12.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock12.Text = @"Button.Margin = "10,0,10,0""";
myStackPanel2.Children.Add(myTextBlock7);
myStackPanel2.Children.Add(myButton4);
myStackPanel2.Children.Add(myButton5);
myStackPanel2.Children.Add(myButton6);
myStackPanel2.Children.Add(myTextBlock8);
myStackPanel2.Children.Add(myTextBlock9);
myStackPanel2.Children.Add(myTextBlock10);
myStackPanel2.Children.Add(myTextBlock11);
myStackPanel2.Children.Add(myTextBlock12);

// Add the final child StackPanel.
StackPanel myStackPanel3 = new StackPanel();
myStackPanel3.HorizontalAlignment = HorizontalAlignment.Left;
myStackPanel3.VerticalAlignment = VerticalAlignment.Top;
Grid.SetColumn(myStackPanel3, 2);
Grid.SetRow(myStackPanel3, 0);
TextBlock myTextBlock13 = new TextBlock();
myTextBlock13.FontSize = 18;
myTextBlock13.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock13.Margin = new Thickness(0, 0, 0, 15);
myTextBlock13.Text = "StackPanel 3";
Button myButton7 = new Button();
myButton7.Margin = new Thickness(10);
myButton7.Content = "Button 7";
Button myButton8 = new Button();
myButton8.Margin = new Thickness(10);
myButton8.Content = "Button 8";
Button myButton9 = new Button();
myButton9.Margin = new Thickness(10);
myButton9.Content = "Button 9";
TextBlock myTextBlock14 = new TextBlock();
myTextBlock14.Text = @"ColumnDefinition.Width = ""Auto""";
TextBlock myTextBlock15 = new TextBlock();
myTextBlock15.Text = @"StackPanel.HorizontalAlignment = ""Left""";
TextBlock myTextBlock16 = new TextBlock();
myTextBlock16.Text = @"StackPanel.VerticalAlignment = ""Top""";
TextBlock myTextBlock17 = new TextBlock();
myTextBlock17.Text = @"StackPanel.Orientation = ""Vertical""";
TextBlock myTextBlock18 = new TextBlock();
myTextBlock18.Text = @"Button.Margin = ""10""";
myStackPanel3.Children.Add(myTextBlock13);
myStackPanel3.Children.Add(myButton7);
myStackPanel3.Children.Add(myButton8);
myStackPanel3.Children.Add(myButton9);
myStackPanel3.Children.Add(myTextBlock14);
myStackPanel3.Children.Add(myTextBlock15);
myStackPanel3.Children.Add(myTextBlock16);
myStackPanel3.Children.Add(myTextBlock17);
myStackPanel3.Children.Add(myTextBlock18);

// Add child content to the parent Grid.
myGrid.Children.Add(myStackPanel1);
myGrid.Children.Add(myStackPanel2);
myGrid.Children.Add(myStackPanel3);

// Add the Grid as the lone child of the Border.
myBorder.Child = myGrid;

// Add the Border to the Window as Content and show the Window.
mainWindow.Content = myBorder;
mainWindow.Title = "Margin, Padding, and Alignment Sample";
mainWindow.Show();

```

```

Dim myBorder As New Border
myBorder.Background = Brushes.LightBlue
myBorder.BorderBrush = Brushes.Black
myBorder.BorderThickness = New Thickness(2)
myBorder.CornerRadius = New CornerRadius(45)
myBorder.Padding = New Thickness(25)

'Define the Grid.
Dim myGrid As New Grid
myGrid.Background = Brushes.White
myGrid.ShowGridLines = True

'Define the Columns.
Dim myColDef1 As New ColumnDefinition
myColDef1.Width = New GridLength(1, GridUnitType.Auto)
Dim myColDef2 As New ColumnDefinition
myColDef2.Width = New GridLength(1, GridUnitType.Star)
Dim myColDef3 As New ColumnDefinition
myColDef3.Width = New GridLength(1, GridUnitType.Auto)

'Add the ColumnDefinitions to the Grid
myGrid.ColumnDefinitions.Add(myColDef1)
myGrid.ColumnDefinitions.Add(myColDef2)
myGrid.ColumnDefinitions.Add(myColDef3)

'Add the first child StackPanel.
Dim myStackPanel As New StackPanel
myStackPanel.HorizontalAlignment = System.Windows.HorizontalAlignment.Left
myStackPanel.VerticalAlignment = System.Windows.VerticalAlignment.Top
Grid.SetColumn(myStackPanel, 0)
Grid.SetRow(myStackPanel, 0)
Dim myTextBlock1 As New TextBlock
myTextBlock1.FontSize = 18
myTextBlock1.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock1.Margin = New Thickness(0, 0, 0, 15)
myTextBlock1.Text = "StackPanel 1"

Dim myButton1 As New Button
myButton1.Margin = New Thickness(0, 10, 0, 10)
myButton1.Content = "Button 1"
Dim myButton2 As New Button
myButton2.Margin = New Thickness(0, 10, 0, 10)
myButton2.Content = "Button 2"
Dim myButton3 As New Button
myButton3.Margin = New Thickness(0, 10, 0, 10)

Dim myTextBlock2 As New TextBlock
myTextBlock2.Text = "ColumnDefinition.Width = ""Auto"""
Dim myTextBlock3 As New TextBlock
myTextBlock3.Text = "StackPanel.HorizontalAlignment = ""Left"""
Dim myTextBlock4 As New TextBlock
myTextBlock4.Text = "StackPanel.VerticalAlignment = ""Top"""
Dim myTextBlock5 As New TextBlock
myTextBlock5.Text = "StackPanel.Orientation = ""Vertical"""
Dim myTextBlock6 As New TextBlock
myTextBlock6.Text = "Button.Margin = ""1,10,0,10"""
myStackPanel.Children.Add(myTextBlock1)
myStackPanel.Children.Add(myButton1)
myStackPanel.Children.Add(myButton2)
myStackPanel.Children.Add(myButton3)
myStackPanel.Children.Add(myTextBlock2)
myStackPanel.Children.Add(myTextBlock3)
myStackPanel.Children.Add(myTextBlock4)
myStackPanel.Children.Add(myTextBlock5)
myStackPanel.Children.Add(myTextBlock6)

'Add the second child StackPanel.
Dim myStackPanel2 As New StackPanel
myStackPanel2.HorizontalAlignment = System.Windows.HorizontalAlignment.Stretch

```

```

myStackPanel2.VerticalAlignment = System.Windows.VerticalAlignment.Top
myStackPanel2.Orientation = Orientation.Vertical
Grid.SetColumn(myStackPanel2, 1)
Grid.SetRow(myStackPanel2, 0)
Dim myTextBlock7 As New TextBlock
myTextBlock7.FontSize = 18
myTextBlock7.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock7.Margin = New Thickness(0, 0, 0, 15)
myTextBlock7.Text = "StackPanel 2"
Dim myButton4 As New Button
myButton4.Margin = New Thickness(10, 0, 10, 0)
myButton4.Content = "Button 4"
Dim myButton5 As New Button
myButton5.Margin = New Thickness(10, 0, 10, 0)
myButton5.Content = "Button 5"
Dim myButton6 As New Button
myButton6.Margin = New Thickness(10, 0, 10, 0)
myButton6.Content = "Button 6"
Dim myTextBlock8 As New TextBlock
myTextBlock8.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock8.Text = "ColumnDefinition.Width = ""*"""
Dim myTextBlock9 As New TextBlock
myTextBlock9.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock9.Text = "StackPanel.HorizontalAlignment = ""Stretch"""
Dim myTextBlock10 As New TextBlock
myTextBlock10.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock10.Text = "StackPanel.VerticalAlignment = ""Top"""
Dim myTextBlock11 As New TextBlock
myTextBlock11.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock11.Text = "StackPanel.Orientation = ""Horizontal"""
Dim myTextBlock12 As New TextBlock
myTextBlock12.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock12.Text = "Button.Margin = ""10,0,10,0"""
myStackPanel2.Children.Add(myTextBlock7)
myStackPanel2.Children.Add(myButton4)
myStackPanel2.Children.Add(myButton5)
myStackPanel2.Children.Add(myButton6)
myStackPanel2.Children.Add(myTextBlock8)
myStackPanel2.Children.Add(myTextBlock9)
myStackPanel2.Children.Add(myTextBlock10)
myStackPanel2.Children.Add(myTextBlock11)
myStackPanel2.Children.Add(myTextBlock12)

'Add the final child StackPanel.
Dim myStackPanel3 As New StackPanel
myStackPanel3.HorizontalAlignment = System.Windows.HorizontalAlignment.Left
myStackPanel3.VerticalAlignment = System.Windows.VerticalAlignment.Top
Grid.SetColumn(myStackPanel3, 2)
Grid.SetRow(myStackPanel3, 0)
Dim myTextBlock13 As New TextBlock
myTextBlock13.FontSize = 18
myTextBlock13.HorizontalAlignment = System.Windows.HorizontalAlignment.Center
myTextBlock13.Margin = New Thickness(0, 0, 0, 15)
myTextBlock13.Text = "StackPanel 3"

Dim myButton7 As New Button
myButton7.Margin = New Thickness(10)
myButton7.Content = "Button 7"
Dim myButton8 As New Button
myButton8.Margin = New Thickness(10)
myButton8.Content = "Button 8"
Dim myButton9 As New Button
myButton9.Margin = New Thickness(10)
myButton9.Content = "Button 9"
Dim myTextBlock14 As New TextBlock
myTextBlock14.Text = "ColumnDefinition.Width = ""Auto"""
Dim myTextBlock15 As New TextBlock
myTextBlock15.Text = "StackPanel.HorizontalAlignment = ""Left"""
Dim myTextBlock16 As New TextBlock

```

```
myTextBlock16.Text = "StackPanel.VerticalAlignment = ""Top"""
Dim myTextBlock17 As New TextBlock
myTextBlock17.Text = "StackPanel.Orientation = ""Vertical"""
Dim myTextBlock18 As New TextBlock
myTextBlock18.Text = "Button.Margin = ""10"""
myStackPanel3.Children.Add(myTextBlock13)
myStackPanel3.Children.Add(myButton7)
myStackPanel3.Children.Add(myButton8)
myStackPanel3.Children.Add(myButton9)
myStackPanel3.Children.Add(myTextBlock14)
myStackPanel3.Children.Add(myTextBlock15)
myStackPanel3.Children.Add(myTextBlock16)
myStackPanel3.Children.Add(myTextBlock17)
myStackPanel3.Children.Add(myTextBlock18)

'Add child content to the parent Grid.
myGrid.Children.Add(myStackPanel1)
myGrid.Children.Add(myStackPanel2)
myGrid.Children.Add(myStackPanel3)

'Add the Grid as the lone child of the Border.
myBorder.Child = myGrid
```

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" WindowTitle="Margins, Padding and
Alignment Sample">
    <Border Background="LightBlue"
        BorderBrush="Black"
        BorderThickness="2"
        CornerRadius="45"
        Padding="25">
        <Grid Background="White" ShowGridLines="True">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>

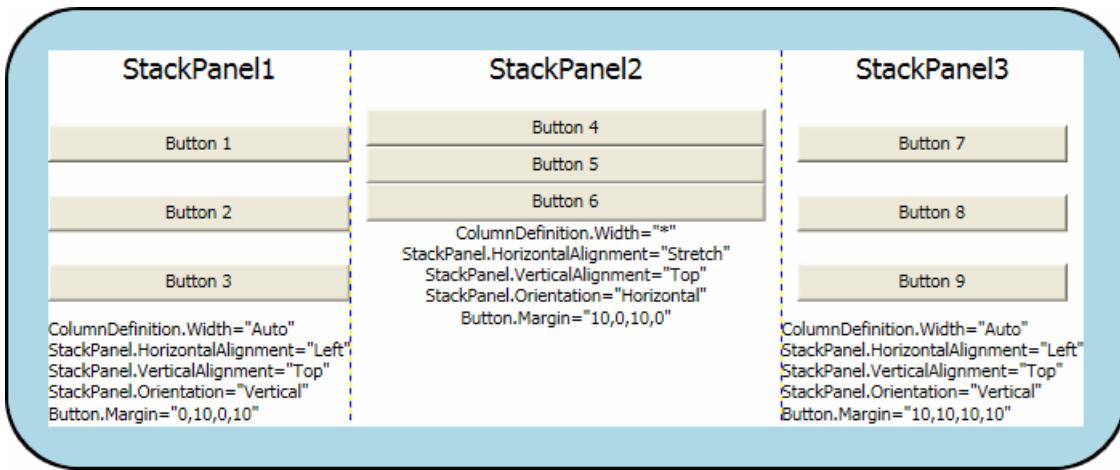
            <StackPanel Grid.Column="0" Grid.Row="0" HorizontalAlignment="Left" Name="StackPanel1"
VerticalAlignment="Top">
                <TextBlock FontSize="18" HorizontalAlignment="Center" Margin="0,0,0,15">StackPanel1</TextBlock>
                <Button Margin="0,10,0,10">Button 1</Button>
                <Button Margin="0,10,0,10">Button 2</Button>
                <Button Margin="0,10,0,10">Button 3</Button>
                <TextBlock>ColumnDefinition.Width="Auto"</TextBlock>
                <TextBlock>StackPanel.HorizontalAlignment="Left"</TextBlock>
                <TextBlock>StackPanel.VerticalAlignment="Top"</TextBlock>
                <TextBlock>StackPanel.Orientation="Vertical"</TextBlock>
                <TextBlock>Button.Margin="0,10,0,10"</TextBlock>
            </StackPanel>

            <StackPanel Grid.Column="1" Grid.Row="0" HorizontalAlignment="Stretch" Name="StackPanel2"
VerticalAlignment="Top" Orientation="Vertical">
                <TextBlock FontSize="18" HorizontalAlignment="Center" Margin="0,0,0,15">StackPanel2</TextBlock>
                <Button Margin="10,0,10,0">Button 4</Button>
                <Button Margin="10,0,10,0">Button 5</Button>
                <Button Margin="10,0,10,0">Button 6</Button>
                <TextBlock HorizontalAlignment="Center">ColumnDefinition.Width="* "</TextBlock>
                <TextBlock HorizontalAlignment="Center">StackPanel.HorizontalAlignment="Stretch"</TextBlock>
                <TextBlock HorizontalAlignment="Center">StackPanel.VerticalAlignment="Top"</TextBlock>
                <TextBlock HorizontalAlignment="Center">StackPanel.Orientation="Horizontal"</TextBlock>
                <TextBlock HorizontalAlignment="Center">Button.Margin="10,0,10,0"</TextBlock>
            </StackPanel>

            <StackPanel Grid.Column="2" Grid.Row="0" HorizontalAlignment="Left" Name="StackPanel3"
VerticalAlignment="Top">
                <TextBlock FontSize="18" HorizontalAlignment="Center" Margin="0,0,0,15">StackPanel3</TextBlock>
                <Button Margin="10">Button 7</Button>
                <Button Margin="10">Button 8</Button>
                <Button Margin="10">Button 9</Button>
                <TextBlock>ColumnDefinition.Width="Auto"</TextBlock>
                <TextBlock>StackPanel.HorizontalAlignment="Left"</TextBlock>
                <TextBlock>StackPanel.VerticalAlignment="Top"</TextBlock>
                <TextBlock>StackPanel.Orientation="Vertical"</TextBlock>
                <TextBlock>Button.Margin="10"</TextBlock>
            </StackPanel>
        </Grid>
    </Border>
</Page>

```

When compiled, the preceding application yields a UI that looks like the following illustration. The effects of the various property values are evident in the spacing between elements, and significant property values for elements in each column are shown within [TextBlock](#) elements.



## What's Next

Positioning properties defined by the [FrameworkElement](#) class enable fine control of element placement within WPF applications. You now have several techniques you can use to better position elements using WPF.

Additional resources are available that explain WPF layout in greater detail. The [Panels Overview](#) topic contains more detail about the various [Panel](#) elements. The topic [Walkthrough: My first WPF desktop application](#) introduces advanced techniques that use layout elements to position components and bind their actions to data sources.

## See also

- [FrameworkElement](#)
- [HorizontalAlignment](#)
- [VerticalAlignment](#)
- [Margin](#)
- [Panels Overview](#)
- [Layout](#)
- [WPF Layout Gallery Sample](#)

# Base Elements How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to use the four WPF base elements: [UIElement](#), [ContentElement](#), [FrameworkElement](#), and [FrameworkContentElement](#).

## In This Section

- [Make a UIElement Transparent or Semi-Transparent](#)
- [Animate the Size of a FrameworkElement](#)
- [Determine Whether a Freezable Is Frozen](#)
- [Handle a Loaded Event](#)
- [Set Margins of Elements and Controls](#)
- [Make a Freezable Read-Only](#)
- [Obtain a Writable Copy of a Read-Only Freezable](#)
- [Flip a UIElement Horizontally or Vertically](#)
- [Use a ThicknessConverter Object](#)
- [Handle the ContextMenuOpening Event](#)

## Reference

- [UIElement](#)
- [ContentElement](#)
- [FrameworkElement](#)
- [FrameworkContentElement](#)

## Related Sections

- [Base Elements](#)

# How to: Make a UIElement Transparent or Semi-Transparent

3/5/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to make a [UIElement](#) transparent or semi-transparent. To make an element transparent or semi-transparent, you set its [Opacity](#) property. A value of `0.0` makes the element completely transparent, while a value of `1.0` makes the element completely opaque. A value of `0.5` makes the element 50% opaque, and so on. An element's [Opacity](#) is set to `1.0` by default.

## Example

The following example sets the [Opacity](#) of a button to `0.25`, making it and its contents (in this case, the button's text) 25% opaque.

```
<!-- Both the button and its text are made 25% opaque. -->
<Button Opacity="0.25">A Button</Button>
```

```
/*
// Both the button and its text are made 25% opaque.
//
Button myTwentyFivePercentOpaqueButton = new Button();
myTwentyFivePercentOpaqueButton.Opacity = new Double();
myTwentyFivePercentOpaqueButton.Opacity = 0.25;
myTwentyFivePercentOpaqueButton.Content = "A Button";
```

If an element's contents have their own [Opacity](#) settings, those values are multiplied against the containing elements [Opacity](#).

The following example sets a button's [Opacity](#) to `0.25`, and the [Opacity](#) of an [Image](#) control contained within in the button to `0.5`. As a result, the image appears 12.5% opaque:  $0.25 * 0.5 = 0.125$ .

```
<!-- The image contained within this button has an effective
     opacity of 0.125 (0.25 * 0.5 = 0.125). -->
<Button Opacity="0.25">
    <StackPanel Orientation="Horizontal">
        <TextBlock VerticalAlignment="Center" Margin="10">A Button</TextBlock>
        <Image Source="sampleImages\berries.jpg" Width="50" Height="50"
              Opacity="0.5"/>
    </StackPanel>
</Button>
```

```

//  

// The image contained within this button has an  

// effective opacity of 0.125 (0.25*0.5 = 0.125);  

//  

Button myImageButton = new Button();  

myImageButton.Opacity = new Double();  

myImageButton.Opacity = 0.25;  

StackPanel myImageStackPanel = new StackPanel();  

myImageStackPanel.Orientation = Orientation.Horizontal;  

TextBlock myTextBlock = new TextBlock();  

myTextBlock.VerticalAlignment = VerticalAlignment.Center;  

myTextBlock.Margin = new Thickness(10);  

myTextBlock.Text = "A Button";  

myImageStackPanel.Children.Add(myTextBlock);  

Image myImage = new Image();  

BitmapImage myBitmapImage = new BitmapImage();  

myBitmapImage.BeginInit();  

myBitmapImage.UriSource = new Uri("sampleImages/berries.jpg", UriKind.Relative);  

myBitmapImage.EndInit();  

myImage.Source = myBitmapImage;  

ImageBrush myImageBrush = new ImageBrush(myBitmapImage);  

myImage.Width = 50;  

myImage.Height = 50;  

myImage.Opacity = 0.5;  

myImageStackPanel.Children.Add(myImage);  

myImageButton.Content = myImageStackPanel;

```

Another way to control the opacity of an element is to set the opacity of the [Brush](#) that paints the element. This approach enables you to selectively alter the opacity of portions of an element, and provides performance benefits over using the element's [Opacity](#) property. The following example sets the [Opacity](#) of a [SolidColorBrush](#) used to paint the button's [Background](#) is set to `0.25`. As a result, the brush's background is 25% opaque, but its contents (the button's text) remain 100% opaque.

```

<!-- This button's background is made 25% opaque, but its  

    text remains 100% opaque. -->  

<Button>  

    <Button.Background>  

        <SolidColorBrush Color="Gray" Opacity="0.25" />  

    </Button.Background>  

    A Button  

</Button>

```

```

//  

// This button's background is made 25% opaque,  

// but its text remains 100% opaque.  

//  

Button myOpaqueTextButton = new Button();  

SolidColorBrush mySolidColorBrush = new SolidColorBrush(Colors.Gray);  

mySolidColorBrush.Opacity = 0.25;  

myOpaqueTextButton.Background = mySolidColorBrush;  

myOpaqueTextButton.Content = "A Button";

```

You may also control the opacity of individual colors within a brush. For more information about colors and brushes, see [Painting with Solid Colors and Gradients Overview](#). For an example showing how to animate an element's opacity, see [Animate the Opacity of an Element or Brush](#).

# How to: Animate the Size of a FrameworkElement

3/5/2019 • 2 minutes to read • [Edit Online](#)

To animate the size of a [FrameworkElement](#), you can either animate its [Width](#) and [Height](#) properties or use an animated [ScaleTransform](#).

In the following example animates the size of two buttons using these two approaches. One button is resized by animating its [Width](#) property and another is resized by animating a [ScaleTransform](#) applied to its [RenderTransform](#) property. Each button contains some text. Initially, the text appears the same in both buttons, but as the buttons are resized, the text in the second button becomes distorted.

## Example

```

<!-- AnimatingSizeExample.xaml
This example shows two ways of animating the size
of a framework element. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Microsoft.Samples.Animation.AnimatingSizeExample"
    WindowTitle="Animating Size Example">
    <Canvas Width="650" Height="400">

        <Button Name="AnimatedWidthButton"
            Canvas.Left="20" Canvas.Top="20"
            Width="200" Height="150"
            BorderBrush="Red" BorderThickness="5">
            Click Me
            <Button.Triggers>

                <!-- Animate the button's Width property. -->
                <EventTrigger RoutedEvent="Button.Loaded">
                    <BeginStoryboard>
                        <Storyboard>
                            <DoubleAnimation
                                Storyboard.TargetName="AnimatedWidthButton"
                                Storyboard.TargetProperty="(Button.Width)"
                                To="500" Duration="0:0:10" AutoReverse="True"
                                RepeatBehavior="Forever" />
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Button.Triggers>
        </Button>

        <Button
            Canvas.Left="20" Canvas.Top="200"
            Width="200" Height="150"
            BorderBrush="Black" BorderThickness="3">
            Click Me
            <Button.RenderTransform>
                <ScaleTransform x:Name="MyAnimatedScaleTransform"
                    ScaleX="1" ScaleY="1" />
            </Button.RenderTransform>
            <Button.Triggers>

                <!-- Animate the ScaleX property of a ScaleTransform
                    applied to the button. -->
                <EventTrigger RoutedEvent="Button.Loaded">
                    <BeginStoryboard>
                        <Storyboard>
                            <DoubleAnimation
                                Storyboard.TargetName="MyAnimatedScaleTransform"
                                Storyboard.TargetProperty="(ScaleTransform.ScaleX)"
                                To="3.0" Duration="0:0:10" AutoReverse="True"
                                RepeatBehavior="Forever" />
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Button.Triggers>
        </Button>
    </Canvas>
</Page>

```

When you transform an element, the entire element and its contents are transformed. When you directly alter the size of an element, as in the case of the first button, the element's contents are not resized unless their size and position depend on the size of their parent element.

Animating the size of an element by applying an animated transform to its [RenderTransform](#) property provides

better performance than animated its [Width](#) and [Height](#) directly, because the [RenderTransform](#) property does not trigger a layout pass.

For more information about animating properties, see the [Animation Overview](#). For more information about transforms, see the [Transforms Overview](#).

# How to: Determine Whether a Freezable Is Frozen

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to determine whether a [Freezable](#) object is frozen. If you try to modify a frozen [Freezable](#) object, it throws an [InvalidOperationException](#). To avoid throwing this exception, use the [IsFrozen](#) property of the [Freezable](#) object to determine whether it is frozen.

## Example

The following example freezes a [SolidColorBrush](#) and then tests it by using the [IsFrozen](#) property to determine whether it is frozen.

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

if (myBrush.IsFrozen) // Evaluates to true.
{
    // If the brush is frozen, create a clone and
    // modify the clone.
    SolidColorBrush myBrushClone = myBrush.Clone();
    myBrushClone.Color = Colors.Red;
    myButton.Background = myBrushClone;
}
else
{
    // If the brush is not frozen,
    // it can be modified directly.
    myBrush.Color = Colors.Red;
}
```

```
Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)

If myBrush.CanFreeze Then
    ' Makes the brush unmodifiable.
    myBrush.Freeze()
End If

myButton.Background = myBrush

If myBrush.IsFrozen Then ' Evaluates to true.
    ' If the brush is frozen, create a clone and
    ' modify the clone.
    Dim myBrushClone As SolidColorBrush = myBrush.Clone()
    myBrushClone.Color = Colors.Red
    myButton.Background = myBrushClone
Else
    ' If the brush is not frozen,
    ' it can be modified directly.
    myBrush.Color = Colors.Red
End If
```

For more information about [Freezable](#) objects, see the [Freezable Objects Overview](#).

## See also

- [Freezable](#)
- [IsFrozen](#)
- [Freezable Objects Overview](#)
- [How-to Topics](#)

# How to: Handle a Loaded Event

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to handle the [FrameworkElement.Loaded](#) event, and an appropriate scenario for handling that event. The handler creates a [Button](#) when the page loads.

## Example

The following example uses Extensible Application Markup Language (XAML) together with a code-behind file.

```
<StackPanel  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    x:Class="SDKSample.FELoaded"  
    Loaded="OnLoad"  
    Name="root"  
>  
</StackPanel>
```

```
void OnLoad(object sender, RoutedEventArgs e)  
{  
    Button b1 = new Button();  
    b1.Content = "New Button";  
    root.Children.Add(b1);  
    b1.Height = 25;  
    b1.Width = 200;  
    b1.HorizontalAlignment = HorizontalAlignment.Left;  
}
```

```
Private Sub OnLoad(ByVal sender As Object, ByVal e As RoutedEventArgs)  
    Dim b1 As Button = New Button()  
    b1.Content = "New Button"  
    root.Children.Add(b1)  
    b1.Height = 25  
    b1.Width = 200  
    b1.HorizontalAlignment = HorizontalAlignment.Left  
End Sub
```

## See also

- [FrameworkElement](#)
- [Object Lifetime Events](#)
- [Routed Events Overview](#)
- [How-to Topics](#)

# How to: Set Margins of Elements and Controls

3/5/2019 • 2 minutes to read • [Edit Online](#)

This example describes how to set the [Margin](#) property, by changing any existing property value for the margin in code-behind. The [Margin](#) property is a property of the [FrameworkElement](#) base element, and is thus inherited by a variety of controls and other elements.

This example is written in Extensible Application Markup Language (XAML), with a code-behind file that the XAML refers to. The code-behind is shown in both a C# and a Microsoft Visual Basic version.

## Example

```
<Button Click="OnClick" Margin="10" Name="btn1">  
Click To See Change!!</Button>
```

```
void OnClick(object sender, RoutedEventArgs e)  
{  
    // Get the current value of the property.  
    Thickness marginThickness = btn1.Margin;  
    // If the current leftlength value of margin is set to 10 then change it to a new value.  
    // Otherwise change it back to 10.  
    if(marginThickness.Left == 10)  
    {  
        btn1.Margin = new Thickness(60);  
    } else {  
        btn1.Margin = new Thickness(10);  
    }  
}
```

```
Private Sub OnClick(ByVal sender As Object, ByVal e As RoutedEventArgs)  
  
    ' Get the current value of the property.  
    Dim marginThickness As Thickness  
    marginThickness = btn1.Margin  
    ' If the current leftlength value of margin is set to 10 then change it to a new value.  
    ' Otherwise change it back to 10.  
    If marginThickness.Left = 10 Then  
        btn1.Margin = New Thickness(60)  
    Else  
        btn1.Margin = New Thickness(10)  
    End If  
End Sub
```

# How to: Make a Freezable Read-Only

11/3/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to make a [Freezable](#) read-only by calling its [Freeze](#) method.

You cannot freeze a [Freezable](#) object if any one of the following conditions is `true` about the object:

- It has animated or data bound properties.
- It has properties that are set by a dynamic resource. For more information about dynamic resources, see the [XAML Resources](#).
- It contains [Freezable](#) sub-objects that cannot be frozen.

If these conditions are `false` for your [Freezable](#) object and you do not intend to modify it, consider freezing it to gain performance benefits.

## Example

The following example freezes a [SolidColorBrush](#), which is a type of [Freezable](#) object.

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;
```

```
Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)

If myBrush.CanFreeze Then
    ' Makes the brush unmodifiable.
    myBrush.Freeze()
End If

myButton.Background = myBrush
```

For more information about [Freezable](#) objects, see the [Freezable Objects Overview](#).

## See also

- [Freezable](#)
- [CanFreeze](#)
- [Freeze](#)
- [Freezable Objects Overview](#)
- [How-to Topics](#)

# How to: Obtain a Writable Copy of a Read-Only Freezable

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use the [Clone](#) method to create a writable copy of a read-only [Freezable](#).

After a [Freezable](#) object is marked as read-only ("frozen"), you cannot modify it. However, you can use the [Clone](#) method to create a modifiable clone of the frozen object.

## Example

The following example creates a modifiable clone of a frozen [SolidColorBrush](#) object.

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

// Freezing a Freezable before it provides
// performance improvements if you don't
// intend on modifying it.
if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

// If you need to modify a frozen brush,
// the Clone method can be used to
// create a modifiable copy.
SolidColorBrush myBrushClone = myBrush.Clone();

// Changing myBrushClone does not change
// the color of myButton, because its
// background is still set by myBrush.
myBrushClone.Color = Colors.Red;

// Replacing myBrush with myBrushClone
// makes the button change to red.
myButton.Background = myBrushClone;
```

```
Dim myButton As New Button()
Dim myBrush As New SolidColorBrush(Colors.Yellow)

' Freezing a Freezable before it provides
' performance improvements if you don't
' intend on modifying it.
If myBrush.CanFreeze Then
    ' Makes the brush unmodifiable.
    myBrush.Freeze()
End If

myButton.Background = myBrush

' If you need to modify a frozen brush,
' the Clone method can be used to
' create a modifiable copy.
Dim myBrushClone As SolidColorBrush = myBrush.Clone()

' Changing myBrushClone does not change
' the color of myButton, because its
' background is still set by myBrush.
myBrushClone.Color = Colors.Red

' Replacing myBrush with myBrushClone
' makes the button change to red.
myButton.Background = myBrushClone
```

For more information about [Freezable](#) objects, see the [Freezable Objects Overview](#).

## See also

- [Freezable](#)
- [CloneCurrentValue](#)
- [Freezable Objects Overview](#)
- [How-to Topics](#)

# How to: Flip a UIElement Horizontally or Vertically

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use a [ScaleTransform](#) to flip a [UIElement](#) horizontally or vertically. In this example, a [Button](#) control (a type of [UIElement](#)) is flipped by applying a [ScaleTransform](#) to its [RenderTransform](#) property.

## Example

The following illustration shows the button to flip.



The UIElement to flip

The following shows the code that creates the button.

```
<Button Content="Flip me!" Padding="5">  
</Button>
```

## Example

To flip the button horizontally, create a [ScaleTransform](#) and set its [ScaleX](#) property to -1. Apply the [ScaleTransform](#) to the button's [RenderTransform](#) property.

```
<Button Content="Flip me!" Padding="5">  
  <Button.RenderTransform>  
    <ScaleTransform ScaleX="-1" />  
  </Button.RenderTransform>  
</Button>
```



The button after applying the ScaleTransform

## Example

As you can see from the previous illustration, the button was flipped, but it was also moved. That's because the button was flipped from its top left corner. To flip the button in place, you want to apply the [ScaleTransform](#) to its center, not its corner. An easy way to apply the [ScaleTransform](#) to the button's center is to set the button's [RenderTransformOrigin](#) property to 0.5, 0.5.

```
<Button Content="Flip me!" Padding="5"
    RenderTransformOrigin="0.5,0.5">
    <Button.RenderTransform>
        <ScaleTransform ScaleX="-1" />
    </Button.RenderTransform>
</Button>
```



The button with a RenderTransformOrigin of 0.5, 0.5

## Example

To flip the button vertically, set the [ScaleTransform](#) object's [ScaleY](#) property instead of its [ScaleX](#) property.

```
<Button Content="Flip me!" Padding="5"
    RenderTransformOrigin="0.5,0.5">
    <Button.RenderTransform>
        <ScaleTransform ScaleY="-1" />
    </Button.RenderTransform>
</Button>
```



The vertically flipped button

## See also

- [Transforms Overview](#)

# How to: Use a ThicknessConverter Object

4/8/2019 • 2 minutes to read • [Edit Online](#)

## Example

This example shows how to create an instance of [ThicknessConverter](#) and use it to change the thickness of a border.

The example defines a custom method called `changeThickness`; this method first converts the contents of a [ListBoxItem](#), as defined in a separate Extensible Application Markup Language (XAML) file, to an instance of [Thickness](#), and later converts the content into a [String](#). This method passes the [ListBoxItem](#) to a [ThicknessConverter](#) object, which converts the [Content](#) of a [ListBoxItem](#) to an instance of [Thickness](#). This value is then passed back as the value of the [BorderThickness](#) property of the [Border](#).

This example does not run.

```
private void changeThickness(object sender, SelectionChangedEventArgs args)
{
    ListBoxItem li = ((sender as ListBox).SelectedItem as ListBoxItem);
    ThicknessConverter myThicknessConverter = new ThicknessConverter();
    Thickness th1 = (Thickness)myThicknessConverter.ConvertFromString(li.Content.ToString());
    border1.BorderThickness = th1;
    bThickness.Text = "Border.BorderThickness =" + li.Content.ToString();
}
```

```
Private Sub changeThickness(ByVal sender As Object, ByVal args As SelectionChangedEventArgs)

    Dim li As ListBoxItem = CType(CType(sender, ListBox).SelectedItem, ListBoxItem)
    Dim myThicknessConverter As System.Windows.ThicknessConverter = New System.Windows.ThicknessConverter()
    Dim th1 As Thickness = CType(myThicknessConverter.ConvertFromString(li.Content.ToString()), Thickness)
    border1.BorderThickness = th1
    bThickness.Text = "Border.BorderThickness =" + li.Content.ToString()
End Sub
```

## See also

- [Thickness](#)
- [ThicknessConverter](#)
- [Border](#)
- [How to: Change the Margin Property](#)
- [How to: Convert a ListBoxItem to a new Data Type](#)
- [Panels Overview](#)

# How to: Handle the ContextMenuOpening Event

4/28/2019 • 5 minutes to read • [Edit Online](#)

The [ContextMenuOpening](#) event can be handled in an application to either adjust an existing context menu prior to display or to suppress the menu that would otherwise be displayed by setting the [Handled](#) property to `true` in the event data. The typical reason for setting [Handled](#) to `true` in the event data is to replace the menu entirely with a new [ContextMenu](#) object, which sometimes requires canceling the operation and starting a new open. If you write handlers for the [ContextMenuOpening](#) event, you should be aware of timing issues between a [ContextMenu](#) control and the service that is responsible for opening and positioning context menus for controls in general. This topic illustrates some of the code techniques for various context menu opening scenarios and illustrates a case where the timing issue comes into play.

There are several scenarios for handling the [ContextMenuOpening](#) event:

- Adjusting the menu items before display.
- Replacing the entire menu before display.
- Completely suppressing any existing context menu and displaying no context menu.

## Example

### Adjusting the Menu Items Before Display

Adjusting the existing menu items is fairly simple and is probably the most common scenario. You might do this in order to add or subtract context menu options in response to current state information in your application or particular state information that is available as a property on the object where the context menu is requested.

The general technique is to get the source of the event, which is the specific control that was right-clicked, and get the [ContextMenu](#) property from it. You typically want to check the [Items](#) collection to see what context menu items already exist in the menu, and then add or remove appropriate new [MenuItem](#) items to or from the collection.

```
void AddItemToCM(object sender, ContextMenuEventArgs e)
{
    //check if Item4 is already there, this will probably run more than once
    FrameworkElement fe = e.Source as FrameworkElement;
    ContextMenu cm = fe.ContextMenu;
    foreach (MenuItem mi in cm.Items)
    {
        if ((String)mi.Header == "Item4") return;
    }
    MenuItem mi4 = new MenuItem();
    mi4.Header = "Item4";
    fe.ContextMenu.Items.Add(mi4);
}
```

### Replacing the Entire Menu Before Display

An alternative scenario is if you want to replace the entire context menu. You could of course also use a variation of the preceding code, to remove every item of an existing context menu and add new ones starting with item zero. But the more intuitive approach for replacing all items in the context menu is to create a new [ContextMenu](#), populate it with items, and then set the [FrameworkElement.ContextMenu](#) property of a control to be the new [ContextMenu](#).

The following is the simple handler code for replacing a [ContextMenu](#). The code references a custom [BuildMenu](#) method, which is separated out because it is called by more than one of the example handlers.

```
void HandlerForCMO(object sender, ContextMenuEventArgs e)
{
    FrameworkElement fe = e.Source as FrameworkElement;
    fe.ContextMenu = BuildMenu();
}
```

```
ContextMenu BuildMenu()
{
    ContextMenu theMenu = new ContextMenu();
    MenuItem mia = new MenuItem();
    mia.Header = "Item1";
    MenuItem mib = new MenuItem();
    mib.Header = "Item2";
    MenuItem mic = new MenuItem();
    mic.Header = "Item3";
    theMenu.Items.Add(mia);
    theMenu.Items.Add(mib);
    theMenu.Items.Add(mic);
    return theMenu;
}
```

However, if you use this style of handler for [ContextMenuOpening](#), you can potentially expose a timing issue if the object where you are setting the [ContextMenu](#) does not have a preexisting context menu. When a user right-clicks a control, [ContextMenuOpening](#) is raised even if the existing [ContextMenu](#) is empty or null. But in this case, whatever new [ContextMenu](#) you set on the source element arrives too late to be displayed. Also, if the user happens to right-click a second time, this time your new [ContextMenu](#) appears, the value is non null, and your handler will properly replace and display the menu when the handler runs a second time. This suggests two possible workarounds:

1. Insure that [ContextMenuOpening](#) handlers always run against controls that have at least a placeholder [ContextMenu](#) available, which you intend to be replaced by the handler code. In this case, you can still use the handler shown in the previous example, but you typically want to assign a placeholder [ContextMenu](#) in the initial markup:

```
<StackPanel>
    <Rectangle Fill="Yellow" Width="200" Height="100" ContextMenuOpening="HandlerForCMO">
        <Rectangle.ContextMenu>
            <ContextMenu>
                <MenuItem>Initial menu; this will be replaced ...</MenuItem>
            </ContextMenu>
        </Rectangle.ContextMenu>
    </Rectangle>
    <TextBlock>Right-click the rectangle above, context menu gets replaced</TextBlock>
</StackPanel>
```

2. Assume that the initial [ContextMenu](#) value might be null, based on some preliminary logic. You could either check [ContextMenu](#) for null, or use a flag in your code to check whether your handler has been run at least once. Because you assume that the [ContextMenu](#) is about to be displayed, your handler then sets [Handled](#) to [true](#) in the event data. To the [ContextMenuService](#) that is responsible for context menu display, a [true](#) value for [Handled](#) in the event data represents a request to cancel the display for the context menu / control combination that raised the event.

Now that you have suppressed the potentially suspect context menu, the next step is to supply a new one, then display it. Setting the new one is basically the same as the previous handler: you build a new [ContextMenu](#) and set

the control source's [FrameworkElement.ContextMenu](#) property with it. The additional step is that you must now force the display of the context menu, because you suppressed the first attempt. To force the display, you set the [Popup.IsOpen](#) property to `true` within the handler. Be careful when you do this, because opening the context menu in the handler raises the [ContextMenuOpening](#) event again. If you reenter your handler, it becomes infinitely recursive. This is why you always need to check for `null` or use a flag if you open a context menu from within a [ContextMenuOpening](#) event handler.

## Suppressing Any Existing Context Menu and Displaying No Context Menu

The final scenario, writing a handler that suppresses a menu totally, is uncommon. If a given control is not supposed to display a context menu, there are probably more appropriate ways to assure this than by suppressing the menu just when a user requests it. But if you want to use the handler to suppress a context menu and show nothing, then your handler should simply set [Handled](#) to `true` in the event data. The [ContextMenuService](#) that is responsible for displaying a context menu will check the event data of the event it raised on the control. If the event was marked [Handled](#) anywhere along the route, then the context menu open action that initiated the event is suppressed.

```
void HandlerForCM02(object sender, ContextMenuEventArgs e)
{
    if (!FlagForCustomContextMenu)
    {
        e.Handled = true; //need to suppress empty menu
        FrameworkElement fe = e.Source as FrameworkElement;
        fe.ContextMenu = BuildMenu();
        FlagForCustomContextMenu = true;
        fe.ContextMenu.IsOpen = true;
    }
}
```

## See also

- [ContextMenu](#)
- [FrameworkElement.ContextMenu](#)
- [Base Elements Overview](#)
- [ContextMenu Overview](#)

# Element Tree and Serialization

11/3/2019 • 2 minutes to read • [Edit Online](#)

WPF programming elements often exist in some form of tree relationship to each other. For instance, an application UI created in XAML can be conceptualized as an object tree. The element tree can be further divided into two discrete yet sometimes parallel trees: the logical tree and the visual tree. Serialization in WPF involves saving the state of these two trees as well as application state and writing it to a file, potentially as XAML.

## In This Section

[Trees in WPF](#)

[Serialization Limitations of XamlWriter.Save](#)

[Initialization for Object Elements Not in an Object Tree](#)

[How-to Topics](#)

## Reference

[System.Windows.Markup](#)

[LogicalTreeHelper](#)

[VisualTreeHelper](#)

## Related Sections

[WPF Architecture](#)

[XAML in WPF](#)

[Base Elements](#)

[Properties](#)

[Events](#)

[Input](#)

[Resources](#)

[Styling and Templating](#)

[Threading Model](#)

# Trees in WPF

11/3/2019 • 10 minutes to read • [Edit Online](#)

In many technologies, elements and components are organized in a tree structure where developers directly manipulate the object nodes in the tree to affect the rendering or behavior of an application. Windows Presentation Foundation (WPF) also uses several tree structure metaphors to define relationships between program elements. For the most part WPF developers can create an application in code or define portions of the application in XAML while thinking conceptually about the object tree metaphor, but will be calling specific API or using specific markup to do so rather than some general object tree manipulation API such as you might use in XML DOM. WPF exposes two helper classes that provide a tree metaphor view, [LogicalTreeHelper](#) and [VisualTreeHelper](#). The terms visual tree and logical tree are also used in the WPF documentation because these same trees are useful for understanding the behavior of certain key WPF features. This topic defines what the visual tree and logical tree represent, discusses how such trees relate to an overall object tree concept, and introduces [LogicalTreeHelper](#) and [VisualTreeHelpers](#).

## Trees in WPF

The most complete tree structure in WPF is the object tree. If you define an application page in XAML and then load the XAML, the tree structure is created based on the nesting relationships of the elements in the markup. If you define an application or a portion of the application in code, then the tree structure is created based on how you assign property values for properties that implement the content model for a given object. In Windows Presentation Foundation (WPF), there are two ways that the complete object tree is conceptualized and can be reported to its public API: as the logical tree and as the visual tree. The distinctions between logical tree and visual tree are not always necessarily important, but they can occasionally cause issues with certain WPF subsystems and affect choices you make in markup or code.

Even though you do not always manipulate either the logical tree or the visual tree directly, understanding the concepts of how the trees interact is useful for understanding WPF as a technology. Thinking of WPF as a tree metaphor of some kind is also crucial to understanding how property inheritance and event routing work in WPF.

### NOTE

Because the object tree is more of a concept than an actual API, another way to think of the concept is as an object graph. In practice, there are relationships between objects at run time where the tree metaphor will break down. Nevertheless, particularly with XAML-defined UI, the tree metaphor is relevant enough that most WPF documentation will use the term object tree when referencing this general concept.

## The Logical Tree

In WPF, you add content to UI elements by setting properties of the objects that back those elements. For example, you add items to a [ListBox](#) control by manipulating its [Items](#) property. By doing this, you are placing items into the [ItemCollection](#) that is the [Items](#) property value. Similarly, to add objects to a [DockPanel](#), you manipulate its [Children](#) property value. Here, you are adding objects to the [UIElementCollection](#). For a code example, see [How to: Add an Element Dynamically](#).

In Extensible Application Markup Language (XAML), when you place list items in a [ListBox](#) or controls or other UI elements in a [DockPanel](#), you also use the [Items](#) and [Children](#) properties, either explicitly or implicitly, as in the following example.

```

<DockPanel
    Name="ParentElement"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <!--implicit: <DockPanel.Children>-->
    <ListBox DockPanel.Dock="Top">
        <!--implicit: <ListBox.Items>-->
        <ListBoxItem>
            <TextBlock>Dog</TextBlock>
        </ListBoxItem>
        <ListBoxItem>
            <TextBlock>Cat</TextBlock>
        </ListBoxItem>
        <ListBoxItem>
            <TextBlock>Fish</TextBlock>
        </ListBoxItem>
    <!--implicit: </ListBox.Items>-->
    </ListBox>
    <Button Height="20" Width="100" DockPanel.Dock="Top">Buy a Pet</Button>
    <!--implicit: </DockPanel.Children>-->
</DockPanel>

```

If you were to process this XAML as XML under a document object model, and if you had included the tags commented out as implicit (which would have been legal), then the resulting XML DOM tree would have included elements for `<ListBox.Items>` and the other implicit items. But XAML does not process that way when you read the markup and write to objects, the resulting object graph does not literally include `ListBox.Items`. It does however have a `ListBox` property named `Items` that contains a `ItemCollection`, and that `ItemCollection` is initialized but empty when the `ListBox` XAML is processed. Then, each child object element that exists as content for the `ListBox` is added to the `ItemCollection` by parser calls to `ItemCollection.Add`. This example of processing XAML into an object tree is so far seemingly an example where the created object tree is basically the logical tree.

However, the logical tree is not the entire object graph that exists for your application UI at run time, even with the XAML implicit syntax items factored out. The main reason for this is visuals and templates. For example, consider the `Button`. The logical tree reports the `Button` object and also its string `Content`. But there is more to this button in the run-time object tree. In particular, the button only appears on screen the way it does because a specific `Button` control template was applied. The visuals that come from an applied template (such as the template-defined `Border` of dark gray around the visual button) are not reported in the logical tree, even if you are looking at the logical tree during run time (such as handling an input event from the visible UI and then reading the logical tree). To find the template visuals, you would instead need to examine the visual tree.

For more information about how XAML syntax maps to the created object graph, and implicit syntax in XAML, see [XAML Syntax In Detail](#) or [XAML Overview \(WPF\)](#).

## The Purpose of the Logical Tree

The logical tree exists so that content models can readily iterate over their possible child objects, and so that content models can be extensible. Also, the logical tree provides a framework for certain notifications, such as when all objects in the logical tree are loaded. Basically, the logical tree is an approximation of a run time object graph at the framework level, which excludes visuals, but is adequate for many querying operations against your own run time application's composition.

In addition, both static and dynamic resource references are resolved by looking upwards through the logical tree for `Resources` collections on the initial requesting object, and then continuing up the logical tree and checking each `FrameworkElement` (or `FrameworkContentElement`) for another `Resources` value that contains a `ResourceDictionary`, possibly containing that key. The logical tree is used for resource lookup when both the logical tree and the visual tree are present. For more information on resource dictionaries and lookup, see [XAML Resources](#).

## Composition of the Logical Tree

The logical tree is defined at the WPF framework-level, which means that the WPF base element that is most relevant for logical tree operations is either [FrameworkElement](#) or [FrameworkContentElement](#). However, as you can see if you actually use the [LogicalTreeHelper](#) API, the logical tree sometimes contains nodes that are not either [FrameworkElement](#) or [FrameworkContentElement](#). For instance, the logical tree reports the [Text](#) value of a [TextBlock](#), which is a string.

## Overriding the Logical Tree

Advanced control authors can override the logical tree by overriding several APIs that define how a general object or content model adds or removes objects within the logical tree. For an example of how to override the logical tree, see [Override the Logical Tree](#).

## Property Value Inheritance

Property value inheritance operates through a hybrid tree. The actual metadata that contains the [Inherits](#) property that enables property inheritance is the WPF framework-level [FrameworkPropertyMetadata](#) class. Therefore, both the parent that holds the original value and the child object that inherits that value must both be [FrameworkElement](#) or [FrameworkContentElement](#), and they must both be part of some logical tree. However, for existing WPF properties that support property inheritance, property value inheritance is able to perpetuate through an intervening object that is not in the logical tree. Mainly this is relevant for having template elements use any inherited property values set either on the instance that is templated, or at still higher levels of page-level composition and therefore higher in the logical tree. In order for property value inheritance to work consistently across such a boundary, the inheriting property must be registered as an attached property, and you should follow this pattern if you intend to define a custom dependency property with property inheritance behavior. The exact tree used for property inheritance cannot be entirely anticipated by a helper class utility method, even at run time. For more information, see [Property Value Inheritance](#).

## The Visual Tree

In addition to the concept of the logical tree, there is also the concept of the visual tree in WPF. The visual tree describes the structure of visual objects, as represented by the [Visual](#) base class. When you write a template for a control, you are defining or redefining the visual tree that applies for that control. The visual tree is also of interest to developers who want lower-level control over drawing for performance and optimization reasons. One exposure of the visual tree as part of conventional WPF application programming is that event routes for a routed event mostly travel along the visual tree, not the logical tree. This subtlety of routed event behavior might not be immediately apparent unless you are a control author. Routing events through the visual tree enables controls that implement composition at the visual level to handle events or create event setters.

## Trees, Content Elements, and Content Hosts

Content elements (classes that derive from [ContentElement](#)) are not part of the visual tree; they do not inherit from [Visual](#) and do not have a visual representation. In order to appear in a UI at all, a [ContentElement](#) must be hosted in a content host that is both a [Visual](#) and a logical tree participant. Usually such an object is a [FrameworkElement](#). You can conceptualize that the content host is somewhat like a "browser" for the content and chooses how to display that content within the screen region that the host controls. When the content is hosted, the content can be made a participant in certain tree processes that are normally associated with the visual tree. Generally, the [FrameworkElement](#) host class includes implementation code that adds any hosted [ContentElement](#) to the event route through subnodes of the content logical tree, even though the hosted content is not part of the true visual tree. This is necessary so that a [ContentElement](#) can source a routed event that routes to any element other than itself.

## Tree Traversal

The [LogicalTreeHelper](#) class provides the [GetChildren](#), [GetParent](#), and [FindLogicalNode](#) methods for logical tree

traversal. In most cases, you should not have to traverse the logical tree of existing controls, because these controls almost always expose their logical child elements as a dedicated collection property that supports collection access such as [Add](#), an indexer, and so on. Tree traversal is mainly a scenario that is used by control authors who choose not to derive from intended control patterns such as [ItemsControl](#) or [Panel](#) where collection properties are already defined, and who intend to provide their own collection property support.

The visual tree also supports a helper class for visual tree traversal, [VisualTreeHelper](#). The visual tree is not exposed as conveniently through control-specific properties, so the [VisualTreeHelper](#) class is the recommended way to traverse the visual tree if that is necessary for your programming scenario. For more information, see [WPF Graphics Rendering Overview](#).

#### NOTE

Sometimes it is necessary to examine the visual tree of an applied template. You should be careful when using this technique. Even if you are traversing a visual tree for a control where you define the template, consumers of your control can always change the template by setting the [Template](#) property on instances, and even the end user can influence the applied template by changing the system theme.

## Routes for Routed Events as a "Tree"

As mentioned before, the route of any given routed event travels along a single and predetermined path of a tree that is a hybrid of the visual and logical tree representations. The event route can travel either in the up or down directions within the tree depending on whether it is a tunneling or bubbling routed event. The event route concept does not have a directly supporting helper class that could be used to "walk" the event route independently of raising an event that actually routes. There is a class that represents the route, [EventRoute](#), but the methods of that class are generally for internal use only.

## Resource Dictionaries and Trees

Resource dictionary lookup for all [Resources](#) defined in a page traverses basically the logical tree. Objects that are not in the logical tree can reference keyed resources, but the resource lookup sequence begins at the point where that object is connected to the logical tree. In WPF, only logical tree nodes can have a [Resources](#) property that contains a [ResourceDictionary](#), therefore there is no benefit in traversing the visual tree looking for keyed resources from a [ResourceDictionary](#).

However, resource lookup can also extend beyond the immediate logical tree. For application markup, the resource lookup can then continue onward to application-level resource dictionaries and then to theme support and system values that are referenced as static properties or keys. Themes themselves can also reference system values outside of the theme logical tree if the resource references are dynamic. For more information on resource dictionaries and the lookup logic, see [XAML Resources](#).

## See also

- [Input Overview](#)
- [WPF Graphics Rendering Overview](#)
- [Routed Events Overview](#)
- [Initialization for Object Elements Not in an Object Tree](#)
- [WPF Architecture](#)

# Serialization Limitations of XamlWriter.Save

11/7/2019 • 2 minutes to read • [Edit Online](#)

The API [Save](#) can be used to serialize the contents of a Windows Presentation Foundation (WPF) application as a Extensible Application Markup Language (XAML) file. However, there are some notable limitations in exactly what is serialized. These limitations and some general considerations are documented in this topic.

## Run-Time, Not Design-Time Representation

The basic philosophy of what is serialized by a call to [Save](#) is that the result will be a representation of the object being serialized, at run-time. Many design-time properties of the original XAML file may already be optimized or lost by the time that the XAML is loaded as in-memory objects, and are not preserved when you call [Save](#) to serialize. The serialized result is an effective representation of the constructed logical tree of the application, but not necessarily of the original XAML that produced it. These issues make it extremely difficult to use the [Save](#) serialization as part of an extensive XAML design surface.

## Serialization is Self-Contained

The serialized output of [Save](#) is self-contained; everything that is serialized is contained inside a XAML single page, with a single root element, and no external references other than URLs. For instance, if your page referenced resources from application resources, these will appear as if they were a component of the page being serialized.

## Extension References are Dereferenced

Common references to objects made by various markup extension formats, such as `StaticResource` or `Binding`, will be dereferenced by the serialization process. These were already dereferenced at the time that in-memory objects were created by the application runtime, and the [Save](#) logic does not revisit the original XAML to restore such references to the serialized output. This potentially freezes any databound or resource obtained value to be the value last used by the run-time representation, with only limited or indirect ability to distinguish such a value from any other value set locally. Images are also serialized as object references to images as they exist in the project, rather than as original source references, losing whatever filename or URI was originally referenced. Even resources declared within the same page are seen serialized into the point where they were referenced, rather than being preserved as a key of a resource collection.

## Event Handling is Not Preserved

When event handlers that are added through XAML are serialized, they are not preserved. XAML without code-behind (and also without the related x:Code mechanism) has no way of serializing runtime procedural logic. Because serialization is self-contained and limited to the logical tree, there is no facility for storing the event handlers. As a result, event handler attributes, both the attribute itself and the string value that names the handler, are removed from the output XAML.

## Realistic Scenarios for Use of XAMLWriter.Save

While the limitations listed here are fairly substantial, there are still several appropriate scenarios for using [Save](#) for serialization.

- Vector or graphical output: The output of the rendered area can be used to reproduce the same vector or graphics when reloaded.

- Rich text and flow documents: Text and all element formatting and element containment within it is preserved in the output. This can be useful for mechanisms that approximate a clipboard functionality.
- Preserving business object data: If you have stored data in custom elements, such as XML data, so long as your business objects follow basic XAML rules such as providing custom constructors and conversion for by-reference property values, these business objects can be perpetuated through serialization.

# Initialization for Object Elements Not in an Object Tree

11/3/2019 • 3 minutes to read • [Edit Online](#)

Some aspects of Windows Presentation Foundation (WPF) initialization are deferred to processes that typically rely on that element being connected to either the logical tree or visual tree. This topic describes the steps that may be necessary in order to initialize an element that is not connected to either tree.

## Elements and the Logical Tree

When you create an instance of a Windows Presentation Foundation (WPF) class in code, you should be aware that several aspects of object initialization for a Windows Presentation Foundation (WPF) class are deliberately not a part of the code that is executed when calling the class constructor. Particularly for a control class, most of the visual representation of that control is not defined by the constructor. Instead, the visual representation is defined by the control's template. The template potentially comes from a variety of sources, but most often the template is obtained from theme styles. Templates are effectively late-binding; the necessary template is not attached to the control in question until the control is ready for layout. And the control is not ready for layout until it is attached to a logical tree that connects to a rendering surface at the root. It is that root-level element that initiates the rendering of all of its child elements as defined in the logical tree.

The visual tree also participates in this process. Elements that are part of the visual tree through the templates are also not fully instantiated until connected.

The consequences of this behavior are that certain operations that rely on the completed visual characteristics of an element require additional steps. An example is if you are attempting to get the visual characteristics of a class that was constructed but not yet attached to a tree. For instance, if you want to call [Render](#) on a [RenderTargetBitmap](#) and the visual you are passing is an element not connected to a tree, that element is not visually complete until additional initialization steps are completed.

### Using BeginInit and EndInit to Initialize the Element

Various classes in WPF implement the [ISupportInitialize](#) interface. You use the [BeginInit](#) and [EndInit](#) methods of the interface to denote a region in your code that contains initialization steps (such as setting property values that affect rendering). After [EndInit](#) is called in the sequence, the layout system can process the element and start looking for an implicit style.

If the element you are setting properties on is a [FrameworkElement](#) or [FrameworkContentElement](#) derived class, then you can call the class versions of [BeginInit](#) and [EndInit](#) rather than casting to [ISupportInitialize](#).

### Sample Code

The following example is sample code for a console application that uses rendering APIs and [XamlReader.Load\(Stream\)](#) of a loose XAML file to illustrate the proper placement of [BeginInit](#) and [EndInit](#) around other API calls that adjust properties that affect rendering.

The example illustrates the main function only. The functions [Rasterize](#) and [Save](#) (not shown) are utility functions that take care of image processing and IO.

```
[STAThread]
static void Main(string[] args)
{
    UIElement e;
    string file = Directory.GetCurrentDirectory() + "\\starting.xaml";
    using (Stream stream = File.Open(file, FileMode.Open))
    {
        // loading files from current directory, project settings take care of copying the file
        ParserContext pc = new ParserContext();
        pc.BaseUri = new Uri(file, UriKind.Absolute);
        e = (UIElement)XamlReader.Load(stream, pc);
    }

    Size paperSize = new Size(8.5 * 96, 11 * 96);
    e.Measure(paperSize);
    e.Arrange(new Rect(paperSize));
    e.UpdateLayout();

/*
 *    Render effect at normal dpi, indicator is the original RED rectangle
 */
RenderTargetBitmap image1 = Rasterize(e, paperSize.Width, paperSize.Height, 96, 96);
Save(image1, "render1.png");

Button b = new Button();
b.BeginInit();
b.Background = Brushes.Blue;
b.Width = b.Height = 200;
b.EndInit();
b.Measure(paperSize);
b.Arrange(new Rect(paperSize));
b.UpdateLayout();

// now render the altered version, with the element built up and initialized

RenderTargetBitmap image2 = Rasterize(b, paperSize.Width, paperSize.Height, 96, 96);
Save(image2, "render2.png");
}
```

```

<STAThread>
Shared Sub Main(ByVal args() As String)
    Dim e As UIElement
    Dim _file As String = Directory.GetCurrentDirectory() & "\starting.xaml"
    Using stream As Stream = File.Open(_file, FileMode.Open)
        ' loading files from current directory, project settings take care of copying the file
        Dim pc As New ParserContext()
        pc.BaseUri = New Uri(_file, UriKind.Absolute)
        e = CType(XamlReader.Load(stream, pc), UIElement)
    End Using

    Dim paperSize As New Size(8.5 * 96, 11 * 96)
    e.Measure(paperSize)
    e.Arrange(New Rect(paperSize))
    e.UpdateLayout()

    '
    * Render effect at normal dpi, indicator is the original RED rectangle

    Dim image1 AsRenderTargetBitmap = Rasterize(e, paperSize.Width, paperSize.Height, 96, 96)
    Save(image1, "render1.png")

    Dim b As New Button()
    b.BeginInit()
    b.Background = Brushes.Blue
    b.Height = 200
    b.Width = b.Height
    b.EndInit()
    b.Measure(paperSize)
    b.Arrange(New Rect(paperSize))
    b.UpdateLayout()

    ' now render the altered version, with the element built up and initialized

    Dim image2 AsRenderTargetBitmap = Rasterize(b, paperSize.Width, paperSize.Height, 96, 96)
    Save(image2, "render2.png")
End Sub

```

## See also

- [Trees in WPF](#)
- [WPF Graphics Rendering Overview](#)
- [XAML Overview \(WPF\)](#)

# Element Tree and Serialization How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to use the WPF element tree.

## In This Section

[Find an Element by Its Name](#)

[Override the Logical Tree](#)

## Reference

[LogicalTreeHelper](#)

[VisualTreeHelper](#)

[System.Windows.Markup](#)

## Related Sections

# How to: Find an Element by Its Name

3/5/2019 • 2 minutes to read • [Edit Online](#)

This example describes how to use the [FindName](#) method to find an element by its [Name](#) value.

## Example

In this example, the method to find a particular element by its name is written as the event handler of a button.

`stackPanel` is the [Name](#) of the root [FrameworkElement](#) being searched, and the example method then visually indicates the found element by casting it as [TextBlock](#) and changing one of the [TextBlock](#) visible UI properties.

```
void Find(object sender, RoutedEventArgs e)
{
    object wantedNode = stackPanel.FindName("dog");
    if (wantedNode is TextBlock)
    {
        // Following executed if Text element was found.
        TextBlock wantedChild = wantedNode as TextBlock;
        wantedChild.Foreground = Brushes.Blue;
    }
}
```

```
Private Sub Find(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim wantedNode As Object = stackPanel.FindName("dog")
    If TypeOf wantedNode Is TextBlock Then
        ' Following executed if Text element was found.
        Dim wantedChild As TextBlock = TryCast(wantedNode, TextBlock)
        wantedChild.Foreground = Brushes.Blue
    End If
End Sub
```

# How to: Override the Logical Tree

3/5/2019 • 2 minutes to read • [Edit Online](#)

Although it is not necessary in most cases, advanced control authors have the option to override the logical tree.

## Example

This example describes how to subclass [StackPanel](#) to override the logical tree, in this case to enforce a behavior that the panel may only have and will only render a single child element. This isn't necessarily a practically desirable behavior, but is shown here as a means of illustrating the scenario for overriding an element's normal logical tree.

```
public class SingletonPanel : StackPanel
{
    //private UIElementCollection _children;
    private FrameworkElement _child;

    public SingletonPanel()
    {

    }

    public FrameworkElement SingleChild
    {

        get { return _child; }
        set
        {
            if (value == null)
            {
                RemoveLogicalChild(_child);
            }
            else
            {
                if (_child == null)
                {
                    _child = value;
                }
                else
                {
                    // raise an exception?
                    MessageBox.Show("Needs to be a single element");
                }
            }
        }
    }

    public void SetSingleChild(object child)
    {
        this.AddLogicalChild(child);
    }

    public new void AddLogicalChild(object child)
    {
        _child = (FrameworkElement)child;
        if (this.Children.Count == 1)
        {
            this.RemoveLogicalChild(this.Children[0]);
            this.Children.Add((UIElement)child);
        }
        else
    }
}
```

```
        {
            this.Children.Add((UIElement)child);
        }
    }

    public new void RemoveLogicalChild(object child)
    {
        _child = null;
        this.Children.Clear();
    }
    protected override IEnumerator LogicalChildren
    {
        get
        {
            // cheat, make a list with one member and return the enumerator
            ArrayList _list = new ArrayList();
            _list.Add(_child);
            return (IEnumerator)_list.GetEnumerator();
        }
    }
}
```

For more information on the logical tree, see [Trees in WPF](#).

# Properties (WPF)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a set of services that can be used to extend the functionality of a common language runtime (CLR) property. Collectively, these services are typically referred to as the WPF property system. A property that is backed by the WPF property system is known as a dependency property.

## In This Section

- [Dependency Properties Overview](#)
- [Attached Properties Overview](#)
- [Custom Dependency Properties](#)
- [Dependency Property Metadata](#)
- [Dependency Property Callbacks and Validation](#)
- [Framework Property Metadata](#)
- [Dependency Property Value Precedence](#)
- [Read-Only Dependency Properties](#)
- [Property Value Inheritance](#)
- [Dependency Property Security](#)
- [Safe Constructor Patterns for DependencyObjects](#)
- [Collection-Type Dependency Properties](#)
- [XAML Loading and Dependency Properties](#)
- [How-to Topics](#)

## Reference

[DependencyProperty](#)

[PropertyMetadata](#)

[FrameworkPropertyMetadata](#)

[DependencyObject](#)

## Related Sections

[WPF Architecture](#)

[XAML in WPF](#)

[Base Elements](#)

[Element Tree and Serialization](#)

[Events](#)

[Input](#)

[Resources](#)

[WPF Content Model](#)

[Threading Model](#)

# Dependency properties overview

11/12/2019 • 13 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a set of services that can be used to extend the functionality of a type's [property](#). Collectively, these services are typically referred to as the WPF property system. A property that is backed by the WPF property system is known as a dependency property. This overview describes the WPF property system and the capabilities of a dependency property. This includes how to use existing dependency properties in XAML and in code. This overview also introduces specialized aspects of dependency properties, such as dependency property metadata, and how to create your own dependency property in a custom class.

## Prerequisites

This topic assumes that you have some basic knowledge of the .NET type system and object-oriented programming. In order to follow the examples in this topic, you should also understand XAML and know how to write WPF applications. For more information, see [Walkthrough: My first WPF desktop application](#).

## Dependency properties and CLR properties

In WPF, properties are typically exposed as standard .NET [properties](#). At a basic level, you could interact with these properties directly and never know that they are implemented as a dependency property. However, you should become familiar with some or all of the features of the WPF property system, so that you can take advantage of these features.

The purpose of dependency properties is to provide a way to compute the value of a property based on the value of other inputs. These other inputs might include system properties such as themes and user preference, just-in-time property determination mechanisms such as data binding and animations/storyboards, multiple-use templates such as resources and styles, or values known through parent-child relationships with other elements in the element tree. In addition, a dependency property can be implemented to provide self-contained validation, default values, callbacks that monitor changes to other properties, and a system that can coerce property values based on potentially runtime information. Derived classes can also change some specific characteristics of an existing property by overriding dependency property metadata, rather than overriding the actual implementation of existing properties or creating new properties.

In the SDK reference, you can identify which property is a dependency property by the presence of the Dependency Property Information section on the managed reference page for that property. The Dependency Property Information section includes a link to the [DependencyProperty](#) identifier field for that dependency property, and also includes a list of the metadata options that are set for that property, per-class override information, and other details.

## Dependency properties back CLR properties

Dependency properties and the WPF property system extend property functionality by providing a type that backs a property, as an alternative implementation to the standard pattern of backing the property with a private field. The name of this type is [DependencyProperty](#). The other important type that defines the WPF property system is [DependencyObject](#). [DependencyObject](#) defines the base class that can register and own a dependency property.

The following lists the terminology that is used with dependency properties:

- **Dependency property:** A property that is backed by a [DependencyProperty](#).
- **Dependency property identifier:** A [DependencyProperty](#) instance, which is obtained as a return value when registering a dependency property, and then stored as a static member of a class. This identifier is used as a parameter for many of the APIs that interact with the WPF property system.
- **CLR "wrapper":** The actual get and set implementations for the property. These implementations incorporate the dependency property identifier by using it in the [GetValue](#) and [SetValue](#) calls, thus providing the backing for the property using the WPF property system.

The following example defines the `IsSpinning` dependency property, and shows the relationship of the [DependencyProperty](#) identifier to the property that it backs.

```
public static readonly DependencyProperty IsSpinningProperty =
    DependencyProperty.Register(
        "IsSpinning", typeof(Boolean),
        typeof(MyCode)
    );
public bool IsSpinning
{
    get { return (bool)GetValue(IsSpinningProperty); }
    set { SetValue(IsSpinningProperty, value); }
}
```

```
Public Shared ReadOnly IsSpinningProperty As DependencyProperty =
    DependencyProperty.Register("IsSpinning",
        GetType(Boolean),
        GetType(MyCode))

Public Property IsSpinning() As Boolean
    Get
        Return CBool(GetValue(IsSpinningProperty))
    End Get
    Set(ByVal value As Boolean)
        SetValue(IsSpinningProperty, value)
    End Set
End Property
```

The naming convention of the property and its backing [DependencyProperty](#) field is important. The name of the field is always the name of the property, with the suffix `Property` appended. For more information about this convention and the reasons for it, see [Custom Dependency Properties](#).

## Setting property values

You can set properties either in code or in XAML.

### Setting property values in XAML

The following XAML example specifies the background color of a button as red. This example illustrates a case where the simple string value for a XAML attribute is type-converted by the WPF XAML parser into a WPF type (a [Color](#), by way of a [SolidColorBrush](#)) in the generated code.

```
<Button Background="Red" Content="Button!" />
```

XAML supports a variety of syntax forms for setting properties. Which syntax to use for a particular property will depend on the value type that a property uses, as well as other factors such as the presence of a type converter. For more information on XAML syntax for property setting, see [XAML Overview \(WPF\)](#) and [XAML Syntax In Detail](#).

As an example of non-attribute syntax, the following XAML example shows another button background. This time rather than setting a simple solid color, the background is set to an image, with an element representing that image and the source of that image specified as an attribute of the nested element. This is an example of property element syntax.

```
<Button Content="Button!">
  <Button.Background>
    <ImageBrush ImageSource="wavy.jpg"/>
  </Button.Background>
</Button>
```

### Setting properties in code

Setting dependency property values in code is typically just a call to the set implementation exposed by the CLR "wrapper".

```
Button myButton = new Button();
myButton.Width = 200.0;
```

```
Dim myButton As New Button()
myButton.Width = 200.0
```

Getting a property value is also essentially a call to the get "wrapper" implementation:

```
double whatWidth;
whatWidth = myButton.Width;
```

```
Dim whatWidth As Double
whatWidth = myButton.Width
```

You can also call the property system APIs [GetValue](#) and [SetValue](#) directly. This is not typically necessary if you are using existing properties (the wrappers are more convenient, and provide better exposure of the property for developer tools), but calling the APIs directly is appropriate for certain scenarios.

Properties can be also set in XAML and then accessed later in code, through code-behind. For details, see [Code-Behind and XAML in WPF](#).

## Property functionality provided by a dependency property

A dependency property provides functionality that extends the functionality of a property as opposed to a property that is backed by a field. Often, such functionality represents or supports one of the following specific features:

- [Resources](#)
- [Data binding](#)
- [Styles](#)
- [Animations](#)
- [Metadata overrides](#)
- [Property value inheritance](#)

- WPF Designer integration

## Resources

A dependency property value can be set by referencing a resource. Resources are typically specified as the `Resources` property value of a page root element, or of the application (these locations enable the most convenient access to the resource). The following example shows how to define a `SolidColorBrush` resource.

```
<DockPanel.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
</DockPanel.Resources>
```

Once the resource is defined, you can reference the resource and use it to provide a property value:

```
<Button Background="{DynamicResource MyBrush}" Content="I am gold" />
```

This particular resource is referenced as a [DynamicResource Markup Extension](#) (in WPF XAML, you can use either a static or dynamic resource reference). To use a dynamic resource reference, you must be setting to a dependency property, so it is specifically the dynamic resource reference usage that is enabled by the WPF property system. For more information, see [XAML Resources](#).

### NOTE

Resources are treated as a local value, which means that if you set another local value, you will eliminate the resource reference. For more information, see [Dependency Property Value Precedence](#).

## Data binding

A dependency property can reference a value through data binding. Data binding works through a specific markup extension syntax in XAML, or the `Binding` object in code. With data binding, the final property value determination is deferred until run time, at which time the value is obtained from a data source.

The following example sets the `Content` property for a `Button`, using a binding declared in XAML. The binding uses an inherited data context and an `XmlDataProvider` data source (not shown). The binding itself specifies the desired source property by `XPath` within the data source.

```
<Button Content="{Binding XPath=Team/@TeamName}" />
```

### NOTE

Bindings are treated as a local value, which means that if you set another local value, you will eliminate the binding. For details, see [Dependency Property Value Precedence](#).

Dependency properties, or the `DependencyObject` class, do not natively support `INotifyPropertyChanged` for purposes of producing notifications of changes in `DependencyObject` source property value for data binding operations. For more information on how to create properties for use in data binding that can report changes to a data binding target, see [Data Binding Overview](#).

## Styles

Styles and templates are two of the chief motivating scenarios for using dependency properties. Styles are particularly useful for setting properties that define application user interface (UI). Styles are typically defined as resources in XAML. Styles interact with the property system because they typically contain "setters" for particular properties, as well as "triggers" that change a property value based on the real-time value for

another property.

The following example creates a very simple style (which would be defined inside a [Resources](#) dictionary, not shown), then applies that style directly to the [Style](#) property for a [Button](#). The setter within the style sets the [Background](#) property for a styled [Button](#) to green.

```
<Style x:Key="GreenButtonStyle">
    <Setter Property="Control.Background" Value="Green"/>
</Style>
```

```
<Button Style="{StaticResource GreenButtonStyle}">I am green!</Button>
```

For more information, see [Styling and Templating](#).

## Animations

Dependency properties can be animated. When an animation is applied and is running, the animated value operates at a higher precedence than any value (such as a local value) that the property otherwise has.

The following example animates the [Background](#) on a [Button](#) property (technically, the [Background](#) is animated by using property element syntax to specify a blank [SolidColorBrush](#) as the [Background](#), then the [Color](#) property of that [SolidColorBrush](#) is the property that is directly animated).

```
<Button>I am animated
<Button.Background>
    <SolidColorBrush x:Name="AnimBrush"/>
</Button.Background>
<Button.Triggers>
    <EventTrigger RoutedEvent="Button.Loaded">
        <BeginStoryboard>
            <Storyboard>
                <ColorAnimation
                    Storyboard.TargetName="AnimBrush"
                    Storyboard.TargetProperty="(SolidColorBrush.Color)"
                    From="Red" To="Green" Duration="0:0:5"
                    AutoReverse="True" RepeatBehavior="Forever" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Button.Triggers>
</Button>
```

For more information on animating properties, see [Animation Overview](#) and [Storyboards Overview](#).

## Metadata overrides

You can change certain behaviors of a dependency property by overriding the metadata for that property when you derive from the class that originally registers the dependency property. Overriding metadata relies on the [DependencyProperty](#) identifier. Overriding metadata does not require re-implementing the property. The metadata change is handled natively by the property system; each class potentially holds individual metadata for all properties that are inherited from base classes, on a per-type basis.

The following example overrides metadata for a dependency property [DefaultStyleKey](#). Overriding this particular dependency property metadata is part of an implementation pattern that creates controls that can use default styles from themes.

```
public class SpinnerControl : ItemsControl
{
    static SpinnerControl()
    {
        DefaultStyleKeyProperty.OverrideMetadata(
            typeof(SpinnerControl),
            new FrameworkPropertyMetadata(typeof(SpinnerControl))
        );
    }
}
```

```
Public Class SpinnerControl
    Inherits ItemsControl
    Shared Sub New()
        DefaultStyleKeyProperty.OverrideMetadata(GetType(SpinnerControl), New
FrameworkPropertyMetadata(GetType(SpinnerControl)))
    End Sub
End Class
```

For more information about overriding or obtaining property metadata, see [Dependency Property Metadata](#).

### Property value inheritance

An element can inherit the value of a dependency property from its parent in the object tree.

#### NOTE

Property value inheritance behavior is not globally enabled for all dependency properties, because the calculation time for inheritance does have some performance impact. Property value inheritance is typically only enabled for properties where a particular scenario suggests that property value inheritance is appropriate. You can determine whether a dependency property inherits by looking at the **Dependency Property Information** section for that dependency property in the SDK reference.

The following example shows a binding, and sets the **DataContext** property that specifies the source of the binding, which was not shown in the earlier binding example. Any subsequent bindings in child objects do not need to specify the source, they can use the inherited value from **DataContext** in the parent **StackPanel** object. (Alternatively, a child object could instead choose to directly specify its own **DataContext** or a **Source** in the **Binding**, and to deliberately not use the inherited value for data context of its bindings.)

```
<StackPanel Canvas.Top="50" DataContext="{Binding Source={StaticResource XmlTeamsSource}}">
    <Button Content="{Binding XPath=Team/@TeamName}" />
</StackPanel>
```

For more information, see [Property Value Inheritance](#).

### WPF designer integration

A custom control with properties that are implemented as dependency properties will receive appropriate WPF Designer for Visual Studio support. One example is the ability to edit direct and attached dependency properties with the **Properties** window. For more information, see [Control Authoring Overview](#).

## Dependency property value precedence

When you get the value of a dependency property, you are potentially obtaining a value that was set on that property through any one of the other property-based inputs that participate in the WPF property system. Dependency property value precedence exists so that a variety of scenarios for how properties obtain their values can interact in a predictable way.

Consider the following example. The example includes a style that applies to all buttons and their [Background](#) properties, but then also specifies one button with a locally set [Background](#) value.

#### NOTE

The SDK documentation uses the terms "local value" or "locally set value" occasionally when discussing dependency properties. A locally set value is a property value that is set directly on an object instance in code, or as an attribute on an element in XAML.

In principle, for the first button, the property is set twice, but only one value applies: the value with the highest precedence. A locally set value has the highest precedence (except for a running animation, but no animation applies in this example) and thus the locally set value is used instead of the style setter value for the background on the first button. The second button has no local value (and no other value with higher precedence than a style setter) and thus the background in that button comes from the style setter.

```
<StackPanel>
  <StackPanel.Resources>
    <Style x:Key="{x:Type Button}" TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Red"/>
    </Style>
  </StackPanel.Resources>
  <Button Background="Green">I am NOT red!</Button>
  <Button>I am styled red</Button>
</StackPanel>
```

#### Why does dependency property precedence exist?

Typically, you would not want styles to always apply and to obscure even a locally set value of an individual element (otherwise, it would be very difficult to use either styles or elements in general). Therefore, the values that come from styles operate at a lower precedent than a locally set value. For a more thorough listing of dependency properties and where a dependency property effective value might come from, see [Dependency Property Value Precedence](#).

#### NOTE

There are a number of properties defined on WPF elements that are not dependency properties. By and large, properties were implemented as dependency properties only when there were needs to support at least one of the scenarios enabled by the property system: data binding, styling, animation, default value support, inheritance, attached properties, or invalidation.

## Learning more about dependency properties

- An attached property is a type of property that supports a specialized syntax in XAML. An attached property often does not have a 1:1 correspondence with a common language runtime (CLR) property, and is not necessarily a dependency property. The typical purpose of a attached property is to allow child elements to report property values to a parent element, even if the parent element and child element do not both possess that property as part of the class members listings. One primary scenario is to enable child elements to inform the parent how they should be presented in UI; for an example, see [Dock](#) or [Left](#). For details, see [Attached Properties Overview](#).
- Component developers or application developers may wish to create their own dependency property, in order to enable capabilities such as data binding or styles support, or for invalidation and value coercion support. For details, see [Custom Dependency Properties](#).
- Dependency properties should generally be considered to be public properties, accessible or at least

discoverable by any caller that has access to an instance. For more information, see [Dependency Property Security](#).

## See also

- [Custom Dependency Properties](#)
- [Read-Only Dependency Properties](#)
- [XAML Overview \(WPF\)](#)
- [WPF Architecture](#)

# Attached Properties Overview

11/14/2019 • 10 minutes to read • [Edit Online](#)

An attached property is a concept defined by XAML. An attached property is intended to be used as a type of global property that is settable on any object. In Windows Presentation Foundation (WPF), attached properties are typically defined as a specialized form of dependency property that does not have the conventional property "wrapper".

## Prerequisites

This topic assumes that you understand dependency properties from the perspective of a consumer of existing dependency properties on Windows Presentation Foundation (WPF) classes, and have read the [Dependency Properties Overview](#). To follow the examples in this topic, you should also understand XAML and know how to write WPF applications.

## Why Use Attached Properties

One purpose of an attached property is to allow different child elements to specify unique values for a property that is actually defined in a parent element. A specific application of this scenario is having child elements inform the parent element of how they are to be presented in the user interface (UI). One example is the [DockPanel.Dock](#) property. The [DockPanel.Dock](#) property is created as an attached property because it is designed to be set on elements that are contained within a [DockPanel](#), rather than on [DockPanel](#) itself. The [DockPanel](#) class defines the static [DependencyProperty](#) field named [DockProperty](#), and then provides the [GetDock](#) and [SetDock](#) methods as public accessors for the attached property.

## Attached Properties in XAML

In XAML, you set attached properties by using the syntax *AttachedPropertyProvider.PropertyName*

The following is an example of how you can set [DockPanel.Dock](#) in XAML:

```
<DockPanel>
  <CheckBox DockPanel.Dock="Top">Hello</CheckBox>
</DockPanel>
```

Note that the usage is somewhat similar to a static property; you always reference the type [DockPanel](#) that owns and registers the attached property, rather than referring to any instance specified by name.

Also, because an attached property in XAML is an attribute that you set in markup, only the set operation has any relevance. You cannot directly get a property in XAML, although there are some indirect mechanisms for comparing values, such as triggers in styles (for details, see [Styling and Templating](#)).

### Attached Property Implementation in WPF

In Windows Presentation Foundation (WPF), most of the attached properties that exist on WPF types that are related to UI presentation are implemented as dependency properties. Attached properties are a XAML concept, whereas dependency properties are a WPF concept. Because WPF attached properties are dependency properties, they support dependency property concepts such as property metadata, and default values from that property metadata.

## How Attached Properties Are Used by the Owning Type

Although attached properties are settable on any object, that does not automatically mean that setting the property will produce a tangible result, or that the value will ever be used by another object. Generally, attached properties are intended so that objects coming from a wide variety of possible class hierarchies or logical relationships can each report common information to the type that defines the attached property. The type that defines the attached property typically follows one of these models:

- The type that defines the attached property is designed so that it can be the parent element of the elements that will set values for the attached property. The type then iterates its child objects through internal logic against some object tree structure, obtains the values, and acts on those values in some manner.
- The type that defines the attached property will be used as the child element for a variety of possible parent elements and content models.
- The type that defines the attached property represents a service. Other types set values for the attached property. Then, when the element that set the property is evaluated in the context of the service, the attached property values are obtained through internal logic of the service class.

### An Example of a Parent-Defined Attached Property

The most typical scenario where WPF defines an attached property is when a parent element supports a child element collection, and also implements a behavior where the specifics of the behavior are reported individually for each child element.

[DockPanel](#) defines the [DockPanel.Dock](#) attached property, and [DockPanel](#) has class-level code as part of its rendering logic (specifically, [MeasureOverride](#) and [ArrangeOverride](#)). A [DockPanel](#) instance will always check to see whether any of its immediate child elements have set a value for [DockPanel.Dock](#). If so, those values become input for the rendering logic applied to that particular child element. Nested [DockPanel](#) instances each treat their own immediate child element collections, but that behavior is implementation-specific to how [DockPanel](#) processes [DockPanel.Dock](#) values. It is theoretically possible to have attached properties that influence elements beyond the immediate parent. If the [DockPanel.Dock](#) attached property is set on an element that has no [DockPanel](#) parent element to act upon it, no error or exception is raised. This simply means that a global property value was set, but it has no current [DockPanel](#) parent that could consume the information.

## Attached Properties in Code

Attached properties in WPF do not have the typical CLR "wrapper" methods for easy get/set access. This is because the attached property is not necessarily part of the CLR namespace for instances where the property is set. However, a XAML processor must be able to set those values when XAML is parsed. To support an effective attached property usage, the owner type of the attached property must implement dedicated accessor methods in the form **Get\_PropertyName\_** and **Set\_PropertyName\_**. These dedicated accessor methods are also useful to get or set the attached property in code. From a code perspective, an attached property is similar to a backing field that has method accessors instead of property accessors, and that backing field can exist on any object rather than needing to be specifically defined.

The following example shows how you can set an attached property in code. In this example, `myCheckBox` is an instance of the [CheckBox](#) class.

```
DockPanel myDockPanel = new DockPanel();
CheckBox myCheckBox = new CheckBox();
myCheckBox.Content = "Hello";
myDockPanel.Children.Add(myCheckBox);
DockPanel.SetDock(myCheckBox, Dock.Top);
```

```
Dim myDockPanel As New DockPanel()
Dim myCheckBox As New CheckBox()
myCheckBox.Content = "Hello"
myDockPanel.Children.Add(myCheckBox)
DockPanel.SetDock(myCheckBox, Dock.Top)
```

Similar to the XAML case, if `myCheckBox` had not already been added as a child element of `myDockPanel` by the third line of code, the fourth line of code would not raise an exception, but the property value would not interact with a `DockPanel` parent and thus would do nothing. Only a `DockPanel.Dock` value set on a child element combined with the presence of a `DockPanel` parent element will cause an effective behavior in the rendered application. (In this case, you could set the attached property, then attach to the tree. Or you could attach to the tree then set the attached property. Either action order provides the same result.)

## Attached Property Metadata

When registering the property, `FrameworkPropertyMetadata` is set to specify characteristics of the property, such as whether the property affects rendering, measurement, and so on. Metadata for an attached property is generally no different than on a dependency property. If you specify a default value in an override to attached property metadata, that value becomes the default value of the implicit attached property on instances of the overriding class. Specifically, your default value is reported if some process queries for the value of an attached property through the `Get` method accessor for that property, specifying an instance of the class where you specified the metadata, and the value for that attached property was otherwise not set.

If you want to enable property value inheritance on a property, you should use attached properties rather than non-attached dependency properties. For details, see [Property Value Inheritance](#).

## Custom Attached Properties

### When to Create an Attached Property

You might create an attached property when there is a reason to have a property setting mechanism available for classes other than the defining class. The most common scenario for this is layout. Examples of existing layout properties are `DockPanel.Dock`, `Panel.ZIndex`, and `Canvas.Top`. The scenario enabled here is that elements that exist as child elements to layout-controlling elements are able to express layout requirements to their layout parent elements individually, each setting a property value that the parent defined as an attached property.

Another scenario for using an attached property is when your class represents a service, and you want classes to be able to integrate the service more transparently.

Yet another scenario is to receive Visual Studio WPF Designer support, such as **Properties** window editing. For more information, see [Control Authoring Overview](#).

As mentioned before, you should register as an attached property if you want to use property value inheritance.

### How to Create an Attached Property

If your class is defining the attached property strictly for use on other types, then the class does not have to derive from `DependencyObject`. But you do need to derive from `DependencyObject` if you follow the overall WPF model of having your attached property also be a dependency property.

Define your attached property as a dependency property by declaring a `public static readonly` field of type `DependencyProperty`. You define this field by using the return value of the `RegisterAttached` method. The field name must match the attached property name, appended with the string `Property`, to follow the established WPF pattern of naming the identifying fields versus the properties that they represent. The attached property provider must also provide static `Get_PropertyName_` and `Set_PropertyName_` methods as accessors for the attached property; failing to do this will result in the property system being unable to use your attached property.

#### NOTE

If you omit the attached property's get accessor, data binding on the property will not work in design tools, such as Visual Studio and Blend for Visual Studio.

#### The Get Accessor

The signature for the **Get\_PropertyName\_** accessor must be:

```
public static object GetPropertyName(object target)
```

- The `target` object can be specified as a more specific type in your implementation. For example, the `DockPanel.GetDock` method types the parameter as `UIElement`, because the attached property is only intended to be set on `UIElement` instances.
- The return value can be specified as a more specific type in your implementation. For example, the `GetDock` method types it as `Dock`, because the value can only be set to that enumeration.

#### The Set Accessor

The signature for the **Set\_PropertyName\_** accessor must be:

```
public static void SetPropertyName(object target, object value)
```

- The `target` object can be specified as a more specific type in your implementation. For example, the `SetDock` method types it as `UIElement`, because the attached property is only intended to be set on `UIElement` instances.
- The `value` object can be specified as a more specific type in your implementation. For example, the `SetDock` method types it as `Dock`, because the value can only be set to that enumeration. Remember that the value for this method is the input coming from the XAML loader when it encounters your attached property in an attached property usage in markup. That input is the value specified as a XAML attribute value in markup. Therefore there must be type conversion, value serializer, or markup extension support for the type you use, such that the appropriate type can be created from the attribute value (which is ultimately just a string).

The following example shows the dependency property registration (using the `RegisterAttached` method), as well as the **Get\_PropertyName\_** and **Set\_PropertyName\_** accessors. In the example, the attached property name is `IsBubbleSource`. Therefore, the accessors must be named `GetIsBubbleSource` and `SetIsBubbleSource`.

```
public static readonly DependencyProperty IsBubbleSourceProperty = DependencyProperty.RegisterAttached(
    "IsBubbleSource",
    typeof(Boolean),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(false, FrameworkPropertyMetadataOptions.AffectsRender)
);
public static void SetIsBubbleSource(UIElement element, Boolean value)
{
    element.SetValue(IsBubbleSourceProperty, value);
}
public static Boolean GetIsBubbleSource(UIElement element)
{
    return (Boolean)element.GetValue(IsBubbleSourceProperty);
}
```

```

Public Shared ReadOnly IsBubbleSourceProperty As DependencyProperty =
    DependencyProperty.RegisterAttached("IsBubbleSource", GetType(Boolean), GetType(AquariumObject), New
    FrameworkPropertyMetadata(False, FrameworkPropertyMetadataOptions.AffectsRender))
Public Shared Sub SetIsBubbleSource(ByVal element As UIElement, ByVal value As Boolean)
    element.SetValue(IsBubbleSourceProperty, value)
End Sub
Public Shared Function GetIsBubbleSource(ByVal element As UIElement) As Boolean
    Return CType(element.GetValue(IsBubbleSourceProperty), Boolean)
End Function

```

### Attached Property Attributes

WPF defines several .NET attributes that are intended to provide information about attached properties to reflection processes, and to typical users of reflection and property information such as designers. Because attached properties have a type of unlimited scope, designers need a way to avoid overwhelming users with a global list of all the attached properties that are defined in a particular technology implementation that uses XAML. The .NET attributes that WPF defines for attached properties can be used to scope the situations where a given attached property should be shown in a properties window. You might consider applying these attributes for your own custom attached properties also. The purpose and syntax of the .NET attributes is described on the appropriate reference pages:

- [AttachedPropertyBrowsableAttribute](#)
- [AttachedPropertyBrowsableForChildrenAttribute](#)
- [AttachedPropertyBrowsableForTypeAttribute](#)
- [AttachedPropertyBrowsableWhenAttributePresentAttribute](#)

## Learning More About Attached Properties

- For more information on creating an attached property, see [Register an Attached Property](#).
- For more advanced usage scenarios for dependency properties and attached properties, see [Custom Dependency Properties](#).
- You can also register a property as an attached property, and as a dependency property, but then still expose "wrapper" implementations. In this case, the property can be set either on that element, or on any element through the XAML attached property syntax. An example of a property with an appropriate scenario for both standard and attached usages is [FrameworkElement.FlowDirection](#).

## See also

- [DependencyProperty](#)
- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)
- [XAML Overview \(WPF\)](#)
- [Register an Attached Property](#)

# Custom Dependency Properties

11/7/2019 • 14 minutes to read • [Edit Online](#)

This topic describes the reasons that Windows Presentation Foundation (WPF) application developers and component authors might want to create custom dependency property, and describes the implementation steps as well as some implementation options that can improve performance, usability, or versatility of the property.

## Prerequisites

This topic assumes that you understand dependency properties from the perspective of a consumer of existing dependency properties on WPF classes, and have read the [Dependency Properties Overview](#) topic. In order to follow the examples in this topic, you should also understand Extensible Application Markup Language (XAML) and know how to write WPF applications.

## What Is a Dependency Property?

You can enable what would otherwise be a common language runtime (CLR) property to support styling, data binding, inheritance, animations, and default values by implementing it as a dependency property. Dependency properties are properties that are registered with the WPF property system by calling the [Register](#) method (or [RegisterReadOnly](#)), and that are backed by a [DependencyProperty](#) identifier field. Dependency properties can be used only by [DependencyObject](#) types, but [DependencyObject](#) is quite high in the WPF class hierarchy, so the majority of classes available in WPF can support dependency properties. For more information about dependency properties and some of the terminology and conventions used for describing them in this SDK, see [Dependency Properties Overview](#).

## Examples of Dependency Properties

Examples of dependency properties that are implemented on WPF classes include the [Background](#) property, the [Width](#) property, and the [Text](#) property, among many others. Each dependency property exposed by a class has a corresponding public static field of type [DependencyProperty](#) exposed on that same class. This is the identifier for the dependency property. The identifier is named using a convention: the name of the dependency property with the string `Property` appended to it. For example, the corresponding [DependencyProperty](#) identifier field for the [Background](#) property is [BackgroundProperty](#). The identifier stores the information about the dependency property as it was registered, and the identifier is then used later for other operations involving the dependency property, such as calling [SetValue](#).

As mentioned in the [Dependency Properties Overview](#), all dependency properties in WPF (except most attached properties) are also CLR properties because of the "wrapper" implementation. Therefore, from code, you can get or set dependency properties by calling CLR accessors that define the wrappers in the same manner that you would use other CLR properties. As a consumer of established dependency properties, you do not typically use the [DependencyObject](#) methods [GetValue](#) and [SetValue](#), which are the connection point to the underlying property system. Rather, the existing implementation of the CLR properties will have already called [GetValue](#) and [SetValue](#) within the `get` and `set` wrapper implementations of the property, using the identifier field appropriately. If you are implementing a custom dependency property yourself, then you will be defining the wrapper in a similar way.

## When Should You Implement a Dependency Property?

When you implement a property on a class, so long as your class derives from [DependencyObject](#), you have

the option to back your property with a [DependencyProperty](#) identifier and thus to make it a dependency property. Having your property be a dependency property is not always necessary or appropriate, and will depend on your scenario needs. Sometimes, the typical technique of backing your property with a private field is adequate. However, you should implement your property as a dependency property whenever you want your property to support one or more of the following WPF capabilities:

- You want your property to be settable in a style. For more information, see [Styling and Templating](#).
- You want your property to support data binding. For more information about data binding dependency properties, see [Bind the Properties of Two Controls](#).
- You want your property to be settable with a dynamic resource reference. For more information, see [XAML Resources](#).
- You want to inherit a property value automatically from a parent element in the element tree. In this case, register with the [RegisterAttached](#) method, even if you also create a property wrapper for CLR access. For more information, see [Property Value Inheritance](#).
- You want your property to be animatable. For more information, see [Animation Overview](#).
- You want the property system to report when the previous value of the property has been changed by actions taken by the property system, the environment, or the user, or by reading and using styles. By using property metadata, your property can specify a callback method that will be invoked each time the property system determines that your property value was definitively changed. A related concept is property value coercion. For more information, see [Dependency Property Callbacks and Validation](#).
- You want to use established metadata conventions that are also used by WPF processes, such as reporting whether changing a property value should require the layout system to recompose the visuals for an element. Or you want to be able to use metadata overrides so that derived classes can change metadata-based characteristics such as the default value.
- You want properties of a custom control to receive Visual Studio WPF Designer support, such as **Properties** window editing. For more information, see [Control Authoring Overview](#).

When you examine these scenarios, you should also consider whether you can achieve your scenario by overriding the metadata of an existing dependency property, rather than implementing a completely new property. Whether a metadata override is practical depends on your scenario and how closely that scenario resembles the implementation in existing WPF dependency properties and classes. For more information about overriding metadata on existing properties, see [Dependency Property Metadata](#).

## Checklist for Defining a Dependency Property

Defining a dependency property consists of four distinct concepts. These concepts are not necessarily strict procedural steps, because some of these end up being combined as single lines of code in the implementation:

- (Optional) Create property metadata for the dependency property.
- Register the property name with the property system, specifying an owner type and the type of the property value. Also specify the property metadata, if used.
- Define a [DependencyProperty](#) identifier as a `public static readonly` field on the owner type.
- Define a CLR "wrapper" property whose name matches the name of the dependency property. Implement the CLR "wrapper" property's `get` and `set` accessors to connect with the dependency property that backs it.

### Registering the Property with the Property System

In order for your property to be a dependency property, you must register that property into a table maintained by the property system, and give it a unique identifier that is used as the qualifier for later property system operations. These operations might be internal operations, or your own code calling property system APIs. To register the property, you call the [Register](#) method within the body of your class (inside the class, but outside of any member definitions). The identifier field is also provided by the [Register](#) method call, as the return value. The reason that the [Register](#) call is done outside of other member definitions is because you use this return value to assign and create a `public static readonly` field of type [DependencyProperty](#) as part of your class. This field becomes the identifier for your dependency property.

```
public static readonly DependencyProperty AquariumGraphicProperty = DependencyProperty.Register(
    "AquariumGraphic",
    typeof(Uri),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(null,
        FrameworkPropertyMetadataOptions.AffectsRender,
        new PropertyChangedCallback(OnUriChanged)
    )
);
```

```
Public Shared ReadOnly AquariumGraphicProperty As DependencyProperty =
DependencyProperty.Register("AquariumGraphic", GetType(Uri), GetType(AquariumObject), New
FrameworkPropertyMetadata(Nothing, FrameworkPropertyMetadataOptions.AffectsRender, New
PropertyChangedCallback(AddressOf OnUriChanged)))
```

## Dependency Property Name Conventions

There are established naming conventions regarding dependency properties that you must follow in all but exceptional circumstances.

The dependency property itself will have a basic name, "AquariumGraphic" as in this example, which is given as the first parameter of [Register](#). That name must be unique within each registering type. Dependency properties inherited through base types are considered to be already part of the registering type; names of inherited properties cannot be registered again. However, there is a technique for adding a class as owner of a dependency property even when that dependency property is not inherited; for details, see [Dependency Property Metadata](#).

When you create the identifier field, name this field by the name of the property as you registered it, plus the suffix `Property`. This field is your identifier for the dependency property, and it will be used later as an input for the [SetValue](#) and [GetValue](#) calls you will make in the wrappers, by any other code access to the property by your own code, by any external code access you allow, by the property system, and potentially by XAML processors.

### NOTE

Defining the dependency property in the class body is the typical implementation, but it is also possible to define a dependency property in the class static constructor. This approach might make sense if you need more than one line of code to initialize the dependency property.

## Implementing the "Wrapper"

Your wrapper implementation should call [GetValue](#) in the `get` implementation, and [SetValue](#) in the `set` implementation (the original registration call and field are shown here too for clarity).

In all but exceptional circumstances, your wrapper implementations should perform only the [GetValue](#) and [SetValue](#) actions, respectively. The reason for this is discussed in the topic [XAML Loading and Dependency Properties](#).

All existing public dependency properties that are provided on the WPF classes use this simple wrapper implementation model; most of the complexity of how dependency properties work is either inherently a behavior of the property system, or is implemented through other concepts such as coercion or property change callbacks through property metadata.

```
public static readonly DependencyProperty AquariumGraphicProperty = DependencyProperty.Register(
    "AquariumGraphic",
    typeof(Uri),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(null,
        FrameworkPropertyMetadataOptions.AffectsRender,
        new PropertyChangedCallback(OnUriChanged)
    )
);
public Uri AquariumGraphic
{
    get { return (Uri)GetValue(AquariumGraphicProperty); }
    set { SetValue(AquariumGraphicProperty, value); }
}
```

```
Public Shared ReadOnly AquariumGraphicProperty As DependencyProperty =
DependencyProperty.Register("AquariumGraphic", GetType(Uri), GetType(AquariumObject), New
FrameworkPropertyMetadata(Nothing, FrameworkPropertyMetadataOptions.AffectsRender, New
PropertyChangedCallback(AddressOf OnUriChanged)))
Public Property AquariumGraphic() As Uri
    Get
        Return CType(GetValue(AquariumGraphicProperty), Uri)
    End Get
    Set(ByVal value As Uri)
        SetValue(AquariumGraphicProperty, value)
    End Set
End Property
```

Again, by convention, the name of the wrapper property must be the same as the name chosen and given as first parameter of the [Register](#) call that registered the property. If your property does not follow the convention, this does not necessarily disable all possible uses, but you will encounter several notable issues:

- Certain aspects of styles and templates will not work.
- Most tools and designers must rely on the naming conventions to properly serialize XAML, or to provide designer environment assistance at a per-property level.
- The current implementation of the WPF XAML loader bypasses the wrappers entirely, and relies on the naming convention when processing attribute values. For more information, see [XAML Loading and Dependency Properties](#).

### Property Metadata for a New Dependency Property

When you register a dependency property, the registration through the property system creates a metadata object that stores property characteristics. Many of these characteristics have defaults that are set if the property is registered with the simple signatures of [Register](#). Other signatures of [Register](#) allow you to specify the metadata that you want as you register the property. The most common metadata given for dependency properties is to give them a default value that is applied on new instances that use the property.

If you are creating a dependency property that exists on a derived class of [FrameworkElement](#), you can use the more specialized metadata class [FrameworkPropertyMetadata](#) rather than the base [PropertyMetadata](#) class. The constructor for the [FrameworkPropertyMetadata](#) class has several signatures where you can specify various metadata characteristics in combination. If you want to specify the default value only, use the

signature that takes a single parameter of type [Object](#). Pass that object parameter as a type-specific default value for your property (the default value provided must be the type you provided as the `propertyType` parameter in the [Register](#) call).

For [FrameworkPropertyMetadata](#), you can also specify metadata option flags for your property. These flags are converted into discrete properties on the property metadata after registration and are used to communicate certain conditionals to other processes such as the layout engine.

#### Setting Appropriate Metadata Flags

- If your property (or changes in its value) affects the user interface (UI), and in particular affects how the layout system should size or render your element in a page, set one or more of the following flags: [AffectsMeasure](#), [AffectsArrange](#), [AffectsRender](#).
  - [AffectsMeasure](#) indicates that a change to this property requires a change to UI rendering where the containing object might require more or less space within the parent. For example, a "Width" property should have this flag set.
  - [AffectsArrange](#) indicates that a change to this property requires a change to UI rendering that typically does not require a change in the dedicated space, but does indicate that the positioning within the space has changed. For example, an "Alignment" property should have this flag set.
  - [AffectsRender](#) indicates that some other change has occurred that will not affect layout and measure, but does require another render. An example would be a property that changes a color of an existing element, such as "Background".
  - These flags are often used as a protocol in metadata for your own override implementations of property system or layout callbacks. For instance, you might have an [OnPropertyChanged](#) callback that will call [InvalidateArrange](#) if any property of the instance reports a value change and has [AffectsArrange](#) as `true` in its metadata.
- Some properties may affect the rendering characteristics of the containing parent element, in ways above and beyond the changes in required size mentioned above. An example is the [MinOrphanLines](#) property used in the flow document model, where changes to that property can change the overall rendering of the flow document that contains the paragraph. Use [AffectsParentArrange](#) or [AffectsParentMeasure](#) to identify similar cases in your own properties.
- By default, dependency properties support data binding. You can deliberately disable data binding, for cases where there is no realistic scenario for data binding, or where performance in data binding for a large object is recognized as a problem.
- By default, data binding [Mode](#) for dependency properties defaults to [OneWay](#). You can always change the binding to be [TwoWay](#) per binding instance; for details, see [Specify the Direction of the Binding](#). But as the dependency property author, you can choose to make the property use [TwoWay](#) binding mode by default. An example of an existing dependency property is [MenuItem.Is\\_submenuOpen](#); the scenario for this property is that the [Is\\_submenuOpen](#) setting logic and the compositing of [MenuItem](#) interact with the default theme style. The [Is\\_submenuOpen](#) property logic uses data binding natively to maintain the state of the property in accordance to other state properties and method calls. Another example property that binds [TwoWay](#) by default is [TextBox.Text](#).
- You can also enable property inheritance in a custom dependency property by setting the [Inherits](#) flag. Property inheritance is useful for a scenario where parent elements and child elements have a property in common, and it makes sense for the child elements to have that particular property value set to the same value as the parent set it. An example inheritable property is [DataContext](#), which is used for binding operations to enable the important master-detail scenario for data presentation. By making [DataContext](#) inheritable, any child elements inherit that data context also. Because of property value inheritance, you can specify a data context at the page or application root, and do not need to respecify

it for bindings in all possible child elements. [DataContext](#) is also a good example to illustrate that inheritance overrides the default value, but it can always be set locally on any particular child element; for details, see [Use the Master-Detail Pattern with Hierarchical Data](#). Property value inheritance does have a possible performance cost, and thus should be used sparingly; for details, see [Property Value Inheritance](#).

- Set the [Journal](#) flag to indicate if your dependency property should be detected or used by navigation journaling services. An example is the [SelectedIndex](#) property; any item selected in a selection control should be persisted when the journaling history is navigated.

## Read-Only Dependency Properties

You can define a dependency property that is read-only. However, the scenarios for why you might define your property as read-only are somewhat different, as is the procedure for registering them with the property system and exposing the identifier. For more information, see [Read-Only Dependency Properties](#).

## Collection-Type Dependency Properties

Collection-type dependency properties have some additional implementation issues to consider. For details, see [Collection-Type Dependency Properties](#).

## Dependency Property Security Considerations

Dependency properties should be declared as public properties. Dependency property identifier fields should be declared as public static fields. Even if you attempt to declare other access levels (such as protected), a dependency property can always be accessed through the identifier in combination with the property system APIs. Even a protected identifier field is potentially accessible because of metadata reporting or value determination APIs that are part of the property system, such as [LocalValueEnumerator](#). For more information, see [Dependency Property Security](#).

## Dependency Properties and Class Constructors

There is a general principle in managed code programming (often enforced by code analysis tools such as FxCop) that class constructors should not call virtual methods. This is because constructors can be called as base initialization of a derived class constructor, and entering the virtual method through the constructor might occur at an incomplete initialization state of the object instance being constructed. When you derive from any class that already derives from [DependencyObject](#), you should be aware that the property system itself calls and exposes virtual methods internally. These virtual methods are part of the WPF property system services. Overriding the methods enables derived classes to participate in value determination. To avoid potential issues with runtime initialization, you should not set dependency property values within constructors of classes, unless you follow a very specific constructor pattern. For details, see [Safe Constructor Patterns for DependencyObjects](#).

## See also

- [Dependency Properties Overview](#)
- [Dependency Property Metadata](#)
- [Control Authoring Overview](#)
- [Collection-Type Dependency Properties](#)
- [Dependency Property Security](#)
- [XAML Loading and Dependency Properties](#)
- [Safe Constructor Patterns for DependencyObjects](#)

# Dependency Property Metadata

8/22/2019 • 10 minutes to read • [Edit Online](#)

The Windows Presentation Foundation (WPF) property system includes a metadata reporting system that goes beyond what can be reported about a property through reflection or general common language runtime (CLR) characteristics. Metadata for a dependency property can also be assigned uniquely by the class that defines a dependency property, can be changed when the dependency property is added to a different class, and can be specifically overridden by all derived classes that inherit the dependency property from the defining base class.

## Prerequisites

This topic assumes that you understand dependency properties from the perspective of a consumer of existing dependency properties on Windows Presentation Foundation (WPF) classes, and have read the [Dependency Properties Overview](#). In order to follow the examples in this topic, you should also understand XAML and know how to write WPF applications.

## How Dependency Property Metadata is Used

Dependency property metadata exists as an object that can be queried to examine the characteristics of a dependency property. This metadata is also accessed frequently by the property system as it processes any given dependency property. The metadata object for a dependency property can contain the following types of information:

- Default value for the dependency property, if no other value can be determined for the dependency property by local value, style, inheritance, etc. For a thorough discussion of how default values participate in the precedence used by the property system when assigning values for dependency properties, see [Dependency Property Value Precedence](#).
- References to callback implementations that affect coercion or change-notification behaviors on a per-owner-type basis. Note that these callbacks are often defined with a nonpublic access level, so obtaining the actual references from metadata is generally not possible unless the references are within your permitted access scope. For more information on dependency property callbacks, see [Dependency Property Callbacks and Validation](#).
- If the dependency property in question is considered to be a WPF framework-level property, the metadata might contain WPF framework-level dependency property characteristics, which report information and state for services such as the WPF framework-level layout engine and property inheritance logic. For more information on this aspect of dependency property metadata, see [Framework Property Metadata](#).

## Metadata APIs

The type that reports most of the metadata information used by the property system is the [PropertyMetadata](#) class. Metadata instances are optionally specified when dependency properties are registered with the property system, and can be specified again for additional types that either add themselves as owners or override metadata they inherit from the base class dependency property definition. (For cases where a property registration does not specify metadata, a default [PropertyMetadata](#) is created with default values for that class.) The registered metadata is returned as [PropertyMetadata](#) when you call the various [GetMetadata](#) overloads that get metadata from a dependency property on a [DependencyObject](#) instance.

The [PropertyMetadata](#) class is then derived from to provide more specific metadata for architectural divisions

such as the WPF framework-level classes. [UIPropertyMetadata](#) adds animation reporting, and [FrameworkPropertyMetadata](#) provides the WPF framework-level properties mentioned in the previous section. When dependency properties are registered, they can be registered with these [PropertyMetadata](#) derived classes. When the metadata is examined, the base [PropertyMetadata](#) type can potentially be cast to the derived classes so that you can examine the more specific properties.

#### NOTE

The property characteristics that can be specified in [FrameworkPropertyMetadata](#) are sometimes referred to in this documentation as "flags". When you create new metadata instances for use in dependency property registrations or metadata overrides, you specify these values using the flagwise enumeration [FrameworkPropertyMetadataOptions](#) and then you supply possibly concatenated values of the enumeration to the [FrameworkPropertyMetadata](#) constructor. However, once constructed, these option characteristics are exposed within a [FrameworkPropertyMetadata](#) as a series of Boolean properties rather than the constructing enumeration value. The Boolean properties enable you to check each conditional, rather than requiring you to apply a mask to a flagwise enumeration value to get the information you are interested in. The constructor uses the concatenated [FrameworkPropertyMetadataOptions](#) in order to keep the length of the constructor signature reasonable, whereas the actual constructed metadata exposes the discrete properties to make querying the metadata more intuitive.

## When to Override Metadata, When to Derive a Class

The WPF property system has established capabilities for changing some characteristics of dependency properties without requiring them to be entirely re-implemented. This is accomplished by constructing a different instance of property metadata for the dependency property as it exists on a particular type. Note that most existing dependency properties are not virtual properties, so strictly speaking "re-implementing" them on inherited classes could only be accomplished by shadowing the existing member.

If the scenario you are trying to enable for a dependency property on a type cannot be accomplished by modifying characteristics of existing dependency properties, it might then be necessary to create a derived class, and then to declare a custom dependency property on your derived class. A custom dependency property behaves identically to dependency properties defined by the WPF APIs. For more details about custom dependency properties, see [Custom Dependency Properties](#).

One notable characteristic of a dependency property that you cannot override is its value type. If you are inheriting a dependency property that has the approximate behavior you require, but you require a different type for it, you will have to implement a custom dependency property and perhaps link the properties through type conversion or other implementation on your custom class. Also, you cannot replace an existing [ValidateValueCallback](#), because this callback exists in the registration field itself and not within its metadata.

## Scenarios for Changing Existing Metadata

If you are working with metadata of an existing dependency property, one common scenario for changing dependency property metadata is to change the default value. Changing or adding property system callbacks is a more advanced scenario. You might want to do this if your implementation of a derived class has different interrelationships between dependency properties. One of the conditionals of having a programming model that supports both code and declarative usage is that properties must enable being set in any order. Thus any dependent properties need to be set just-in-time without context and cannot rely on knowing a setting order such as might be found in a constructor. For more information on this aspect of the property system, see [Dependency Property Callbacks and Validation](#). Note that validation callbacks are not part of the metadata; they are part of the dependency property identifier. Therefore, validation callbacks cannot be changed by overriding the metadata.

In some cases you might also want to alter the WPF framework-level property metadata options on existing dependency properties. These options communicate certain known conditionals about WPF framework-level

properties to other WPF framework-level processes such as the layout system. Setting the options is generally done only when registering a new dependency property, but it is also possible to change the WPF framework-level property metadata as part of a [OverrideMetadata](#) or [AddOwner](#) call. For the specific values to use and more information, see [Framework Property Metadata](#). For more information that is pertinent to how these options should be set for a newly registered dependency property, see [Custom Dependency Properties](#).

## Overriding Metadata

The purpose of overriding metadata is primarily so that you have the opportunity to change the various metadata-derived behaviors that are applied to the dependency property as it exists on your type. The reasons for this are explained in more detail in the [Metadata](#) section. For more information including some code examples, see [Override Metadata for a Dependency Property](#).

Property metadata can be supplied for a dependency property during the registration call ([Register](#)). However, in many cases, you might want to provide type-specific metadata for your class when it inherits that dependency property. You can do this by calling the [OverrideMetadata](#) method. For an example from the WPF APIs, the [FrameworkElement](#) class is the type that first registers the [Focusable](#) dependency property. But the [Control](#) class overrides metadata for the dependency property to provide its own initial default value, changing it from `false` to `true`, and otherwise re-uses the original [Focusable](#) implementation.

When you override metadata, the different metadata characteristics are either merged or replaced.

- [PropertyChangedCallback](#) is merged. If you add a new [PropertyChangedCallback](#), that callback is stored in the metadata. If you do not specify a [PropertyChangedCallback](#) in the override, the value of [PropertyChangedCallback](#) is promoted as a reference from the nearest ancestor that specified it in metadata.
- The actual property system behavior for [PropertyChangedCallback](#) is that implementations for all metadata owners in the hierarchy are retained and added to a table, with order of execution by the property system being that the most derived class's callbacks are invoked first.
- [DefaultValue](#) is replaced. If you do not specify a [DefaultValue](#) in the override, the value of [DefaultValue](#) comes from the nearest ancestor that specified it in metadata.
- [CoerceValueCallback](#) implementations are replaced. If you add a new [CoerceValueCallback](#), that callback is stored in the metadata. If you do not specify a [CoerceValueCallback](#) in the override, the value of [CoerceValueCallback](#) is promoted as a reference from the nearest ancestor that specified it in metadata.
- The property system behavior is that only the [CoerceValueCallback](#) in the immediate metadata is invoked. No references to other [CoerceValueCallback](#) implementations in the hierarchy are retained.

This behavior is implemented by [Merge](#), and can be overridden on derived metadata classes.

## Overriding Attached Property Metadata

In WPF, attached properties are implemented as dependency properties. This means that they also have property metadata, which individual classes can override. The scoping considerations for an attached property in WPF are generally that any [DependencyObject](#) can have an attached property set on them. Therefore, any [DependencyObject](#) derived class can override the metadata for any attached property, as it might be set on an instance of the class. You can override default values, callbacks, or WPF framework-level characteristic-reporting properties. If the attached property is set on an instance of your class, those override property metadata characteristics apply. For instance, you can override the default value, such that your override value is reported as the value of the attached property on instances of your class, whenever the property is not otherwise set.

### NOTE

The [Inherits](#) property is not relevant for attached properties.

## Adding a Class as an Owner of an Existing Dependency Property

A class can add itself as an owner of a dependency property that has already been registered, by using the [AddOwner](#) method. This enables the class to use a dependency property that was originally registered for a different type. The adding class is typically not a derived class of the type that first registered that dependency property as owner. Effectively, this allows your class and its derived classes to "inherit" a dependency property implementation without the original owner class and the adding class being in the same true class hierarchy. In addition, the adding class (and all derived classes as well) can then provide type-specific metadata for the original dependency property.

As well as adding itself as owner through the property system utility methods, the adding class should declare additional public members on itself in order to make the dependency property] a full participant in the property system with exposure to both code and markup. A class that adds an existing dependency property has the same responsibilities as far as exposing the object model for that dependency property as does a class that defines a new custom dependency property. The first such member to expose is a dependency property identifier field. This field should be a `public static readonly` field of type [DependencyProperty](#), which is assigned to the return value of the [AddOwner](#) call. The second member to define is the common language runtime (CLR) "wrapper" property. The wrapper makes it much more convenient to manipulate your dependency property in code (you avoid calls to [SetValue](#) each time, and can make that call only once in the wrapper itself). The wrapper is implemented identically to how it would be implemented if you were registering a custom dependency property. For more information about implementing a dependency property, see [Custom Dependency Properties](#) and [Add an Owner Type for a Dependency Property](#).

### AddOwner and Attached Properties

You can call [AddOwner](#) for a dependency property that is defined as an attached property by the owner class. Generally the reason for doing this is to expose the previously attached property as a non-attached dependency property. You then will expose the [AddOwner](#) return value as a `public static readonly` field for use as the dependency property identifier, and will define appropriate "wrapper" properties so that the property appears in the members table and supports a non-attached property usage in your class.

## See also

- [PropertyMetadata](#)
- [DependencyObject](#)
- [DependencyProperty](#)
- [GetMetadata](#)
- [Dependency Properties Overview](#)
- [Framework Property Metadata](#)

# Dependency Property Callbacks and Validation

8/22/2019 • 7 minutes to read • [Edit Online](#)

This topic describes how to create dependency properties using alternative custom implementations for property-related features such as validation determination, callbacks that are invoked whenever the property's effective value is changed, and overriding possible outside influences on value determination. This topic also discusses scenarios where expanding on the default property system behaviors by using these techniques is appropriate.

## Prerequisites

This topic assumes that you understand the basic scenarios of implementing a dependency property, and how metadata is applied to a custom dependency property. See [Custom Dependency Properties](#) and [Dependency Property Metadata](#) for context.

## Validation Callbacks

Validation callbacks can be assigned to a dependency property when you first register it. The validation callback is not part of property metadata; it is a direct input of the [Register](#) method. Therefore, once a validation callback is created for a dependency property, it cannot be overridden by a new implementation.

```
public static readonly DependencyProperty CurrentReadingProperty = DependencyProperty.Register(
    "CurrentReading",
    typeof(double),
    typeof(Gauge),
    new FrameworkPropertyMetadata(
        Double.NaN,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(OnCurrentReadingChanged),
        new CoerceValueCallback(CoerceCurrentReading)
    ),
    new ValidateValueCallback(IsValidReading)
);
public double CurrentReading
{
    get { return (double)GetValue(CurrentReadingProperty); }
    set { SetValue(CurrentReadingProperty, value); }
}
```

```

Public Shared ReadOnly CurrentReadingProperty As DependencyProperty =
    DependencyProperty.Register("CurrentReading",
        GetType(Double), GetType(Gauge),
        New FrameworkPropertyMetadata(Double.NaN,
            FrameworkPropertyMetadataOptions.AffectsMeasure,
            New PropertyChangedCallback(AddressOf OnCurrentReadingChanged),
            New CoerceValueCallback(AddressOf CoerceCurrentReading)),
        New ValidateValueCallback(AddressOf IsValidReading))

```

```

Public Property CurrentReading() As Double
    Get
        Return CDblGetValue(CurrentReadingProperty))
    End Get
    Set(ByVal value As Double)
        SetValue(CurrentReadingProperty, value)
    End Set
End Property

```

The callbacks are implemented such that they are provided an object value. They return `true` if the provided value is valid for the property; otherwise, they return `false`. It is assumed that the property is of the correct type per the type registered with the property system, so checking type within the callbacks is not ordinarily done. The callbacks are used by the property system in a variety of different operations. This includes the initial type initialization by default value, programmatic change by invoking `SetValue`, or attempts to override metadata with new default value provided. If the validation callback is invoked by any of these operations, and returns `false`, then an exception will be raised. Application writers must be prepared to handle these exceptions. A common use of validation callbacks is validating enumeration values, or constraining values of integers or doubles when the property sets measurements that must be zero or greater.

Validation callbacks specifically are intended to be class validators, not instance validators. The parameters of the callback do not communicate a specific `DependencyObject` on which the properties to validate are set. Therefore the validation callbacks are not useful for enforcing the possible "dependencies" that might influence a property value, where the instance-specific value of a property is dependent on factors such as instance-specific values of other properties, or run-time state.

The following is example code for a very simple validation callback scenario: validating that a property that is typed as the `Double` primitive is not `PositiveInfinity` or `NegativeInfinity`.

```

public static bool IsValidReading(object value)
{
    Double v = (Double)value;
    return (!v.Equals(Double.NegativeInfinity) && !v.Equals(Double.PositiveInfinity));
}

```

```

Public Shared Function IsValidReading(ByVal value As Object) As Boolean
    Dim v As Double = CType(value, Double)
    Return ((Not v.Equals(Double.NegativeInfinity)) AndAlso
            (Not v.Equals(Double.PositiveInfinity)))
End Function

```

## Coerce Value Callbacks and Property Changed Events

Coerce value callbacks do pass the specific `DependencyObject` instance for properties, as do `PropertyChangedCallback` implementations that are invoked by the property system whenever the value of a dependency property changes. Using these two callbacks in combination, you can create a series of properties on elements where changes in one property will force a coercion or reevaluation of another property.

A typical scenario for using a linkage of dependency properties is when you have a user interface driven property where the element holds one property each for the minimum and maximum value, and a third property for the actual or current value. Here, if the maximum was adjusted in such a way that the current value exceeded the new maximum, you would want to coerce the current value to be no greater than the new maximum, and a similar relationship for minimum to current.

The following is very brief example code for just one of the three dependency properties that illustrate this relationship. The example shows how the `CurrentReading` property of a Min/Max/Current set of related \*Reading properties is registered. It uses the validation as shown in the previous section.

```
public static readonly DependencyProperty CurrentReadingProperty = DependencyProperty.Register(
    "CurrentReading",
    typeof(double),
    typeof(Gauge),
    new FrameworkPropertyMetadata(
        Double.NaN,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(OnCurrentReadingChanged),
        new CoerceValueCallback(CoerceCurrentReading)
    ),
    new ValidateValueCallback(IsValidReading)
);
public double CurrentReading
{
    get { return (double)GetValue(CurrentReadingProperty); }
    set { SetValue(CurrentReadingProperty, value); }
}
```

```
Public Shared ReadOnly CurrentReadingProperty As DependencyProperty =
    DependencyProperty.Register("CurrentReading",
        GetType(Double), GetType(Gauge),
        New FrameworkPropertyMetadata(Double.NaN,
            FrameworkPropertyMetadataOptions.AffectsMeasure,
            New PropertyChangedCallback(AddressOf OnCurrentReadingChanged),
            New CoerceValueCallback(AddressOf CoerceCurrentReading)),
        New ValidateValueCallback(AddressOf IsValidReading))

Public Property CurrentReading() As Double
    Get
        Return CDbl(GetValue(CurrentReadingProperty))
    End Get
    Set(ByVal value As Double)
        SetValue(CurrentReadingProperty, value)
    End Set
End Property
```

The property changed callback for Current is used to forward the change to other dependent properties, by explicitly invoking the coerce value callbacks that are registered for those other properties:

```
private static void OnCurrentReadingChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    d.CoerceValue(MinReadingProperty);
    d.CoerceValue(MaxReadingProperty);
}
```

```

Private Shared Sub OnCurrentReadingChanged(ByVal d As DependencyObject, ByVal e As
DependencyPropertyChangedEventArgs)
    d.CoerceValue(MinReadingProperty)
    d.CoerceValue(MaxReadingProperty)
End Sub

```

The coerce value callback checks the values of properties that the current property is potentially dependent upon, and coerces the current value if necessary:

```

private static object CoerceCurrentReading(DependencyObject d, object value)
{
    Gauge g = (Gauge)d;
    double current = (double)value;
    if (current < g.MinReading) current = g.MinReading;
    if (current > g.MaxReading) current = g.MaxReading;
    return current;
}

```

```

Private Shared Function CoerceCurrentReading(ByVal d As DependencyObject, ByVal value As Object) As Object
    Dim g As Gauge = CType(d, Gauge)
    Dim current As Double = CDbl(value)
    If current < g.MinReading Then
        current = g.MinReading
    End If
    If current > g.MaxReading Then
        current = g.MaxReading
    End If
    Return current
End Function

```

#### **NOTE**

Default values of properties are not coerced. A property value equal to the default value might occur if a property value still has its initial default, or through clearing other values with [ClearValue](#).

The coerce value and property changed callbacks are part of property metadata. Therefore, you can change the callbacks for a particular dependency property as it exists on a type that you derive from the type that owns the dependency property, by overriding the metadata for that property on your type.

## Advanced Coercion and Callback Scenarios

### Constraints and Desired Values

The [CoerceValueCallback](#) callbacks will be used by the property system to coerce a value in accordance to the logic you declare, but a coerced value of a locally set property will still retain a "desired value" internally. If the constraints are based on other property values that may change dynamically during the application lifetime, the coercion constraints are changed dynamically also, and the constrained property can change its value to get as close to the desired value as possible given the new constraints. The value will become the desired value if all constraints are lifted. You can potentially introduce some fairly complicated dependency scenarios if you have multiple properties that are dependent on one another in a circular manner. For instance, in the Min/Max/Current scenario, you could choose to have Minimum and Maximum be user settable. If so, you might need to coerce that Maximum is always greater than Minimum and vice versa. But if that coercion is active, and Maximum coerces to Minimum, it leaves Current in an unsettable state, because it is dependent on both and is constrained to the range between the values, which is zero. Then, if Maximum or Minimum are adjusted, Current will seem to "follow" one of the values, because the desired value of Current is still stored and is attempting to reach the desired value as

the constraints are loosened.

There is nothing technically wrong with complex dependencies, but they can be a slight performance detriment if they require large numbers of reevaluations, and can also be confusing to users if they affect the UI directly. Be careful with property changed and coerce value callbacks and make sure that the coercion being attempted can be treated as unambiguously as possible, and does not "overconstrain".

### Using `CoerceValue` to Cancel Value Changes

The property system will treat any `CoerceValueCallback` that returns the value `UnsetValue` as a special case. This special case means that the property change that resulted in the `CoerceValueCallback` being called should be rejected by the property system, and that the property system should instead report whatever previous value the property had. This mechanism can be useful to check that changes to a property that were initiated asynchronously are still valid for the current object state, and suppress the changes if not. Another possible scenario is that you can selectively suppress a value depending on which component of property value determination is responsible for the value being reported. To do this, you can use the `DependencyProperty` passed in the callback and the property identifier as input for `GetValueSource`, and then process the `ValueSource`.

## See also

- [Dependency Properties Overview](#)
- [Dependency Property Metadata](#)
- [Custom Dependency Properties](#)

# Framework Property Metadata

11/3/2019 • 5 minutes to read • [Edit Online](#)

Framework property metadata options are reported for the properties of object elements considered to be at the WPF framework level in the Windows Presentation Foundation (WPF) architecture. In general the WPF framework-level designation entails that features such as rendering, data binding, and property system refinements are handled by the WPF presentation APIs and executables. Framework property metadata is queried by these systems to determine feature-specific characteristics of particular element properties.

## Prerequisites

This topic assumes that you understand dependency properties from the perspective of a consumer of existing dependency properties on Windows Presentation Foundation (WPF) classes, and have read the [Dependency Properties Overview](#). You should also have read [Dependency Property Metadata](#).

## What Is Communicated by Framework Property Metadata

Framework property metadata can be divided into the following categories:

- Reporting layout properties that affect an element ([AffectsArrange](#), [AffectsMeasure](#), [AffectsRender](#)). You might set these flags in metadata if the property affects those respective aspects, and you are also implementing the [MeasureOverride / ArrangeOverride](#) methods in your class to supply specific rendering behavior and information to the layout system. Typically, such an implementation would check for property invalidations in dependency properties where any of these layout properties were true in the property metadata, and only those invalidations would necessitate requesting a new layout pass.
- Reporting layout properties that affect the parent element of an element ([AffectsParentArrange](#), [AffectsParentMeasure](#)). Some examples where these flags are set by default are [FixedPage.Left](#) and [Paragraph.KeepWithNext](#).
- **Inherits**. By default, dependency properties do not inherit values. [OverridesInheritanceBehavior](#) allows the pathway of inheritance to also travel into a visual tree, which is necessary for some control compositing scenarios.

### NOTE

The term "inherits" in the context of property values means something specific for dependency properties; it means that child elements can inherit the actual dependency property value from parent elements because of a WPF framework-level capability of the WPF property system. It has nothing to do directly with managed code type and members inheritance through derived types. For details, see [Property Value Inheritance](#).

- Reporting data binding characteristics ( [IsNotDataBindable](#),  [BindsTwoWayByDefault](#)). By default, dependency properties in the framework support data binding, with a one-way binding behavior. You might disable data binding if there were no scenario for it whatsoever (because they are intended to be flexible and extensible, there aren't many examples of such properties in the default WPF APIs). You might set binding to have a two-way default for properties that tie together a control's behaviors amongst its component pieces ( [IsSubMenuOpen](#) is an example) or where two-way binding is the common and expected scenario for users ( [Text](#) is an example). Changing the data binding-related metadata only influences the default; on a per-binding basis that default can always be changed. For details on the binding modes and binding in general, see [Data Binding Overview](#).

- Reporting whether properties should be journaled by applications or services that support journaling ([Journal](#)). For general elements, journaling is not enabled by default, but it is selectively enabled for certain user input controls. This property is intended to be read by journaling services including the WPF implementation of journaling, and is typically set on user controls such as user selections within lists that should be persisted across navigation steps. For information about the journal, see [Navigation Overview](#).

## Reading FrameworkPropertyMetadata

Each of the properties linked above are the specific properties that the [FrameworkPropertyMetadata](#) adds to its immediate base class [UIPropertyMetadata](#). Each of these properties will be `false` by default. A metadata request for a property where knowing the value of these properties is important should attempt to cast the returned metadata to [FrameworkPropertyMetadata](#), and then check the values of the individual properties as needed.

## Specifying Metadata

When you create a new metadata instance for purposes of applying metadata to a new dependency property registration, you have the choice of which metadata class to use: the base [PropertyMetadata](#) or some derived class such as [FrameworkPropertyMetadata](#). In general, you should use [FrameworkPropertyMetadata](#), particularly if your property has any interaction with property system and WPF functions such as layout and data binding. Another option for more sophisticated scenarios is to derive from [FrameworkPropertyMetadata](#) to create your own metadata reporting class with extra information carried in its members. Or you might use [PropertyMetadata](#) or [UIPropertyMetadata](#) to communicate the degree of support for features of your implementation.

For existing properties ([AddOwner](#) or [OverrideMetadata](#) call), you should always override with the metadata type used by the original registration.

If you are creating a [FrameworkPropertyMetadata](#) instance, there are two ways to populate that metadata with values for the specific properties that communicate the framework property characteristics:

1. Use the [FrameworkPropertyMetadata](#) constructor signature that allows a `flags` parameter. This parameter should be filled with all desired combined values of the [FrameworkPropertyMetadataOptions](#) enumeration flags.
2. Use one of the signatures without a `flags` parameter, and then set each reporting Boolean property on [FrameworkPropertyMetadata](#) to `true` for each desired characteristic change. If you do this, you must set these properties before any elements with this dependency property are constructed; the Boolean properties are read-write in order to allow this behavior of avoiding the `flags` parameter and still populate the metadata, but the metadata must become effectively sealed before property use. Thus, attempting to set the properties after metadata is requested will be an invalid operation.

## Framework Property Metadata Merge Behavior

When you override framework property metadata, the different metadata characteristics are either merged or replaced.

- [PropertyChangedCallback](#) is merged. If you add a new [PropertyChangedCallback](#), that callback is stored in the metadata. If you do not specify a [PropertyChangedCallback](#) in the override, the value of [PropertyChangedCallback](#) is promoted as a reference from the nearest ancestor that specified it in metadata.
- The actual property system behavior for [PropertyChangedCallback](#) is that implementations for all metadata owners in the hierarchy are retained and added to a table, with order of execution by the property system being that the callbacks of the most deeply derived class are invoked first. Inherited callbacks run only once, counting as being owned by the class that placed them in metadata.

- `DefaultValue` is replaced. If you do not specify a `PropertyChangedCallback` in the override, the value of `DefaultValue` comes from the nearest ancestor that specified it in metadata.
- `CoerceValueCallback` implementations are replaced. If you add a new `CoerceValueCallback`, that callback is stored in the metadata. If you do not specify a `CoerceValueCallback` in the override, the value of `CoerceValueCallback` is promoted as a reference from the nearest ancestor that specified it in metadata.
- The property system behavior is that only the `CoerceValueCallback` in the immediate metadata is invoked. No references to other `CoerceValueCallback` implementations in the hierarchy are retained.
- The flags of `FrameworkPropertyMetadataOptions` enumeration are combined as a bitwise OR operation. If you specify `FrameworkPropertyMetadataOptions`, the original options are not overwritten. To change an option, set the corresponding property on `FrameworkPropertyMetadata`. For example, if the original `FrameworkPropertyMetadata` object sets the `FrameworkPropertyMetadataOptions.NotDataBindable` flag, you can change that by setting `FrameworkPropertyMetadata.IsNotDataBindable` to `false`.

This behavior is implemented by `Merge`, and can be overridden on derived metadata classes.

## See also

- [GetMetadata](#)
- [Dependency Property Metadata](#)
- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)

# Dependency Property Value Precedence

11/7/2019 • 14 minutes to read • [Edit Online](#)

This topic explains how the workings of the Windows Presentation Foundation (WPF) property system can affect the value of a dependency property, and describes the precedence by which aspects of the property system apply to the effective value of a property.

## Prerequisites

This topic assumes that you understand dependency properties from the perspective of a consumer of existing dependency properties on WPF classes, and have read [Dependency Properties Overview](#). To follow the examples in this topic, you should also understand Extensible Application Markup Language (XAML) and know how to write WPF applications.

## The WPF Property System

The WPF property system offers a powerful way to have the value of dependency properties be determined by a variety of factors, enabling features such as real-time property validation, late binding, and notifying related properties of changes to values for other properties. The exact order and logic that is used to determine dependency property values is reasonably complex. Knowing this order will help you avoid unnecessary property setting, and might also clear up confusion over exactly why some attempt to influence or anticipate a dependency property value did not end up resulting in the value you expected.

## Dependency Properties Might Be "Set" in Multiple Places

The following is example XAML where the same property ([Background](#)) has three different "set" operations that might influence the value.

```
<Button Background="Red">
  <Button.Style>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Green"/>
      <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
          <Setter Property="Background" Value="Blue" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </Button.Style>
  Click
</Button>
```

Here, which color do you expect will apply—red, green, or blue?

With the exception of animated values and coercion, local property sets are set at the highest precedence. If you set a value locally you can expect that the value will be honored, even above any styles or control templates. Here in the example, [Background](#) is set to Red locally. Therefore, the style defined in this scope, even though it is an implicit style that would otherwise apply to all elements of that type in that scope, is not the highest precedence for giving the [Background](#) property its value. If you removed the local value of Red from that Button instance, then the style would have precedence and the button would obtain the Background value from the style. Within the style, triggers take precedence, so the button will be blue if the mouse is over it, and green otherwise.

# Dependency Property Setting Precedence List

The following is the definitive order that the property system uses when assigning the run-time values of dependency properties. Highest precedence is listed first. This list expands on some of the generalizations made in the [Dependency Properties Overview](#).

1. **Property system coercion.** For details on coercion, see [Coercion, Animation, and Base Value](#) later in this topic.
2. **Active animations, or animations with a Hold behavior.** In order to have any practical effect, an animation of a property must be able to have precedence over the base (unanimated) value, even if that value was set locally. For details, see [Coercion, Animation, and Base Value](#) later in this topic.
3. **Local value.** A local value might be set through the convenience of the "wrapper" property, which also equates to setting as an attribute or property element in XAML, or by a call to the [SetValue](#) API using a property of a specific instance. If you set a local value by using a binding or a resource, these each act in the precedence as if a direct value was set.
4. **TemplatedParent template properties.** An element has a [TemplatedParent](#) if it was created as part of a template (a [ControlTemplate](#) or [DataTemplate](#)). For details on when this applies, see [TemplatedParent](#) later in this topic. Within the template, the following precedence applies:
  - a. Triggers from the [TemplatedParent](#) template.
  - b. Property sets (typically through XAML attributes) in the [TemplatedParent](#) template.
5. **Implicit style.** Applies only to the `Style` property. The `Style` property is filled by any style resource with a key that matches the type of that element. That style resource must exist either in the page or the application; lookup for an implicit style resource does not proceed into the themes.
6. **Style triggers.** The triggers within styles from page or application (these styles might be either explicit or implicit styles, but not from the default styles, which have lower precedence).
7. **Template triggers.** Any trigger from a template within a style, or a directly applied template.
8. **Style setters.** Values from a [Setter](#) within styles from page or application.
9. **Default (theme) style.** For details on when this applies, and how theme styles relate to the templates within theme styles, see [Default \(Theme\) Styles](#) later in this topic. Within a default style, the following order of precedence applies:
  - a. Active triggers in the theme style.
  - b. Setters in the theme style.
10. **Inheritance.** A few dependency properties inherit their values from parent element to child elements, such that they need not be set specifically on each element throughout an application. For details see [Property Value Inheritance](#).
11. **Default value from dependency property metadata.** Any given dependency property may have a default value as established by the property system registration of that particular property. Also, derived classes that inherit a dependency property have the option to override that metadata (including the default value) on a per-type basis. See [Dependency Property Metadata](#) for more information. Because inheritance is checked before default value, for an inherited property, a parent element default value takes precedence over a child element. Consequently, if an inheritable property is not set anywhere, the default value as specified on the root or parent is used instead of the child element default value.

## TemplatedParent

TemplatedParent as a precedence item does not apply to any property of an element that you declare directly in standard application markup. The TemplatedParent concept exists only for child items within a visual tree that come into existence through the application of the template. When the property system searches the [TemplatedParent](#) template for a value, it is searching the template that created that element. The property values from the [TemplatedParent](#) template generally act as if they were set as a local value on the child element, but this lesser precedence versus the local value exists because the templates are potentially shared. For details, see [TemplatedParent](#).

## The Style Property

The order of lookup described earlier applies to all possible dependency properties except one: the [Style](#) property. The [Style](#) property is unique in that it cannot itself be styled, so the precedence items 5 through 8 do not apply. Also, either animating or coercing [Style](#) is not recommended (and animating [Style](#) would require a custom animation class). This leaves three ways that the [Style](#) property might be set:

- **Explicit style.** The [Style](#) property is set directly. In most scenarios, the style is not defined inline, but instead is referenced as a resource, by explicit key. In this case the [Style](#) property itself acts as if it were a local value, precedence item 3.
- **Implicit style.** The [Style](#) property is not set directly. However, the [Style](#) exists at some level in the resource lookup sequence (page, application) and is keyed using a resource key that matches the type the style is to be applied to. In this case, the [Style](#) property itself acts by a precedence identified in the sequence as item 5. This condition can be detected by using [DependencyPropertyHelper](#) against the [Style](#) property and looking for [ImplicitStyleReference](#) in the results.
- **Default style**, also known as **theme style**. The [Style](#) property is not set directly, and in fact will read as `null` up until run time. In this case, the style comes from the run-time theme evaluation that is part of the WPF presentation engine.

For implicit styles not in themes, the type must match exactly - a `MyButton` `Button`-derived class will not implicitly use a style for `Button`.

## Default (Theme) Styles

Every control that ships with WPF has a default style. That default style potentially varies by theme, which is why this default style is sometimes referred to as a theme style.

The most important information that is found within a default style for a control is its control template, which exists in the theme style as a setter for its [Template](#) property. If there were no template from default styles, a control without a custom template as part of a custom style would have no visual appearance at all. The template from the default style gives the visual appearance of each control a basic structure, and also defines the connections between properties defined in the visual tree of the template and the corresponding control class. Each control exposes a set of properties that can influence the visual appearance of the control without completely replacing the template. For example, consider the default visual appearance of a [Thumb](#) control, which is a component of a [ScrollBar](#).

A [Thumb](#) has certain customizable properties. The default template of a [Thumb](#) creates a basic structure / visual tree with several nested [Border](#) components to create a bevel look. If a property that is part of the template is intended to be exposed for customization by the [Thumb](#) class, then that property must be exposed by a [TemplateBinding](#), within the template. In the case of [Thumb](#), various properties of these borders share a template binding to properties such as [Background](#) or [BorderThickness](#). But certain other properties or visual arrangements are hard-coded into the control template or are bound to values that come directly from the theme, and cannot be changed short of replacing the entire template. Generally, if a property comes from a templated parent and is not exposed by a template binding, it cannot be adjusted by styles because there is no easy way to target it. But that property could still be influenced by property value inheritance in the applied

template, or by default value.

The theme styles use a type as the key in their definitions. However, when themes are applied to a given element instance, themes lookup for this type is performed by checking the [DefaultStyleKey](#) property on a control. This is in contrast to using the literal Type, as implicit styles do. The value of [DefaultStyleKey](#) would inherit to derived classes even if the implementer did not change it (the intended way of changing the property is not to override it at the property level, but to instead change its default value in property metadata). This indirection enables base classes to define the theme styles for derived elements that do not otherwise have a style (or more importantly, do not have a template within that style and would thus have no default visual appearance at all). Thus, you can derive `MyButton` from `Button` and will still get the `Button` default template. If you were the control author of `MyButton` and you wanted a different behavior, you could override the dependency property metadata for [DefaultStyleKey](#) on `MyButton` to return a different key, and then define the relevant theme styles including template for `MyButton` that you must package with your `MyButton` control. For more details on themes, styles, and control authoring, see [Control Authoring Overview](#).

## Dynamic Resource References and Binding

Dynamic resource references and binding operations respect the precedence of the location at which they are set. For example, a dynamic resource applied to a local value acts per precedence item 3, a binding for a property setter within a theme style applies at precedence item 9, and so on. Because dynamic resource references and binding must both be able to obtain values from the run time state of the application, this entails that the actual process of determining the property value precedence for any given property extends into the run time as well.

Dynamic resource references are not strictly speaking part of the property system, but they do have a lookup order of their own which interacts with the sequence listed above. That precedence is documented more thoroughly in the [XAML Resources](#). The basic summation of that precedence is: element to page root, application, theme, system.

Dynamic resources and bindings have the precedence of where they were set, but the value is deferred. One consequence of this is that if you set a dynamic resource or binding to a local value, any change to the local value replaces the dynamic resource or binding entirely. Even if you call the [ClearValue](#) method to clear the locally set value, the dynamic resource or binding will not be restored. In fact, if you call [ClearValue](#) on a property that has a dynamic resource or binding in place (with no literal local value), they are cleared by the [ClearValue](#) call too.

## SetCurrentValue

The [SetCurrentValue](#) method is another way to set a property, but it is not in the order of precedence. Instead, [SetCurrentValue](#) enables you to change the value of a property without overwriting the source of a previous value. You can use [SetCurrentValue](#) any time that you want to set a value without giving that value the precedence of a local value. For example, if a property is set by a trigger and then assigned another value via [SetCurrentValue](#), the property system still respects the trigger and the property will change if the trigger's action occurs. [SetCurrentValue](#) enables you to change the property's value without giving it a source with a higher precedence. Likewise, you can use [SetCurrentValue](#) to change the value of a property without overwriting a binding.

## Coercion, Animations, and Base Value

Coercion and animation both act on a value that is termed as the "base value" throughout this SDK. The base value is thus whatever value is determined through evaluating upwards in the items until item 2 is reached.

For an animation, the base value can have an effect on the animated value, if that animation does not specify both "From" and "To" for certain behaviors, or if the animation deliberately reverts to the base value when completed. To see this in practice, run the [From, To, and By Animation Target Values Sample](#). Try setting the local values of the rectangle height in the example, such that the initial local value differs from any "From" in the

animation. You will note that the animations start right away using the "From" values and replace the base value once started. The animation might specify to return to the value found before animation once it is completed by specifying the Stop [FillBehavior](#). Afterwards, normal precedence is used for the base value determination.

Multiple animations might be applied to a single property, with each of these animations possibly having been defined from different points in the value precedence. However, these animations will potentially composite their values, rather than just applying the animation from the higher precedence. This depends on exactly how the animations are defined, and the type of the value that is being animated. For more information about animating properties, see [Animation Overview](#).

Coercion applies at the highest level of all. Even an already running animation is subject to value coercion. Certain existing dependency properties in WPF have built-in coercion. For a custom dependency property, you define the coercion behavior for a custom dependency property by writing a [CoerceValueCallback](#) and passing the callback as part of metadata when you create the property. You can also override coercion behavior of existing properties by overriding the metadata on that property in a derived class. Coercion interacts with the base value in such a way that the constraints on coercion are applied as those constraints exist at the time, but the base value is still retained. Therefore, if constraints in coercion are later lifted, the coercion will return the closest value possible to that base value, and potentially the coercion influence on a property will cease as soon as all constraints are lifted. For more information about coercion behavior, see [Dependency Property Callbacks and Validation](#).

## Trigger Behaviors

Controls often define trigger behaviors as part of their default style in themes. Setting local properties on controls might prevent the triggers from being able to respond to user-driven events either visually or behaviorally. The most common use of a property trigger is for control or state properties such as [IsSelected](#). For example, by default when a [Button](#) is disabled (trigger for [IsEnabled](#) is `false`) then the [Foreground](#) value in the theme style is what causes the control to appear "grayed out". But if you have set a local [Foreground](#) value, that normal gray-out color will be overruled in precedence by your local property set, even in this property-triggered scenario. Be cautious of setting values for properties that have theme-level trigger behaviors and make sure you are not unduly interfering with the intended user experience for that control.

## ClearValue and Value Precedence

The [ClearValue](#) method provides an expedient means to clear any locally applied value from a dependency property that is set on an element. However, calling [ClearValue](#) is not a guarantee that the default as established in metadata during property registration is the new effective value. All of the other participants in value precedence are still active. Only the locally set value has been removed from the precedence sequence. For example, if you call [ClearValue](#) on a property where that property is also set by a theme style, then the theme value is applied as the new value rather than the metadata-based default. If you want to take all property value participants out of the process and set the value to the registered metadata default, you can obtain that default value definitively by querying the dependency property metadata, and then you can use the default value to locally set the property with a call to [SetValue](#).

## See also

- [DependencyObject](#)
- [DependencyProperty](#)
- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)
- [Dependency Property Callbacks and Validation](#)

# Read-Only Dependency Properties

11/3/2019 • 3 minutes to read • [Edit Online](#)

This topic describes read-only dependency properties, including existing read-only dependency properties and the scenarios and techniques for creating a custom read-only dependency property.

## Prerequisites

This topic assumes that you understand the basic scenarios of implementing a dependency property, and how metadata is applied to a custom dependency property. See [Custom Dependency Properties](#) and [Dependency Property Metadata](#) for context.

## Existing Read-Only Dependency Properties

Some of the dependency properties defined in the Windows Presentation Foundation (WPF) framework are read-only. The typical reason for specifying a read-only dependency property is that these are properties that should be used for state determination, but where that state is influenced by a multitude of factors, but just setting the property to that state isn't desirable from a user interface design perspective. For example, the property [IsMouseOver](#) is really just surfacing state as determined from the mouse input. Any attempt to set this value programmatically by circumventing the true mouse input would be unpredictable and would cause inconsistency.

By virtue of not being settable, read-only dependency properties aren't appropriate for many of the scenarios for which dependency properties normally offer a solution (namely: data binding, directly styvable to a value, validation, animation, inheritance). Despite not being settable, read-only dependency properties still have some of the additional capabilities supported by dependency properties in the property system. The most important remaining capability is that the read-only dependency property can still be used as a property trigger in a style. You can't enable triggers with a normal common language runtime (CLR) property; it needs to be a dependency property. The aforementioned [IsMouseOver](#) property is a perfect example of a scenario where it might be quite useful to define a style for a control, where some visible property such as a background, foreground, or similar properties of composited elements within the control will change when the user places a mouse over some defined region of your control. Changes in a read-only dependency property can also be detected and reported by the property system's inherent invalidation processes, and this in fact supports the property trigger functionality internally.

## Creating Custom Read-Only Dependency Properties

Make sure to read the section above regarding why read-only dependency properties won't work for many typical dependency-property scenarios. But if you have an appropriate scenario, you may wish to create your own read-only dependency property.

Much of the process of creating a read-only dependency property is the same as is described in the [Custom Dependency Properties](#) and [Implement a Dependency Property](#) topics. There are three important differences:

- When registering your property, call the [RegisterReadOnly](#) method instead of the normal [Register](#) method for property registration.
- When implementing the CLR "wrapper" property, make sure that the wrapper too doesn't have a set implementation, so that there is no inconsistency in read-only state for the public wrapper you expose.
- The object returned by the read-only registration is [DependencyPropertyKey](#) rather than [DependencyProperty](#). You should still store this field as a member but typically you would not make it a

public member of the type.

Whatever private field or value you have backing your read-only dependency property can of course be fully writable using whatever logic you decide. However, the most straightforward way to set the property either initially or as part of runtime logic is to use the property system's APIs, rather than circumventing the property system and setting the private backing field directly. In particular, there is a signature of [SetValue](#) that accepts a parameter of type [DependencyPropertyKey](#). How and where you set this value programmatically within your application logic will affect how you may wish to set access on the [DependencyPropertyKey](#) created when you first registered the dependency property. If you handle this logic all within the class you could make it private, or if you require it to be set from other portions of the assembly you might set it internal. One approach is to call [SetValue](#) within a class event handler of a relevant event that informs a class instance that the stored property value needs to be changed. Another approach is to tie dependency properties together by using paired [PropertyChangedCallback](#) and [CoerceValueCallback](#) callbacks as part of those properties' metadata during registration.

Because the [DependencyPropertyKey](#) is private, and is not propagated by the property system outside of your code, a read-only dependency property does have better setting security than a read-write dependency property. For a read-write dependency property, the identifying field is explicitly or implicitly public and thus the property is widely settable. For more specifics, see [Dependency Property Security](#).

## See also

- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)
- [Styling and Templating](#)

# Property Value Inheritance

8/22/2019 • 4 minutes to read • [Edit Online](#)

Property value inheritance is a feature of the Windows Presentation Foundation (WPF) property system. Property value inheritance enables child elements in a tree of elements to obtain the value of a particular property from parent elements, inheriting that value as it was set anywhere in the nearest parent element. The parent element might also have obtained its value through property value inheritance, so the system potentially recurses all the way to the page root. Property value inheritance is not the default property system behavior; a property must be established with a particular metadata setting in order to cause that property to initiate property value inheritance on child elements.

## Property Value Inheritance Is Containment Inheritance

"Inheritance" as a term here is not quite the same concept as inheritance in the context of types and general object-oriented programming, where derived classes inherit member definitions from their base classes. That meaning of inheritance is also active in WPF: properties defined in various base classes are exposed as attributes for derived XAML classes when used as elements, and exposed as members for code. Property value inheritance is particularly about how property values can inherit from one element to another on the basis of the parent-child relationships within a tree of elements. That tree of elements is most directly visible when nesting elements inside other elements as you define applications in XAML markup. Trees of objects can also be created programmatically by adding objects to designated collections of other objects, and property value inheritance works the same way in the finished tree at run time.

## Practical Applications of Property Value Inheritance

The WPF APIs include several properties that have property inheritance enabled. Typically, the scenario for these is that they involve a property where it is appropriate that the property be set only once per page, but where that property is also a member of one of the base element classes and thus would also exist on most of the child elements. For example, the [FlowDirection](#) property controls which direction flowed content should be presented and arranged on the page. Typically, you want the text flow concept to be handled consistently throughout all child elements. If flow direction were for some reason reset in some level of the element tree by user or environment action, it should typically be reset throughout. When the [FlowDirection](#) property is made to inherit, the value need only be set or reset once at the level in the element tree that encompasses the presentation needs of each page in the application. Even the initial default value will inherit in this way. The property value inheritance model still enables individual elements to reset the value for the rare cases where having a mix of flow directions is intentional.

## Making a Custom Property Inheritable

By changing a custom property's metadata, you can also make your own custom properties inheritable. Note, however, that designating a property as inheritable does have some performance considerations. In cases where that property does not have an established local value, or a value obtained through styles, templates, or data binding, an inheritable property provides its assigned property values to all child elements in the logical tree.

To make a property participate in value inheritance, create a custom attached property, as described in [Register an Attached Property](#). Register the property with metadata ([FrameworkPropertyMetadata](#)) and specify the "Inherits" option in the options settings within that metadata. Also make sure that the property has an established default value, because that value will now inherit. Although you registered the property as attached, you might also want to create a property "wrapper" for get/set access on the owner type, just as you would for

an "nonattached" dependency property. After doing so, the inheritable property can either be set by using the direct property wrapper on the owner type or derived types, or it can be set by using the attached property syntax on any [DependencyObject](#).

Attached properties are conceptually similar to global properties; you can check for the value on any [DependencyObject](#) and get a valid result. The typical scenario for attached properties is to set property values on child elements, and that scenario is more effective if the property in question is an attached property that is always implicitly present as an attached property on each element ([DependencyObject](#)) in the tree.

#### NOTE

Although property value inheritance might appear to work for nonattached dependency properties, the inheritance behavior for a nonattached property through certain element boundaries in the run-time tree is undefined. Always use [RegisterAttached](#) to register properties where you specify [Inherits](#) in the metadata.

## Inheriting Property Values Across Tree Boundaries

Property inheritance works by traversing a tree of elements. This tree is often parallel to the logical tree. However, whenever you include a WPF core-level object in the markup that defines an element tree, such as a [Brush](#), you have created a discontinuous logical tree. A true logical tree does not conceptually extend through the [Brush](#), because the logical tree is a WPF framework-level concept. You can see this reflected in the results when using the methods of [LogicalTreeHelper](#). However, property value inheritance can bridge this gap in the logical tree and can still pass inherited values through, so long as the inheritable property was registered as an attached property and no deliberate inheritance-blocking boundary (such as a [Frame](#)) is encountered.

## See also

- [Dependency Property Metadata](#)
- [Attached Properties Overview](#)
- [Dependency Property Value Precedence](#)

# Dependency Property Security

8/22/2019 • 2 minutes to read • [Edit Online](#)

Dependency properties should generally be considered to be public properties. The nature of the Windows Presentation Foundation (WPF) property system prevents the ability to make security guarantees about a dependency property value.

## Access and Security of Wrappers and Dependency Properties

Typically, dependency properties are implemented along with "wrapper" common language runtime (CLR) properties that simplify getting or setting the property from an instance. But the wrappers are really just convenience methods that implement the underlying [GetValue](#) and [SetValue](#) static calls that are used when interacting with dependency properties. Thinking of it in another way, the properties are exposed as common language runtime (CLR) properties that happen to be backed by a dependency property rather than by a private field. Security mechanisms applied to the wrappers do not parallel the property system behavior and access of the underlying dependency property. Placing a security demand on the wrapper will only prevent the usage of the convenience method but will not prevent calls to [GetValue](#) or [SetValue](#). Similarly, placing protected or private access level on the wrappers does not provide any effective security.

If you are writing your own dependency properties, you should declare the wrappers and the [DependencyProperty](#) identifier field as public members, so that callers do not get misleading information about the true access level of that property (because of its store being implemented as a dependency property).

For a custom dependency property, you can register your property as a read-only dependency property, and this does provide an effective means of preventing a property being set by anyone that does not hold a reference to the [DependencyPropertyKey](#) for that property. For more information, see [Read-Only Dependency Properties](#).

### NOTE

Declaring a [DependencyProperty](#) identifier field private is not forbidden, and it can conceivably be used to help reduce the immediately exposed namespace of a custom class, but such a property should not be considered "private" in the same sense as the common language runtime (CLR) language definitions define that access level, for reasons described in the next section.

## Property System Exposure of Dependency Properties

It is not generally useful, and it is potentially misleading, to declare a [DependencyProperty](#) as any access level other than public. That access level setting only prevents someone from being able to get a reference to the instance from the declaring class. But there are several aspects of the property system that will return a [DependencyProperty](#) as the means of identifying a particular property as it exists on an instance of a class or a derived class instance, and this identifier is still usable in a [SetValue](#) call even if the original static identifier is declared as nonpublic. Also, [OnPropertyChanged](#) virtual methods receive information of any existing dependency property that changed value. In addition, the [GetLocalValueEnumerator](#) method returns identifiers for any property on instances with a locally set value.

### Validation and Security

Applying a demand to a [ValidateValueCallback](#) and expecting the validation failure on a demand failure to prevent a property from being set is not an adequate security mechanism. Set-value invalidation enforced through [ValidateValueCallback](#) could also be suppressed by malicious callers, if those callers are operating within the application domain.

## See also

- [Custom Dependency Properties](#)

# Safe Constructor Patterns for DependencyObjects

11/21/2019 • 5 minutes to read • [Edit Online](#)

Generally, class constructors should not call callbacks such as virtual methods or delegates, because constructors can be called as base initialization of constructors for a derived class. Entering the virtual might be done at an incomplete initialization state of any given object. However, the property system itself calls and exposes callbacks internally, as part of the dependency property system. As simple an operation as setting a dependency property value with [SetValue](#) call potentially includes a callback somewhere in the determination. For this reason, you should be careful when setting dependency property values within the body of a constructor, which can become problematic if your type is used as a base class. There is a particular pattern for implementing [DependencyObject](#) constructors that avoids specific problems with dependency property states and the inherent callbacks, which is documented here.

## Property System Virtual Methods

The following virtual methods or callbacks are potentially called during the computations of the [SetValue](#) call that sets a dependency property value: [ValidateValueCallback](#), [PropertyChangedCallback](#), [CoerceValueCallback](#), [OnPropertyChanged](#). Each of these virtual methods or callbacks serves a particular purpose in expanding the versatility of the Windows Presentation Foundation (WPF) property system and dependency properties. For more information on how to use these virtuals to customize property value determination, see [Dependency Property Callbacks and Validation](#).

### **FXCop Rule Enforcement vs. Property System Virtuals**

If you use the Microsoft tool FXCop as part of your build process, and you either derive from certain WPF framework classes calling the base constructor, or implement your own dependency properties on derived classes, you might encounter a particular FXCop rule violation. The name string for this violation is:

`DoNotCallOverridableMethodsInConstructors`

This is a rule that is part of the default public rule set for FXCop. What this rule might be reporting is a trace through the dependency property system that eventually calls a dependency property system virtual method. This rule violation might continue to appear even after following the recommended constructor patterns documented in this topic, so you might need to disable or suppress that rule in your FXCop rule set configuration.

### **Most Issues Come From Deriving Classes, Not Using Existing Classes**

The issues reported by this rule occur when a class that you implement with virtual methods in its construction sequence is then derived from. If you seal your class, or otherwise know or enforce that your class will not be derived from, the considerations explained here and the issues that motivated the FXCop rule do not apply to you. However, if you are authoring classes in such a way that they are intended to be used as base classes, for instance if you are creating templates, or an expandable control library set, you should follow the patterns recommended here for constructors.

### **Default Constructors Must Initialize All Values Requested By Callbacks**

Any instance members that are used by your class overrides or callbacks (the callbacks from the list in the Property System Virtuals section) must be initialized in your class parameterless constructor, even if some of those values are filled by "real" values through parameters of the nonparameterless constructors.

The following example code (and subsequent examples) is a pseudo-C# example that violates this rule and explains the problem:

```

public class MyClass : DependencyObject
{
    public MyClass() {}
    public MyClass(object toSetWobble)
        : this()
    {
        Wobble = toSetWobble; //this is backed by a DependencyProperty
        _myList = new ArrayList(); // this line should be in the default ctor
    }
    public static readonly DependencyProperty WobbleProperty =
        DependencyProperty.Register("Wobble", typeof(object), typeof(MyClass));
    public object Wobble
    {
        get { return GetValue(WobbleProperty); }
        set { SetValue(WobbleProperty, value); }
    }
    protected override void OnPropertyChanged(DependencyPropertyChangedEventArgs e)
    {
        int count = _myList.Count; // null-reference exception
    }
    private ArrayList _myList;
}

```

When application code calls `new MyClass(objectvalue)`, this calls the parameterless constructor and base class constructors. Then it sets `Property1 = object1`, which calls the virtual method `OnPropertyChanged` on the owning `MyClass DependencyObject`. The override refers to `_myList`, which has not been initialized yet.

One way to avoid these issues is to make sure that callbacks use only other dependency properties, and that each such dependency property has an established default value as part of its registered metadata.

## Safe Constructor Patterns

To avoid the risks of incomplete initialization if your class is used as a base class, follow these patterns:

### Parameterless constructors calling base initialization

Implement these constructors calling the base default:

```

public MyClass : SomeBaseClass {
    public MyClass() : base() {
        // ALL class initialization, including initial defaults for
        // possible values that other ctors specify or that callbacks need.
    }
}

```

### Non-default (convenience) constructors, not matching any base signatures

If these constructors use the parameters to set dependency properties in the initialization, first call your own class parameterless constructor for initialization, and then use the parameters to set dependency properties. These could either be dependency properties defined by your class, or dependency properties inherited from base classes, but in either case use the following pattern:

```

public MyClass : SomeBaseClass {
    public MyClass(object toSetProperty1) : this() {
        // Class initialization NOT done by default.
        // Then, set properties to values as passed in ctor parameters.
        Property1 = toSetProperty1;
    }
}

```

### Non-default (convenience) constructors, which do match base signatures

Instead of calling the base constructor with the same parameterization, again call your own class' parameterless constructor. Do not call the base initializer; instead you should call `this()`. Then reproduce the original constructor behavior by using the passed parameters as values for setting the relevant properties. Use the original base constructor documentation for guidance in determining the properties that the particular parameters are intended to set:

```
public MyClass : SomeBaseClass {  
    public MyClass(object toSetProperty1) : this() {  
        // Class initialization NOT done by default.  
        // Then, set properties to values as passed in ctor parameters.  
        Property1 = toSetProperty1;  
    }  
}
```

#### Must match all signatures

For cases where the base type has multiple signatures, you must deliberately match all possible signatures with a constructor implementation of your own that uses the recommended pattern of calling the class parameterless constructor before setting further properties.

#### Setting dependency properties with SetValue

These same patterns apply if you are setting a property that does not have a wrapper for property setting convenience, and set values with `SetValue`. Your calls to `SetValue` that pass through constructor parameters should also call the class' parameterless constructor for initialization.

## See also

- [Custom Dependency Properties](#)
- [Dependency Properties Overview](#)
- [Dependency Property Security](#)

# Collection-Type Dependency Properties

11/12/2019 • 4 minutes to read • [Edit Online](#)

This topic provides guidance and suggested patterns for how to implement a dependency property where the type of the property is a collection type.

## Implementing a Collection-Type Dependency Property

For a dependency property in general, the implementation pattern that you follow is that you define a CLR property wrapper, where that property is backed by a [DependencyProperty](#) identifier rather than a field or other construct. You follow this same pattern when you implement a collection-type property. However, a collection-type property introduces some complexity to the pattern whenever the type that is contained within the collection is itself a [DependencyObject](#) or [Freezable](#) derived class.

## Initializing the Collection Beyond the Default Value

When you create a dependency property, you do not specify the property default value as the initial field value. Instead, you specify the default value through the dependency property metadata. If your property is a reference type, the default value specified in dependency property metadata is not a default value per instance; instead it is a default value that applies to all instances of the type. Therefore you must be careful to not use the singular static collection defined by the collection property metadata as the working default value for newly created instances of your type. Instead, you must make sure that you deliberately set the collection value to a unique (instance) collection as part of your class constructor logic. Otherwise you will have created an unintentional singleton class.

Consider the following example. The following section of the example shows the definition for a class `Aquarium`, which contains a flaw with the default value. The class defines the collection type dependency property `AquariumObjects`, which uses the generic `List<T>` type with a [FrameworkElement](#) type constraint. In the `Register(String, Type, Type, PropertyMetadata)` call for the dependency property, the metadata establishes the default value to be a new generic `List<T>`.

### WARNING

The following code does not behave correctly.

```

public class Fish : FrameworkElement { }
public class Aquarium : DependencyObject {
    private static readonly DependencyPropertyKey AquariumContentsPropertyKey =
        DependencyProperty.RegisterReadOnly(
            "AquariumContents",
            typeof(List<FrameworkElement>),
            typeof(Aquarium),
            new FrameworkPropertyMetadata(new List<FrameworkElement>())
        );
    public static readonly DependencyProperty AquariumContentsProperty =
        AquariumContentsPropertyKey.DependencyProperty;

    public List<FrameworkElement> AquariumContents
    {
        get { return (List<FrameworkElement>)GetValue(AquariumContentsProperty); }
    }

    // ...
}

```

```

Public Class Fish
    Inherits FrameworkElement
End Class
Public Class Aquarium
    Inherits DependencyObject
    Private Shared ReadOnly AquariumContentsPropertyKey As DependencyProperty =
        DependencyProperty.RegisterReadOnly("AquariumContents", GetType(List(Of FrameworkElement)), GetType(Aquarium),
        New FrameworkPropertyMetadata(New List(Of FrameworkElement)()))
    Public Shared ReadOnly AquariumContentsProperty As DependencyProperty =
        AquariumContentsPropertyKey.DependencyProperty

    Public ReadOnly Property AquariumContents() As List(Of FrameworkElement)
        Get
            Return CType(GetValue(AquariumContentsProperty), List(Of FrameworkElement))
        End Get
    End Property
    ' ...
End Class

```

However, if you just left the code as shown, that single list default value is shared for all instances of `Aquarium`. If you ran the following test code, which is intended to show how you would instantiate two separate `Aquarium` instances and add a single different `Fish` to each of them, you would see a surprising result:

```

Aquarium myAq1 = new Aquarium();
Aquarium myAq2 = new Aquarium();
Fish f1 = new Fish();
Fish f2 = new Fish();
myAq1.AquariumContents.Add(f1);
myAq2.AquariumContents.Add(f2);
MessageBox.Show("aq1 contains " + myAq1.AquariumContents.Count.ToString() + " things");
MessageBox.Show("aq2 contains " + myAq2.AquariumContents.Count.ToString() + " things");

```

```

Dim myAq1 As New Aquarium()
Dim myAq2 As New Aquarium()
Dim f1 As New Fish()
Dim f2 As New Fish()
myAq1.AquariumContents.Add(f1)
myAq2.AquariumContents.Add(f2)
MessageBox.Show("aq1 contains " & myAq1.AquariumContents.Count.ToString() & " things")
MessageBox.Show("aq2 contains " & myAq2.AquariumContents.Count.ToString() & " things")

```

Instead of each collection having a count of one, each collection has a count of two! This is because each `Aquarium` added its `Fish` to the default value collection, which resulted from a single constructor call in the metadata and is therefore shared between all instances. This situation is almost never what you want.

To correct this problem, you must reset the collection dependency property value to a unique instance, as part of the class constructor call. Because the property is a read-only dependency property, you use the `SetValue(DependencyPropertyKey, Object)` method to set it, using the `DependencyPropertyKey` that is only accessible within the class.

```

public Aquarium() : base()
{
    SetValue(AquariumContentsPropertyKey, new List<FrameworkElement>());
}

```

```

Public Sub New()
    MyBase.New()
    SetValue(AquariumContentsPropertyKey, New List(Of FrameworkElement)())
End Sub

```

Now, if you ran that same test code again, you could see more expected results, where each `Aquarium` supported its own unique collection.

There would be a slight variation on this pattern if you chose to have your collection property be read-write. In that case, you could call the public set accessor from the constructor to do the initialization, which would still be calling the nonkey signature of `SetValue(DependencyProperty, Object)` within your set wrapper, using a public `DependencyProperty` identifier.

## Reporting Binding Value Changes from Collection Properties

A collection property that is itself a dependency property does not automatically report changes to its subproperties. If you are creating bindings into a collection, this can prevent the binding from reporting changes, thus invalidating some data binding scenarios. However, if you use the collection type `FreezableCollection<T>` as your collection type, then subproperty changes to contained elements in the collection are properly reported, and binding works as expected.

To enable subproperty binding in a dependency object collection, create the collection property as type `FreezableCollection<T>`, with a type constraint for that collection to any `DependencyObject` derived class.

## See also

- [FreezableCollection<T>](#)
- [XAML and Custom Classes for WPF](#)
- [Data Binding Overview](#)
- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)

- Dependency Property Metadata

# XAML Loading and Dependency Properties

11/3/2019 • 2 minutes to read • [Edit Online](#)

The current WPF implementation of its XAML processor is inherently dependency property aware. The WPF XAML processor uses property system methods for dependency properties when loading binary XAML and processing attributes that are dependency properties. This effectively bypasses the property wrappers. When you implement custom dependency properties, you must account for this behavior and should avoid placing any other code in your property wrapper other than the property system methods [GetValue](#) and [SetValue](#).

## Prerequisites

This topic assumes that you understand dependency properties both as consumer and author and have read [Dependency Properties Overview](#) and [Custom Dependency Properties](#). You should also have read [XAML Overview \(WPF\)](#) and [XAML Syntax In Detail](#).

## The WPF XAML Loader Implementation, and Performance

For implementation reasons, it is computationally less expensive to identify a property as a dependency property and access the property system [SetValue](#) method to set it, rather than using the property wrapper and its setter. This is because a XAML processor must infer the entire object model of the backing code based only on knowing the type and member relationships that are indicated by the structure of the markup and various strings.

The type is looked up through a combination of xmlns and assembly attributes, but identifying the members, determining which could support being set as an attribute, and resolving what types the property values support would otherwise require extensive reflection using  [PropertyInfo](#). Because dependency properties on a given type are accessible as a storage table through the property system, the WPF implementation of its XAML processor uses this table and infers that any given property ABC can be more efficiently set by calling  [SetValue](#) on the containing  [DependencyObject](#) derived type, using the dependency property identifier  [ABCProperty](#).

## Implications for Custom Dependency Properties

Because the current WPF implementation of the XAML processor behavior for property setting bypasses the wrappers entirely, you should not put any additional logic into the set definitions of the wrapper for your custom dependency property. If you put such logic in the set definition, then the logic will not be executed when the property is set in XAML rather than in code.

Similarly, other aspects of the XAML processor that obtain property values from XAML processing also use  [GetValue](#) rather than using the wrapper. Therefore, you should also avoid any additional implementation in the `get` definition beyond the  [GetValue](#) call.

The following example is a recommended dependency property definition with wrappers, where the property identifier is stored as a `public static readonly` field, and the `get` and `set` definitions contain no code beyond the necessary property system methods that define the dependency property backing.

```

public static readonly DependencyProperty AquariumGraphicProperty = DependencyProperty.Register(
    "AquariumGraphic",
    typeof(Uri),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(null,
        FrameworkPropertyMetadataOptions.AffectsRender,
        new PropertyChangedCallback(OnUriChanged)
    )
);
public Uri AquariumGraphic
{
    get { return (Uri)GetValue(AquariumGraphicProperty); }
    set { SetValue(AquariumGraphicProperty, value); }
}

```

```

Public Shared ReadOnly AquariumGraphicProperty As DependencyProperty =
DependencyProperty.Register("AquariumGraphic", GetType(Uri), GetType(AquariumObject), New
FrameworkPropertyMetadata(Nothing, FrameworkPropertyMetadataOptions.AffectsRender, New
PropertyChangedCallback(AddressOf OnUriChanged)))
Public Property AquariumGraphic() As Uri
    Get
        Return CType(GetValue(AquariumGraphicProperty), Uri)
    End Get
    Set(ByVal value As Uri)
        SetValue(AquariumGraphicProperty, value)
    End Set
End Property

```

## See also

- [Dependency Properties Overview](#)
- [XAML Overview \(WPF\)](#)
- [Dependency Property Metadata](#)
- [Collection-Type Dependency Properties](#)
- [Dependency Property Security](#)
- [Safe Constructor Patterns for DependencyObjects](#)

# Properties How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

## In This Section

[Implement a Dependency Property](#)

[Add an Owner Type for a Dependency Property](#)

[Register an Attached Property](#)

[Override Metadata for a Dependency Property](#)

## Reference

[DependencyProperty](#)

[PropertyMetadata](#)

[FrameworkPropertyMetadata](#)

[DependencyObject](#)

## Related Sections

[Properties](#)

# How to: Implement a Dependency Property

7/23/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to back a common language runtime (CLR) property with a `DependencyProperty` field, thus defining a dependency property. When you define your own properties and want them to support many aspects of Windows Presentation Foundation (WPF) functionality, including styles, data binding, inheritance, animation, and default values, you should implement them as a dependency property.

## Example

The following example first registers a dependency property by calling the `Register` method. The name of the identifier field that you use to store the name and characteristics of the dependency property must be the `Name` you chose for the dependency property as part of the `Register` call, appended by the literal string `Property`. For instance, if you register a dependency property with a `Name` of `Location`, then the identifier field that you define for the dependency property must be named `LocationProperty`.

In this example, the name of the dependency property and its CLR accessor is `State`; the identifier field is `StateProperty`; the type of the property is `Boolean`; and the type that registers the dependency property is `MyStateControl`.

If you fail to follow this naming pattern, designers might not report your property correctly, and certain aspects of property system style application might not behave as expected.

You can also specify default metadata for a dependency property. This example registers the default value of the `State` dependency property to be `false`.

```
public class MyStateControl : ButtonBase
{
    public MyStateControl() : base() { }

    public Boolean State
    {
        get { return (Boolean)this.GetValue(StateProperty); }
        set { this.SetValue(StateProperty, value); }
    }

    public static readonly DependencyProperty StateProperty = DependencyProperty.Register(
        "State", typeof(Boolean), typeof(MyStateControl), new PropertyMetadata(false));
}
```

```
Public Class MyStateControl
    Inherits ButtonBase
    Public Sub New()
        MyBase.New()
    End Sub
    Public Property State() As Boolean
        Get
            Return CType(Me.GetValue(StateProperty), Boolean)
        End Get
        Set(ByVal value As Boolean)
            Me.SetValue(StateProperty, value)
        End Set
    End Property
    Public Shared ReadOnly StateProperty As DependencyProperty = DependencyProperty.Register("State",
        GetType(Boolean), GetType(MyStateControl), New PropertyMetadata(False))
End Class
```

For more information about how and why to implement a dependency property, as opposed to just backing a CLR property with a private field, see [Dependency Properties Overview](#).

## See also

- [Dependency Properties Overview](#)
- [How-to Topics](#)

# How to: Add an Owner Type for a Dependency Property

7/23/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to add a class as an owner of a dependency property registered for a different type. By doing this, the WPF XAML reader and property system are both able to recognize the class as an additional owner of the property. Adding as owner optionally allows the adding class to provide type-specific metadata.

In the following example, `StateProperty` is a property registered by the `MyStateControl` class. The class `UnrelatedStateControl` adds itself as an owner of the `StateProperty` using the `AddOwner` method, specifically using the signature that allows for new metadata for the dependency property as it exists on the adding type. Notice that you should provide common language runtime (CLR) accessors for the property similar to the example shown in the [Implement a Dependency Property](#) example, as well as re-expose the dependency property identifier on the class being added as owner.

Without wrappers, the dependency property would still work from the perspective of programmatic access using `GetValue` or `SetValue`. But you typically want to parallel this property-system behavior with the CLR property wrappers. The wrappers make it easier to set the dependency property programmatically, and make it possible to set the properties as XAML attributes.

To find out how to override default metadata, see [Override Metadata for a Dependency Property](#).

## Example

```
public class MyStateControl : ButtonBase
{
    public MyStateControl() : base() { }
    public Boolean State
    {
        get { return (Boolean)this.GetValue(StateProperty); }
        set { this.SetValue(StateProperty, value); }
    }
    public static readonly DependencyProperty StateProperty = DependencyProperty.Register(
        "State", typeof(Boolean), typeof(MyStateControl), new PropertyMetadata(false));
}
```

```
Public Class MyStateControl
    Inherits ButtonBase
    Public Sub New()
        MyBase.New()
    End Sub
    Public Property State() As Boolean
        Get
            Return CType(Me.GetValue(StateProperty), Boolean)
        End Get
        Set(ByVal value As Boolean)
            Me.SetValue(StateProperty, value)
        End Set
    End Property
    Public Shared ReadOnly StateProperty As DependencyProperty = DependencyProperty.Register("State",
        GetType(Boolean), GetType(MyStateControl), New PropertyMetadata(False))
End Class
```

```
public class UnrelatedStateControl : Control
{
    public UnrelatedStateControl() { }
    public static readonly DependencyProperty StateProperty =
        MyStateControl.StateProperty.AddOwner(typeof(UnrelatedStateControl), new PropertyMetadata(true));
    public Boolean State
    {
        get { return (Boolean)this.GetValue(StateProperty); }
        set { this.SetValue(StateProperty, value); }
    }
}
```

```
Public Class UnrelatedStateControl
    Inherits Control
    Public Sub New()
    End Sub
    Public Shared ReadOnly StateProperty As DependencyProperty =
        MyStateControl.StateProperty.AddOwner(GetType(UnrelatedStateControl), New PropertyMetadata(True))
    Public Property State() As Boolean
        Get
            Return CType(Me.GetValue(StateProperty), Boolean)
        End Get
        Set(ByVal value As Boolean)
            Me.SetValue(StateProperty, value)
        End Set
    End Property
End Class
```

## See also

- [Custom Dependency Properties](#)
- [Dependency Properties Overview](#)

# How to: Register an Attached Property

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to register an attached property and provide public accessors so that you can use the property in both Extensible Application Markup Language (XAML) and code. Attached properties are a syntax concept defined by Extensible Application Markup Language (XAML). Most attached properties for WPF types are also implemented as dependency properties. You can use dependency properties on any [DependencyObject](#) types.

## Example

The following example shows how to register an attached property as a dependency property, by using the [RegisterAttached](#) method. The provider class has the option of providing default metadata for the property that is applicable when the property is used on another class, unless that class overrides the metadata. In this example, the default value of the `IsBubbleSource` property is set to `false`.

The provider class for an attached property (even if it is not registered as a dependency property) must provide static get and set accessors that follow the naming convention `Set [AttachedPropertyName]` and `Get [AttachedPropertyName]`. These accessors are required so that the acting XAML reader can recognize the property as an attribute in XAML and resolve the appropriate types.

```
public static readonly DependencyProperty IsBubbleSourceProperty = DependencyProperty.RegisterAttached(
    "IsBubbleSource",
    typeof(Boolean),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(false, FrameworkPropertyMetadataOptions.AffectsRender)
);
public static void SetIsBubbleSource(UIElement element, Boolean value)
{
    element.SetValue(IsBubbleSourceProperty, value);
}
public static Boolean GetIsBubbleSource(UIElement element)
{
    return (Boolean)element.GetValue(IsBubbleSourceProperty);
}
```

```
Public Shared ReadOnly IsBubbleSourceProperty As DependencyProperty =
DependencyProperty.RegisterAttached("IsBubbleSource", GetType(Boolean), GetType(AquariumObject), New
FrameworkPropertyMetadata(False, FrameworkPropertyMetadataOptions.AffectsRender))
Public Shared Sub SetIsBubbleSource(ByVal element As UIElement, ByVal value As Boolean)
    element.SetValue(IsBubbleSourceProperty, value)
End Sub
Public Shared Function GetIsBubbleSource(ByVal element As UIElement) As Boolean
    Return CType(element.GetValue(IsBubbleSourceProperty), Boolean)
End Function
```

## See also

- [DependencyProperty](#)
- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)
- [How-to Topics](#)



# How to: Override Metadata for a Dependency Property

7/9/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to override default dependency property metadata that comes from an inherited class, by calling the [OverrideMetadata](#) method and providing type-specific metadata.

## Example

By defining its [PropertyMetadata](#), a class can define the dependency property's behaviors, such as its default value and property system callbacks. Many dependency property classes already have default metadata established as part of their registration process. This includes the dependency properties that are part of the WPF API. A class that inherits the dependency property through its class inheritance can override the original metadata so that the characteristics of the property that can be altered through metadata will match any subclass-specific requirements.

Overriding metadata on a dependency property must be done prior to that property being placed in use by the property system (this equates to the time that specific instances of objects that register the property are instantiated). Calls to [OverrideMetadata](#) must be performed within the static constructors of the type that provides itself as the `forType` parameter of [OverrideMetadata](#). If you attempt to change metadata once instances of the owner type exist, this will not raise exceptions, but will result in inconsistent behaviors in the property system. Also, metadata can only be overridden once per type. Subsequent attempts to override metadata on the same type will raise an exception.

In the following example, the custom class `MyAdvancedStateControl` overrides the metadata provided for `StateProperty` by `MyAdvancedStateControl` with new property metadata. For instance, the default value of the `StateProperty` is now `true` when the property is queried on a newly constructed `MyAdvancedStateControl` instance.

```
public class MyStateControl : ButtonBase
{
    public MyStateControl() : base() { }

    public Boolean State
    {
        get { return (Boolean)this.GetValue(StateProperty); }
        set { this.SetValue(StateProperty, value); }
    }

    public static readonly DependencyProperty StateProperty = DependencyProperty.Register(
        "State", typeof(Boolean), typeof(MyStateControl), new PropertyMetadata(false));
}
```

```

Public Class MyStateControl
    Inherits ButtonBase
    Public Sub New()
        MyBase.New()
    End Sub
    Public Property State() As Boolean
        Get
            Return CType(Me.GetValue(StateProperty), Boolean)
        End Get
        Set(ByVal value As Boolean)
            Me.SetValue(StateProperty, value)
        End Set
    End Property
    Public Shared ReadOnly StateProperty As DependencyProperty = DependencyProperty.Register("State",
        GetType(Boolean), GetType(MyStateControl), New PropertyMetadata(False))
End Class

```

```

public class MyAdvancedStateControl : MyStateControl
{
    public MyAdvancedStateControl() : base() { }
    static MyAdvancedStateControl()
    {
        MyStateControl.StateProperty.OverrideMetadata(typeof(MyAdvancedStateControl), new PropertyMetadata(true));
    }
}

```

```

Public Class MyAdvancedStateControl
    Inherits MyStateControl
    Public Sub New()
        MyBase.New()
    End Sub
    Shared Sub New()
        MyStateControl.StateProperty.OverrideMetadata(GetType(MyAdvancedStateControl), New PropertyMetadata(True))
    End Sub
End Class

```

## See also

- [DependencyProperty](#)
- [Dependency Properties Overview](#)
- [Custom Dependency Properties](#)
- [How-to Topics](#)

# Events (WPF)

11/3/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) introduces routed events that can invoke handlers that exist on various listeners in the element tree of an application.

## In This Section

- [Routed Events Overview](#)
- [Attached Events Overview](#)
- [Object Lifetime Events](#)
- [Marking Routed Events as Handled, and Class Handling](#)
- [Preview Events](#)
- [Property Change Events](#)
- [Visual Basic and WPF Event Handling](#)
- [Weak Event Patterns](#)
- [How-to Topics](#)

## Reference

- [RoutedEventArgs](#)
- [EventManager](#)
- [RoutingStrategy](#)

## Related Sections

- [WPF Architecture](#)
- [XAML in WPF](#)
- [Base Elements](#)
- [Element Tree and Serialization](#)
- [Properties](#)
- [Input](#)
- [Resources](#)
- [Styling and Templating](#)
- [WPF Content Model](#)
- [Threading Model](#)

# Routed Events Overview

11/3/2019 • 23 minutes to read • [Edit Online](#)

This topic describes the concept of routed events in Windows Presentation Foundation (WPF). The topic defines routed events terminology, describes how routed events are routed through a tree of elements, summarizes how you handle routed events, and introduces how to create your own custom routed events.

## Prerequisites

This topic assumes that you have basic knowledge of the common language runtime (CLR) and object-oriented programming, as well as the concept of how the relationships between WPF elements can be conceptualized as a tree. In order to follow the examples in this topic, you should also understand Extensible Application Markup Language (XAML) and know how to write very basic WPF applications or pages. For more information, see [Walkthrough: My first WPF desktop application](#) and [XAML Overview \(WPF\)](#).

## What Is a Routed Event?

You can think about routed events either from a functional or implementation perspective. Both definitions are presented here, because some people find one or the other definition more useful.

Functional definition: A routed event is a type of event that can invoke handlers on multiple listeners in an element tree, rather than just on the object that raised the event.

Implementation definition: A routed event is a CLR event that is backed by an instance of the [RoutedEventArgs](#) class and is processed by the Windows Presentation Foundation (WPF) event system.

A typical WPF application contains many elements. Whether created in code or declared in XAML, these elements exist in an element tree relationship to each other. The event route can travel in one of two directions depending on the event definition, but generally the route travels from the source element and then "bubbles" upward through the element tree until it reaches the element tree root (typically a page or a window). This bubbling concept might be familiar to you if you have worked with the DHTML object model previously.

Consider the following simple element tree:

```
<Border Height="50" Width="300" BorderBrush="Gray" BorderThickness="1">
    <StackPanel Background="LightGray" Orientation="Horizontal" Button.Click="CommonClickHandler">
        <Button Name="YesButton" Width="Auto" >Yes</Button>
        <Button Name="NoButton" Width="Auto" >No</Button>
        <Button Name="CancelButton" Width="Auto" >Cancel</Button>
    </StackPanel>
</Border>
```

This element tree produces something like the following:



In this simplified element tree, the source of a [Click](#) event is one of the [Button](#) elements, and whichever [Button](#) was clicked is the first element that has the opportunity to handle the event. But if no handler attached to the [Button](#) acts on the event, then the event will bubble upwards to the [Button](#) parent in the element tree, which is the [StackPanel](#). Potentially, the event bubbles to [Border](#), and then beyond to the page root of the element tree (not shown).

In other words, the event route for this [Click](#) event is:

Button-->StackPanel-->Border-->...

### Top-level Scenarios for Routed Events

The following is a brief summary of the scenarios that motivated the routed event concept, and why a typical CLR event was not adequate for these scenarios:

**Control composition and encapsulation:** Various controls in WPF have a rich content model. For example, you can place an image inside of a [Button](#), which effectively extends the visual tree of the button. However, the added image must not break the hit-testing behavior that causes a button to respond to a [Click](#) of its content, even if the user clicks on pixels that are technically part of the image.

**Singular handler attachment points:** In Windows Forms, you would have to attach the same handler multiple times to process events that could be raised from multiple elements. Routed events enable you to attach that handler only once, as was shown in the previous example, and use handler logic to determine where the event came from if necessary. For instance, this might be the handler for the previously shown XAML:

```
private void CommonClickHandler(object sender, RoutedEventArgs e)
{
    FrameworkElement feSource = e.Source as FrameworkElement;
    switch (feSource.Name)
    {
        case "YesButton":
            // do something here ...
            break;
        case "NoButton":
            // do something ...
            break;
        case "CancelButton":
            // do something ...
            break;
    }
    e.Handled=true;
}
```

```
Private Sub CommonClickHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim feSource As FrameworkElement = TryCast(e.Source, FrameworkElement)
    Select Case feSource.Name
        Case "YesButton"
            ' do something here ...
        Case "NoButton"
            ' do something ...
        Case "CancelButton"
            ' do something ...
    End Select
    e.Handled=True
End Sub
```

**Class handling:** Routed events permit a static handler that is defined by the class. This class handler has the opportunity to handle an event before any attached instance handlers can.

**Referencing an event without reflection:** Certain code and markup techniques require a way to identify a specific event. A routed event creates a [RoutedEventArgs](#) field as an identifier, which provides a robust event identification technique that does not require static or run-time reflection.

### How Routed Events Are Implemented

A routed event is a CLR event that is backed by an instance of the [RoutedEventArgs](#) class and registered with the

WPF event system. The [RoutedEventArgs](#) instance obtained from registration is typically retained as a `public static readonly` field member of the class that registers and thus "owns" the routed event. The connection to the identically named CLR event (which is sometimes termed the "wrapper" event) is accomplished by overriding the `add` and `remove` implementations for the CLR event. Ordinarily, the `add` and `remove` are left as an implicit default that uses the appropriate language-specific event syntax for adding and removing handlers of that event. The routed event backing and connection mechanism is conceptually similar to how a dependency property is a CLR property that is backed by the [DependencyProperty](#) class and registered with the WPF property system.

The following example shows the declaration for a custom `Tap` routed event, including the registration and exposure of the [RoutedEventArgs](#) identifier field and the `add` and `remove` implementations for the `Tap` CLR event.

```
public static readonly RoutedEvent TapEvent = EventManager.RegisterRoutedEvent(
    "Tap", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(MyButtonSimple));

// Provide CLR accessors for the event
public event RoutedEventHandler Tap
{
    add { AddHandler(TapEvent, value); }
    remove { RemoveHandler(TapEvent, value); }
}
```

```
Public Shared ReadOnly TapEvent As RoutedEvent = EventManager.RegisterRoutedEvent("Tap",
    RoutingStrategy.Bubble, GetType(RoutedEventHandler), GetType(MyButtonSimple))

' Provide CLR accessors for the event
Public Custom Event Tap As RoutedEventHandler
    AddHandler(ByVal value As RoutedEventHandler)
        Me.AddHandler(TapEvent, value)
    End AddHandler

    RemoveHandler(ByVal value As RoutedEventHandler)
        Me.RemoveHandler(TapEvent, value)
    End RemoveHandler

    RaiseEvent(ByVal sender As Object, ByVal e As RoutedEventArgs)
        Me.RaiseEvent(e)
    End RaiseEvent
End Event
```

## Routed Event Handlers and XAML

To add a handler for an event using XAML, you declare the event name as an attribute on the element that is an event listener. The value of the attribute is the name of your implemented handler method, which must exist in the partial class of the code-behind file.

```
<Button Click="b1SetColor">button</Button>
```

The XAML syntax for adding standard CLR event handlers is the same for adding routed event handlers, because you are really adding handlers to the CLR event wrapper, which has a routed event implementation underneath. For more information about adding event handlers in XAML, see [XAML Overview \(WPF\)](#).

## Routing Strategies

Routed events use one of three routing strategies:

- **Bubbling:** Event handlers on the event source are invoked. The routed event then routes to successive

parent elements until reaching the element tree root. Most routed events use the bubbling routing strategy. Bubbling routed events are generally used to report input or state changes from distinct controls or other UI elements.

- **Direct:** Only the source element itself is given the opportunity to invoke handlers in response. This is analogous to the "routing" that Windows Forms uses for events. However, unlike a standard CLR event, direct routed events support class handling (class handling is explained in an upcoming section) and can be used by [EventSetter](#) and [EventTrigger](#).
- **Tunneling:** Initially, event handlers at the element tree root are invoked. The routed event then travels a route through successive child elements along the route, towards the node element that is the routed event source (the element that raised the routed event). Tunneling routed events are often used or handled as part of the compositing for a control, such that events from composite parts can be deliberately suppressed or replaced by events that are specific to the complete control. Input events provided in WPF often come implemented as a tunneling/bubbling pair. Tunneling events are also sometimes referred to as Preview events, because of a naming convention that is used for the pairs.

## Why Use Routed Events?

As an application developer, you do not always need to know or care that the event you are handling is implemented as a routed event. Routed events have special behavior, but that behavior is largely invisible if you are handling an event on the element where it is raised.

Where routed events become powerful is if you use any of the suggested scenarios: defining common handlers at a common root, compositing your own control, or defining your own custom control class.

Routed event listeners and routed event sources do not need to share a common event in their hierarchy. Any [UIElement](#) or [ContentElement](#) can be an event listener for any routed event. Therefore, you can use the full set of routed events available throughout the working API set as a conceptual "interface" whereby disparate elements in the application can exchange event information. This "interface" concept for routed events is particularly applicable for input events.

Routed events can also be used to communicate through the element tree, because the event data for the event is perpetuated to each element in the route. One element could change something in the event data, and that change would be available to the next element in the route.

Other than the routing aspect, there are two other reasons that any given WPF event might be implemented as a routed event instead of a standard CLR event. If you are implementing your own events, you might also consider these principles:

- Certain WPF styling and templating features such as [EventSetter](#) and [EventTrigger](#) require the referenced event to be a routed event. This is the event identifier scenario mentioned earlier.
- Routed events support a class handling mechanism whereby the class can specify static methods that have the opportunity to handle routed events before any registered instance handlers can access them. This is very useful in control design, because your class can enforce event-driven class behaviors that cannot be accidentally suppressed by handling an event on an instance.

Each of the above considerations is discussed in a separate section of this topic.

## Adding and Implementing an Event Handler for a Routed Event

To add an event handler in XAML, you simply add the event name to an element as an attribute and set the attribute value as the name of the event handler that implements an appropriate delegate, as in the following example.

```
<Button Click="b1SetColor">button</Button>
```

`b1SetColor` is the name of the implemented handler that contains the code that handles the [Click](#) event. `b1SetColor` must have the same signature as the [RoutedEventHandler](#) delegate, which is the event handler delegate for the [Click](#) event. The first parameter of all routed event handler delegates specifies the element to which the event handler is added, and the second parameter specifies the data for the event.

```
void b1SetColor(object sender, RoutedEventArgs args)
{
    //logic to handle the Click event
}
```

```
Private Sub b1SetColor(ByVal sender As Object, ByVal args As RoutedEventArgs)
    'logic to handle the Click event
End Sub
```

[RoutedEventHandler](#) is the basic routed event handler delegate. For routed events that are specialized for certain controls or scenarios, the delegates to use for the routed event handlers also might become more specialized, so that they can transmit specialized event data. For instance, in a common input scenario, you might handle a [DragEnter](#) routed event. Your handler should implement the [DragEventHandler](#) delegate. By using the most specific delegate, you can process the [DragEventArgs](#) in the handler and read the [Data](#) property, which contains the clipboard payload of the drag operation.

For a complete example of how to add an event handler to an element using XAML, see [Handle a Routed Event](#).

Adding a handler for a routed event in an application that is created in code is straightforward. Routed event handlers can always be added through a helper method [AddHandler](#) (which is the same method that the existing backing calls for `add`.) However, existing WPF routed events generally have backing implementations of `add` and `remove` logic that allow the handlers for routed events to be added by a language-specific event syntax, which is more intuitive syntax than the helper method. The following is an example usage of the helper method:

```
void MakeButton()
{
    Button b2 = new Button();
    b2.AddHandler(Button.ClickEvent, new RoutedEventHandler(Onb2Click));
}
void Onb2Click(object sender, RoutedEventArgs e)
{
    //logic to handle the Click event
}
```

```
Private Sub MakeButton()
    Dim b2 As New Button()
    b2.AddHandler(Button.ClickEvent, New RoutedEventHandler(AddressOf Onb2Click))
End Sub
Private Sub Onb2Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    'logic to handle the Click event
End Sub
```

The next example shows the C# operator syntax (Visual Basic has slightly different operator syntax because of its handling of dereferencing):

```

void MakeButton2()
{
    Button b2 = new Button();
    b2.Click += new RoutedEventHandler(Onb2Click2);
}
void Onb2Click2(object sender, RoutedEventArgs e)
{
    //logic to handle the Click event
}

```

```

Private Sub MakeButton2()
    Dim b2 As New Button()
    AddHandler b2.Click, AddressOf Onb2Click2
End Sub
Private Sub Onb2Click2(ByVal sender As Object, ByVal e As RoutedEventArgs)
    'logic to handle the Click event
End Sub

```

For an example of how to add an event handler in code, see [Add an Event Handler Using Code](#).

If you are using Visual Basic, you can also use the `Handles` keyword to add handlers as part of the handler declarations. For more information, see [Visual Basic and WPF Event Handling](#).

### The Concept of Handled

All routed events share a common event data base class, `RoutedEventArgs`. `RoutedEventArgs` defines the `Handled` property, which takes a Boolean value. The purpose of the `Handled` property is to enable any event handler along the route to mark the routed event as *handled*, by setting the value of `Handled` to `true`. After being processed by the handler at one element along the route, the shared event data is again reported to each listener along the route.

The value of `Handled` affects how a routed event is reported or processed as it travels further along the route. If `Handled` is `true` in the event data for a routed event, then handlers that listen for that routed event on other elements are generally no longer invoked for that particular event instance. This is true both for handlers attached in XAML and for handlers added by language-specific event handler attachment syntaxes such as `+ =` or `Handles`. For most common handler scenarios, marking an event as handled by setting `Handled` to `true` will "stop" routing for either a tunneling route or a bubbling route, and also for any event that is handled at a point in the route by a class handler.

However, there is a "handledEventsToo" mechanism whereby listeners can still run handlers in response to routed events where `Handled` is `true` in the event data. In other words, the event route is not truly stopped by marking the event data as handled. You can only use the `handledEventsToo` mechanism in code, or in an `EventSetter`:

- In code, instead of using a language-specific event syntax that works for general CLR events, call the WPF method `AddHandler(RoutedEventArgs, Delegate, Boolean)` to add your handler. Specify the value of `handledEventsToo` as `true`.
- In an `EventSetter`, set the `HandledEventsToo` attribute to be `true`.

In addition to the behavior that `Handled` state produces in routed events, the concept of `Handled` has implications for how you should design your application and write the event handler code. You can conceptualize `Handled` as being a simple protocol that is exposed by routed events. Exactly how you use this protocol is up to you, but the conceptual design for how the value of `Handled` is intended to be used is as follows:

- If a routed event is marked as handled, then it does not need to be handled again by other elements along that route.

- If a routed event is not marked as handled, then other listeners that were earlier along the route have chosen either not to register a handler, or the handlers that were registered chose not to manipulate the event data and set `Handled` to `true`. (Or, it is of course possible that the current listener is the first point in the route.) Handlers on the current listener now have three possible courses of action:
  - Take no action at all; the event remains unhandled, and the event routes to the next listener.
  - Execute code in response to the event, but make the determination that the action taken was not substantial enough to warrant marking the event as handled. The event routes to the next listener.
  - Execute code in response to the event. Mark the event as handled in the event data passed to the handler, because the action taken was deemed substantial enough to warrant marking as handled. The event still routes to the next listener, but with `Handled=true` in its event data, so only `handledEventsToo` listeners have the opportunity to invoke further handlers.

This conceptual design is reinforced by the routing behavior mentioned earlier: it is more difficult (although still possible in code or styles) to attach handlers for routed events that are invoked even if a previous handler along the route has already set `Handled` to `true`.

For more information about `Handled`, class handling of routed events, and recommendations about when it is appropriate to mark a routed event as `Handled`, see [Marking Routed Events as Handled, and Class Handling](#).

In applications, it is quite common to just handle a bubbling routed event on the object that raised it, and not be concerned with the event's routing characteristics at all. However, it is still a good practice to mark the routed event as handled in the event data, to prevent unanticipated side effects just in case an element that is further up the element tree also has a handler attached for that same routed event.

## Class Handlers

If you are defining a class that derives in some way from `DependencyObject`, you can also define and attach a class handler for a routed event that is a declared or inherited event member of your class. Class handlers are invoked before any instance listener handlers that are attached to an instance of that class, whenever a routed event reaches an element instance in its route.

Some WPF controls have inherent class handling for certain routed events. This might give the outward appearance that the routed event is not ever raised, but in reality it is being class handled, and the routed event can potentially still be handled by your instance handlers if you use certain techniques. Also, many base classes and controls expose virtual methods that can be used to override class handling behavior. For more information both on how to work around undesired class handling and on defining your own class handling in a custom class, see [Marking Routed Events as Handled, and Class Handling](#).

## Attached Events in WPF

The XAML language also defines a special type of event called an *attached event*. An attached event enables you to add a handler for a particular event to an arbitrary element. The element handling the event need not define or inherit the attached event, and neither the object potentially raising the event nor the destination handling instance must define or otherwise "own" that event as a class member.

The WPF input system uses attached events extensively. However, nearly all of these attached events are forwarded through base elements. The input events then appear as equivalent non-attached routed events that are members of the base element class. For instance, the underlying attached event `Mouse.MouseDown` can more easily be handled on any given `UIElement` by using `MouseDown` on that `UIElement` rather than dealing with attached event syntax either in XAML or code.

For more information about attached events in WPF, see [Attached Events Overview](#).

## Qualified Event Names in XAML

Another syntax usage that resembles `typename.eventname` attached event syntax but is not strictly speaking an attached event usage is when you attach handlers for routed events that are raised by child elements. You attach the handlers to a common parent, to take advantage of event routing, even though the common parent might not have the relevant routed event as a member. Consider this example again:

```
<Border Height="50" Width="300" BorderBrush="Gray" BorderThickness="1">
  <StackPanel Background="LightGray" Orientation="Horizontal" Button.Click="CommonClickHandler">
    <Button Name="YesButton" Width="Auto" >Yes</Button>
    <Button Name="NoButton" Width="Auto" >No</Button>
    <Button Name="CancelButton" Width="Auto" >Cancel</Button>
  </StackPanel>
</Border>
```

Here, the parent element listener where the handler is added is a [StackPanel](#). However, it is adding a handler for a routed event that was declared and will be raised by the [Button](#) class ([ButtonBase](#) actually, but available to [Button](#) through inheritance). [Button](#) "owns" the event, but the routed event system permits handlers for any routed event to be attached to any [UIElement](#) or [ContentElement](#) instance listener that could otherwise attach listeners for a common language runtime (CLR) event. The default xmlns namespace for these qualified event attribute names is typically the default WPF xmlns namespace, but you can also specify prefixed namespaces for custom routed events. For more information about xmlns, see [XAML Namespaces and Namespace Mapping for WPF XAML](#).

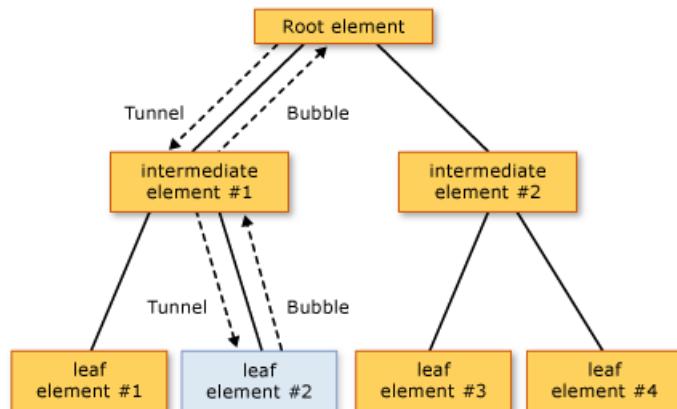
## WPF Input Events

One frequent application of routed events within the WPF platform is for input events. In WPF, tunneling routed events names are prefixed with the word "Preview" by convention. Input events often come in pairs, with one being the bubbling event and the other being the tunneling event. For example, the [KeyDown](#) event and the [PreviewKeyDown](#) event have the same signature, with the former being the bubbling input event and the latter being the tunneling input event. Occasionally, input events only have a bubbling version, or perhaps only a direct routed version. In the documentation, routed event topics cross-reference similar routed events with alternative routing strategies if such routed events exist, and sections in the managed reference pages clarify the routing strategy of each routed event.

WPF input events that come in pairs are implemented so that a single user action from input, such as a mouse button press, will raise both routed events of the pair in sequence. First, the tunneling event is raised and travels its route. Then the bubbling event is raised and travels its route. The two events literally share the same event data instance, because the [RaiseEvent](#) method call in the implementing class that raises the bubbling event listens for the event data from the tunneling event and reuses it in the new raised event. Listeners with handlers for the tunneling event have the first opportunity to mark the routed event handled (class handlers first, then instance handlers). If an element along the tunneling route marked the routed event as handled, the already-handled event data is sent on for the bubbling event, and typical handlers attached for the equivalent bubbling input events will not be invoked. To outward appearances it will be as if the handled bubbling event has not even been raised. This handling behavior is useful for control compositing, where you might want all hit-test based input events or focus-based input events to be reported by your final control, rather than its composite parts. The final control element is closer to the root in the compositing, and therefore has the opportunity to class handle the tunneling event first and perhaps to "replace" that routed event with a more control-specific event, as part of the code that backs the control class.

As an illustration of how input event processing works, consider the following input event example. In the following tree illustration, `leaf element #2` is the source of both a `PreviewMouseDown` and then a `MouseDown`

event:



The order of event processing is as follows:

1. `PreviewMouseDown` (tunnel) on root element.
2. `PreviewMouseDown` (tunnel) on intermediate element #1.
3. `PreviewMouseDown` (tunnel) on source element #2.
4. `MouseDown` (bubble) on source element #2.
5. `MouseDown` (bubble) on intermediate element #1.
6. `MouseDown` (bubble) on root element.

A routed event handler delegate provides references to two objects: the object that raised the event and the object where the handler was invoked. The object where the handler was invoked is the object reported by the `sender` parameter. The object where the event was first raised is reported by the `Source` property in the event data. A routed event can still be raised and handled by the same object, in which case `sender` and `Source` are identical (this is the case with Steps 3 and 4 in the event processing example list).

Because of tunneling and bubbling, parent elements receive input events where the `Source` is one of their child elements. When it is important to know what the source element is, you can identify the source element by accessing the `Source` property.

Usually, once the input event is marked `Handled`, further handlers are not invoked. Typically, you should mark input events as handled as soon as a handler is invoked that addresses your application-specific logical handling of the meaning of the input event.

The exception to this general statement about `Handled` state is that input event handlers that are registered to deliberately ignore `Handled` state of the event data would still be invoked along either route. For more information, see [Preview Events](#) or [Marking Routed Events as Handled, and Class Handling](#).

The shared event data model between tunneling and bubbling events, and the sequential raising of first tunneling then bubbling events, is not a concept that is generally true for all routed events. That behavior is specifically implemented by how WPF input devices choose to raise and connect the input event pairs. Implementing your own input events is an advanced scenario, but you might choose to follow that model for your own input events also.

Certain classes choose to class-handle certain input events, usually with the intent of redefining what a particular user-driven input event means within that control and raising a new event. For more information, see [Marking Routed Events as Handled, and Class Handling](#).

For more information on input and how input and events interact in typical application scenarios, see [Input Overview](#).

## EventSetters and EventTriggers

In styles, you can include some pre-declared XAML event handling syntax in the markup by using an [EventSetter](#). When the style is applied, the referenced handler is added to the styled instance. You can declare an [EventSetter](#) only for a routed event. The following is an example. Note that the `b1SetColor` method referenced here is in a code-behind file.

```
<StackPanel
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.EventOvw2"
    Name="dpanel2"
    Initialized="PrimeHandledToo"
>
    <StackPanel.Resources>
        <Style TargetType="{x:Type Button}">
            <EventSetter Event="Click" Handler="b1SetColor"/>
        </Style>
    </StackPanel.Resources>
    <Button>Click me</Button>
    <Button Name="ThisButton" Click="HandleThis">
        Raise event, handle it, use handled=true handler to get it anyway.
    </Button>
</StackPanel>
```

The advantage gained here is that the style is likely to contain a great deal of other information that could apply to any button in your application, and having the [EventSetter](#) be part of that style promotes code reuse even at the markup level. Also, an [EventSetter](#) abstracts method names for handlers one step further away from the general application and page markup.

Another specialized syntax that combines the routed event and animation features of WPF is an [EventTrigger](#). As with [EventSetter](#), only routed events may be used for an [EventTrigger](#). Typically, an [EventTrigger](#) is declared as part of a style, but an [EventTrigger](#) can also be declared on page-level elements as part of the [Triggers](#) collection, or in a [ControlTemplate](#). An [EventTrigger](#) enables you to specify a [Storyboard](#) that runs whenever a routed event reaches an element in its route that declares an [EventTrigger](#) for that event. The advantage of an [EventTrigger](#) over just handling the event and causing it to start an existing storyboard is that an [EventTrigger](#) provides better control over the storyboard and its run-time behavior. For more information, see [Use Event Triggers to Control a Storyboard After It Starts](#).

## More About Routed Events

This topic mainly discusses routed events from the perspective of describing the basic concepts and offering guidance on how and when to respond to the routed events that are already present in the various base elements and controls. However, you can create your own routed event on your custom class along with all the necessary support, such as specialized event data classes and delegates. The routed event owner can be any class, but routed events must be raised by and handled by [UIElement](#) or [ContentElement](#) derived classes in order to be useful. For more information about custom events, see [Create a Custom Routed Event](#).

## See also

- [EventManager](#)
- [RoutedEventArgs](#)
- [RoutedEventArgs](#)
- [Marking Routed Events as Handled, and Class Handling](#)
- [Input Overview](#)
- [Commanding Overview](#)

- Custom Dependency Properties
- Trees in WPF
- Weak Event Patterns

# Attached Events Overview

11/3/2019 • 6 minutes to read • [Edit Online](#)

Extensible Application Markup Language (XAML) defines a language component and type of event called an *attached event*. The concept of an attached event enables you to add a handler for a particular event to an arbitrary element rather than to an element that actually defines or inherits the event. In this case, neither the object potentially raising the event nor the destination handling instance defines or otherwise "owns" the event.

## Prerequisites

This topic assumes that you have read [Routed Events Overview](#) and [XAML Overview \(WPF\)](#).

## Attached Event Syntax

Attached events have a XAML syntax and a coding pattern that must be used by the backing code in order to support the attached event usage.

In XAML syntax, the attached event is specified not just by its event name, but by its owning type plus the event name, separated by a dot (.). Because the event name is qualified with the name of its owning type, the attached event syntax allows any attached event to be attached to any element that can be instantiated.

For example, the following is the XAML syntax for attaching a handler for a custom `NeedsCleaning` attached event:

```
<aqua:Aquarium Name="theAquarium" Height="600" Width="800" aqua:AquariumFilter.NeedsCleaning="WashMe"/>
```

Note the `aqua:` prefix; the prefix is necessary in this case because the attached event is a custom event that comes from a custom mapped xmlns.

## How WPF Implements Attached Events

In WPF, attached events are backed by a [RoutedEventArgs](#) field and are routed through the tree after they are raised. Typically, the source of the attached event (the object that raises the event) is a system or service source, and the object that runs the code that raises the event is therefore not a direct part of the element tree.

## Scenarios for Attached Events

In WPF, attached events are present in certain feature areas where there is service-level abstraction, such as for the events enabled by the static [Mouse](#) class or the [Validation](#) class. Classes that interact with or use the service can either use the event in the attached event syntax, or they can choose to surface the attached event as a routed event that is part of how the class integrates the capabilities of the service.

Although WPF defines a number of attached events, the scenarios where you will either use or handle the attached event directly are very limited. Generally, the attached event serves an architecture purpose, but is then forwarded to a non-attached (backed with a CLR event "wrapper") routed event.

For instance, the underlying attached event [Mouse.MouseDown](#) can more easily be handled on any given [UIElement](#) by using [MouseDown](#) on that [UIElement](#) rather than dealing with attached event syntax either in XAML or code. The attached event serves a purpose in the architecture because it allows for future expansion of input devices. The hypothetical device would only need to raise [Mouse.MouseDown](#) in order to simulate mouse input, and would not need to derive from [Mouse](#) to do so. However, this scenario involves code handling of the

events, and XAML handling of the attached event is not relevant to this scenario.

## Handling an Attached Event in WPF

The process for handling an attached event, and the handler code that you will write, is basically the same as for a routed event.

In general, a WPF attached event is not very different from a WPF routed event. The differences are how the event is sourced and how it is exposed by a class as a member (which also affects the XAML handler syntax).

However, as noted earlier, the existing WPF attached events are not particularly intended for handling in WPF. More often, the purpose of the event is to enable a composited element to report a state to a parent element in compositing, in which case the event is usually raised in code and also relies on class handling in the relevant parent class. For instance, items within a [Selector](#) are expected to raise the attached [Selected](#) event, which is then class handled by the [Selector](#) class and then potentially converted by the [Selector](#) class into a different routed event, [SelectionChanged](#). For more information on routed events and class handling, see [Marking Routed Events as Handled](#), and [Class Handling](#).

## Defining Your Own Attached Events as Routed Events

If you are deriving from common WPF base classes, you can implement your own attached events by including certain pattern methods in your class and by using utility methods that are already present on the base classes.

The pattern is as follows:

- A method **AddEventNameHandler** with two parameters. The first parameter is the instance to which the event handler is added. The second parameter is the event handler to add. The method must be `public` and `static`, with no return value.
- A method **RemoveEventNameHandler** with two parameters. The first parameter is the instance from which the event handler is removed. The second parameter is the event handler to remove. The method must be `public` and `static`, with no return value.

The **AddEventNameHandler** accessor method facilitates XAML processing when attached event handler attributes are declared on an element. The **AddEventNameHandler** and **RemoveEventNameHandler** methods also enable code access to the event handler store for the attached event.

This general pattern is not yet precise enough for practical implementation in a framework, because any given XAML reader implementation might have different schemes for identifying underlying events in the supporting language and architecture. This is one of the reasons that WPF implements attached events as routed events; the identifier to use for an event ([RoutedEventArgs](#)) is already defined by the WPF event system. Also, routing an event is a natural implementation extension on the XAML language-level concept of an attached event.

The **AddEventNameHandler** implementation for a WPF attached event consists of calling the [AddHandler](#) with the routed event and handler as arguments.

This implementation strategy and the routed event system in general restrict handling for attached events to either [UIElement](#) derived classes or [ContentElement](#) derived classes, because only those classes have [AddHandler](#) implementations.

For example, the following code defines the `NeedsCleaning` attached event on the owner class `Aquarium`, using the WPF attached event strategy of declaring the attached event as a routed event.

```

public static readonly RoutedEvent NeedsCleaningEvent = EventManager.RegisterRoutedEvent("NeedsCleaning",
    RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(AquariumFilter));
public static void AddNeedsCleaningHandler(DependencyObject d, RoutedEventHandler handler)
{
    UIElement uie = d as UIElement;
    if (uie != null)
    {
        uie.AddHandler(AquariumFilter.NeedsCleaningEvent, handler);
    }
}
public static void RemoveNeedsCleaningHandler(DependencyObject d, RoutedEventHandler handler)
{
    UIElement uie = d as UIElement;
    if (uie != null)
    {
        uie.RemoveHandler(AquariumFilter.NeedsCleaningEvent, handler);
    }
}

```

```

Public Shared ReadOnly NeedsCleaningEvent As RoutedEvent = EventManager.RegisterRoutedEvent("NeedsCleaning",
    RoutingStrategy.Bubble, GetType(RoutedEventHandler), GetType(AquariumFilter))
Public Shared Sub AddNeedsCleaningHandler(ByVal d As DependencyObject, ByVal handler As RoutedEventHandler)
    Dim uie As UIElement = TryCast(d, UIElement)
    If uie IsNot Nothing Then
        uie.AddHandler(AquariumFilter.NeedsCleaningEvent, handler)
    End If
End Sub
Public Shared Sub RemoveNeedsCleaningHandler(ByVal d As DependencyObject, ByVal handler As RoutedEventHandler)
    Dim uie As UIElement = TryCast(d, UIElement)
    If uie IsNot Nothing Then
        uie.RemoveHandler(AquariumFilter.NeedsCleaningEvent, handler)
    End If
End Sub

```

Note that the method used to establish the attached event identifier field, [RegisterRoutedEvent](#), is actually the same method that is used to register a non-attached routed event. Attached events and routed events all are registered to a centralized internal store. This event store implementation enables the "events as an interface" conceptual consideration that is discussed in [Routed Events Overview](#).

## Raising a WPF Attached Event

You do not typically need to raise existing WPF-defined attached events from your code. These events follow the general "service" conceptual model, and service classes such as [InputManager](#) are responsible for raising the events.

However, if you are defining a custom attached event based on the WPF model of basing attached events on [RoutedEvent](#), you can use [RaiseEvent](#) to raise an attached event from any [UIElement](#) or [ContentElement](#). Raising a routed event (attached or not) requires that you declare a particular element in the element tree as the event source; that source is reported as the [RaiseEvent](#) caller. Determining which element is reported as the source in the tree is your service's responsibility

## See also

- [Routed Events Overview](#)
- [XAML Syntax In Detail](#)
- [XAML and Custom Classes for WPF](#)

# Object Lifetime Events

11/3/2019 • 3 minutes to read • [Edit Online](#)

This topic describes the specific WPF events that signify stages in an object lifetime of creation, use, and destruction.

## Prerequisites

This topic assumes that you understand dependency properties from the perspective of a consumer of existing dependency properties on Windows Presentation Foundation (WPF) classes, and have read the [Dependency Properties Overview](#) topic. In order to follow the examples in this topic, you should also understand Extensible Application Markup Language (XAML) (see [XAML Overview \(WPF\)](#)) and know how to write WPF applications.

## Object Lifetime Events

All objects in Microsoft .NET Framework managed code go through a similar set of stages of life, creation, use, and destruction. Many objects also have a finalization stage of life that occurs as part of the destruction phase. WPF objects, more specifically the visual objects that WPF identifies as elements, also have a set of common stages of object life. The WPF programming and application models expose these stages as a series of events. There are four main types of objects in WPF with respect to lifetime events; elements in general, window elements, navigation hosts, and application objects. Windows and navigation hosts are also within the larger grouping of visual objects (elements). This topic describes the lifetime events that are common to all elements and then introduces the more specific ones that apply to application definitions, windows or navigation hosts.

## Common Lifetime Events for Elements

Any WPF framework-level element (those objects deriving from either [FrameworkElement](#) or [FrameworkContentElement](#)) has three common lifetime events: [Initialized](#), [Loaded](#), and [Unloaded](#).

### Initialized

[Initialized](#) is raised first, and roughly corresponds to the initialization of the object by the call to its constructor. Because the event happens in response to initialization, you are guaranteed that all properties of the object are set. (An exception is expression usages such as dynamic resources or binding; these will be unevaluated expressions.) As a consequence of the requirement that all properties are set, the sequence of [Initialized](#) being raised by nested elements that are defined in markup appears to occur in order of deepest elements in the element tree first, then parent elements toward the root. This order is because the parent-child relationships and containment are properties, and therefore the parent cannot report initialization until the child elements that fill the property are also completely initialized.

When you are writing handlers in response to the [Initialized](#) event, you must consider that there is no guarantee that all other elements in the element tree (either logical tree or visual tree) around where the handler is attached have been created, particularly parent elements. Member variables may be null, or data sources might not yet be populated by the underlying binding (even at the expression level).

### Loaded

[Loaded](#) is raised next. The [Loaded](#) event is raised before the final rendering, but after the layout system has calculated all necessary values for rendering. [Loaded](#) entails that the logical tree that an element is contained within is complete, and connects to a presentation source that provides the HWND and the rendering surface. Standard data binding (binding to local sources, such as other properties or directly defined data sources) will have occurred prior to [Loaded](#). Asynchronous data binding (external or dynamic sources) might have occurred, but by

definition of its asynchronous nature cannot be guaranteed to have occurred.

The mechanism by which the [Loaded](#) event is raised is different than [Initialized](#). The [Initialized](#) event is raised element by element, without a direct coordination by a completed element tree. By contrast, the [Loaded](#) event is raised as a coordinated effort throughout the entire element tree (specifically, the logical tree). When all elements in the tree are in a state where they are considered loaded, the [Loaded](#) event is first raised on the root element. The [Loaded](#) event is then raised successively on each child element.

#### NOTE

This behavior might superficially resemble tunneling for a routed event. However, no information is carried from event to event. Each element always has the opportunity to handle its [Loaded](#) event, and marking the event data as handled has no effect beyond that element.

### Unloaded

[Unloaded](#) is raised last and is initiated by either the presentation source or the visual parent being removed. When [Unloaded](#) is raised and handled, the element that is the event source parent (as determined by [Parent](#) property) or any given element upwards in the logical or visual trees may have already been unset, meaning that data binding, resource references, and styles may not be set to their normal or last known run-time value.

## Lifetime Events Application Model Elements

Building on the common lifetime events for elements are the following application model elements: [Application](#), [Window](#), [Page](#), [NavigationWindow](#), and [Frame](#). These extend the common lifetime events with additional events that are relevant to their specific purpose. These are discussed in detail in the following locations:

- [Application: Application Management Overview](#).
- [Window: WPF Windows Overview](#).
- [Page, NavigationWindow, and Frame: Navigation Overview](#).

## See also

- [Dependency Property Value Precedence](#)
- [Routed Events Overview](#)

# Marking Routed Events as Handled, and Class Handling

7/23/2019 • 14 minutes to read • [Edit Online](#)

Handlers for a routed event can mark the event handled within the event data. Handling the event will effectively shorten the route. Class handling is a programming concept that is supported by routed events. A class handler has the opportunity to handle a particular routed event at a class level with a handler that is invoked before any instance handler on any instance of the class.

## Prerequisites

This topic elaborates on concepts introduced in the [Routed Events Overview](#).

## When to Mark Events as Handled

When you set the value of the [Handled](#) property to `true` in the event data for a routed event, this is referred to as "marking the event handled". There is no absolute rule for when you should mark routed events as handled, either as an application author, or as a control author who responds to existing routed events or implements new routed events. For the most part, the concept of "handled" as carried in the routed event's event data should be used as a limited protocol for your own application's responses to the various routed events exposed in WPF APIs as well as for any custom routed events. Another way to consider the "handled" issue is that you should generally mark a routed event handled if your code responded to the routed event in a significant and relatively complete way. Typically, there should not be more than one significant response that requires separate handler implementations for any single routed event occurrence. If more responses are needed, then the necessary code should be implemented through application logic that is chained within a single handler rather than by using the routed event system for forwarding. The concept of what is "significant" is also subjective, and depends on your application or code. As general guidance, some "significant response" examples include: setting focus, modifying public state, setting properties that affect the visual representation, and raising other new events. Examples of nonsignificant responses include: modifying private state (with no visual impact, or programmatic representation), logging of events, or looking at arguments of an event and choosing not to respond to it.

The routed event system behavior reinforces this "significant response" model for using handled state of a routed event, because handlers added in XAML or the common signature of [AddHandler](#) are not invoked in response to a routed event where the event data is already marked handled. You must go through the extra effort of adding a handler with the `handledEventsToo` parameter version ([AddHandler\(RoutedEventArgs, Delegate, Boolean\)](#)) in order to handle routed events that are marked handled by earlier participants in the event route.

In some circumstances, controls themselves mark certain routed events as handled. A handled routed event represents a decision by WPF control authors that the control's actions in response to the routed event are significant or complete as part of the control implementation, and the event needs no further handling. Usually this is done by adding a class handler for an event, or by overriding one of the class handler virtuals that exist on a base class. You can still work around this event handling if necessary; see [Working Around Event Suppression by Controls](#) later in this topic.

## "Preview" (Tunneling) Events vs. Bubbling Events, and Event Handling

Preview routed events are events that follow a tunneling route through the element tree. The "Preview" expressed in the naming convention is indicative of the general principle for input events that preview (tunneling) routed events are raised prior to the equivalent bubbling routed event. Also, input routed events that

have a tunneling and bubbling pair have a distinct handling logic. If the tunneling/preview routed event is marked as handled by an event listener, then the bubbling routed event will be marked handled even before any listeners of the bubbling routed event receive it. The tunneling and bubbling routed events are technically separate events, but they deliberately share the same instance of event data to enable this behavior.

The connection between the tunneling and bubbling routed events is accomplished by the internal implementation of how any given WPF class raises its own declared routed events, and this is true of the paired input routed events. But unless this class-level implementation exists, there is no connection between a tunneling routed event and a bubbling routed event that share the naming scheme: without such implementation they would be two entirely separate routed events and would not be raised in sequence or share event data.

For more information about how to implement tunnel/bubble input routed event pairs in a custom class, see [Create a Custom Routed Event](#).

## Class Handlers and Instance Handlers

Routed events consider two different types of listeners to the event: class listeners and instance listeners. Class listeners exist because types have called a particular [EventManager API](#), [RegisterClassHandler](#), in their static constructor, or have overridden a class handler virtual method from an element base class. Instance listeners are particular class instances/elements where one or more handlers have been attached for that routed event by a call to [AddHandler](#). Existing WPF routed events make calls to [AddHandler](#) as part of the common language runtime (CLR) event wrapper add{} and remove{} implementations of the event, which is also how the simple XAML mechanism of attaching event handlers via an attribute syntax is enabled. Therefore even the simple XAML usage ultimately equates to an [AddHandler](#) call.

Elements within the visual tree are checked for registered handler implementations. Handlers are potentially invoked throughout the route, in the order that is inherent in the type of the routing strategy for that routed event. For instance, bubbling routed events will first invoke those handlers that are attached to the same element that raised the routed event. Then the routed event "bubbles" to the next parent element and so on until the application root element is reached.

From the perspective of the root element in a bubbling route, if class handling or any element closer to the source of the routed event invoke handlers that mark the event arguments as being handled, then handlers on the root elements are not invoked, and the event route is effectively shortened before reaching that root element. However, the route is not completely halted, because handlers can be added using a special conditional that they should still be invoked, even if a class handler or instance handler has marked the routed event as handled. This is explained in [Adding Instance Handlers That Are Raised Even When Events Are Marked Handled](#), later in this topic.

At a deeper level than the event route, there are also potentially multiple class handlers acting on any given instance of a class. This is because the class handling model for routed events enables all possible classes in a class hierarchy to each register its own class handler for each routed event. Each class handler is added to an internal store, and when the event route for an application is constructed, the class handlers are all added to the event route. Class handlers are added to the route such that the most-derived class handler is invoked first, and class handlers from each successive base class are invoked next. Generally, class handlers are not registered such that they also respond to routed events that were already marked handled. Therefore, this class handling mechanism enables one of two choices:

- Derived classes can supplement the class handling that is inherited from the base class by adding a handler that does not mark the routed event handled, because the base class handler will be invoked sometime after the derived class handler.
- Derived classes can replace the class handling from the base class by adding a class handler that marks the routed event handled. You should be cautious with this approach, because it will potentially change the intended base control design in areas such as visual appearance, state logic, input handling, and

command handling.

## Class Handling of Routed Events by Control Base Classes

On each given element node in an event route, class listeners have the opportunity to respond to the routed event before any instance listener on the element can. For this reason, class handlers are sometimes used to suppress routed events that a particular control class implementation does not wish to propagate further, or to provide special handling of that routed event that is a feature of the class. For instance, a class might raise its own class-specific event that contains more specifics about what some user input condition means in the context of that particular class. The class implementation might then mark the more general routed event as handled. Class handlers are typically added such that they are not invoked for routed events where shared event data was already marked handled, but for atypical cases there is also a [RegisterClassHandler\(Type, RoutedEvent, Delegate, Boolean\)](#) signature that registers class handlers to invoke even when routed events are marked handled.

### Class Handler Virtuals

Some elements, particularly the base elements such as [UIElement](#), expose empty "On\*Event" and "OnPreview\*Event" virtual methods that correspond to their list of public routed events. These virtual methods can be overridden to implement a class handler for that routed event. The base element classes register these virtual methods as their class handler for each such routed event using [RegisterClassHandler\(Type, RoutedEvent, Delegate, Boolean\)](#) as described earlier. The On\*Event virtual methods make it much simpler to implement class handling for the relevant routed events, without requiring special initialization in static constructors for each type. For instance, you can add class handling for the [DragEnter](#) event in any [UIElement](#) derived class by overriding the [OnDragEnter](#) virtual method. Within the override, you could handle the routed event, raise other events, initiate class-specific logic that might change element properties on instances, or any combination of those actions. You should generally call the base implementation in such overrides even if you mark the event handled. Calling the base implementation is strongly recommended because the virtual method is on the base class. The standard protected virtual pattern of calling the base implementations from each virtual essentially replaces and parallels a similar mechanism that is native to routed event class handling, whereby class handlers for all classes in a class hierarchy are called on any given instance, starting with the most-derived class' handler and continuing to the base class handler. You should only omit the base implementation call if your class has a deliberate requirement to change the base class handling logic. Whether you call the base implementation before or after your overriding code will depend on the nature of your implementation.

### Input Event Class Handling

The class handler virtual methods are all registered such that they are only invoked in cases where any shared event data are not already marked handled. Also, for the input events uniquely, the tunneling and bubbling versions typically are raised in sequence and share event data. This entails that for a given pair of class handlers of input events where one is the tunneling version and the other is the bubbling version, you may not want to mark the event handled immediately. If you implement the tunneling class handling virtual method to mark the event handled, that will prevent the bubbling class handler from being invoked (as well as preventing any normally registered instance handlers for either the tunneling or bubbling event from being invoked).

Once class handling on a node is complete, the instance listeners are considered.

## Adding Instance Handlers That Are Raised Even When Events Are Marked Handled

The [AddHandler](#) method supplies a particular overload that allows you to add handlers that will be invoked by the event system whenever an event reaches the handling element in the route, even if some other handler has already adjusted the event data to mark that event as handled. This is not typically done. Generally, handlers can be written to adjust all areas of application code that might be influenced by an event, regardless of where it was handled in an element tree, even if multiple end results are desired. Also, typically there is really only one element that needs to respond to that event, and the appropriate application logic had already happened. But the

`handledEventsToo` overload is available for the exceptional cases where some other element in an element tree or control compositing has already marked an event as handled, but other elements either higher or lower in the element tree (depending on route) still wish to have their own handlers invoked.

#### When to Mark Handled Events as Unhandled

Generally, routed events that are marked handled should not be marked unhandled (`Handled` set back to `false`) even by handlers that act on `handledEventsToo`. However, some input events have high-level and lower-level event representations that can overlap when the high-level event is seen at one position in the tree and the low-level event at another position. For instance, consider the case where a child element listens to a high-level key event such as `TextInput` while a parent element listens to a low-level event such as `KeyDown`. If the parent element handles the low-level event, the higher-level event can be suppressed even in the child element that intuitively should have first opportunity to handle the event.

In these situations it may be necessary to add handlers to both parent elements and child elements for the low-level event. The child element handler implementation can mark the low-level event as handled, but the parent element handler implementation would set it unhandled again so that further elements up the tree (as well as the high-level event) can have the opportunity to respond. This situation is should be fairly rare.

## Deliberately Suppressing Input Events for Control Compositing

The main scenario where class handling of routed events is used is for input events and composited controls. A composited control is by definition composed of multiple practical controls or control base classes. Often the author of the control wishes to amalgamate all of the possible input events that each of the subcomponents might raise, in order to report the entire control as the singular event source. In some cases the control author might wish to suppress the events from components entirely, or substitute a component-defined event that carries more information or implies a more specific behavior. The canonical example that is immediately visible to any component author is how a Windows Presentation Foundation (WPF) `Button` handles any mouse event that will eventually resolve to the intuitive event that all buttons have: a `Click` event.

The `Button` base class (`ButtonBase`) derives from `Control` which in turn derives from `FrameworkElement` and `UIElement`, and much of the event infrastructure needed for control input processing is available at the `UIElement` level. In particular, `UIElement` processes general `Mouse` events that handle hit testing for the mouse cursor within its bounds, and provides distinct events for the most common button actions, such as `MouseLeftButtonDown`. `UIElement` also provides an empty virtual `OnMouseLeftButtonDown` as the preregistered class handler for `MouseLeftButtonDown`, and `ButtonBase` overrides it. Similarly, `ButtonBase` uses class handlers for `MouseLeftButtonUp`. In the overrides, which are passed the event data, the implementations mark that `RoutedEventArgs` instance as handled by setting `Handled` to `true`, and that same event data is what continues along the remainder of the route to other class handlers and also to instance handlers or event setters. Also, the `OnMouseLeftButtonUp` override will next raise the `Click` event. The end result for most listeners will be that the `MouseLeftButtonDown` and `MouseLeftButtonUp` events "disappear" and are replaced instead by `Click`, an event that holds more meaning because it is known that this event originated from a true button and not some composite piece of the button or from some other element entirely.

#### Working Around Event Suppression by Controls

Sometimes this event suppression behavior within individual controls can interfere with some more general intentions of event handling logic for your application. For instance, if for some reason your application had a handler for `MouseLeftButtonDown` located at the application root element, you would notice that any mouse click on a button would not invoke `MouseLeftButtonDown` or `MouseLeftButtonUp` handlers at the root level. The event itself actually did bubble up (again, event routes are not truly ended, but the routed event system changes their handler invocation behavior after being marked handled). When the routed event reached the button, the `ButtonBase` class handling marked the `MouseLeftButtonDown` handled because it wished to substitute the `Click` event with more meaning. Therefore, any standard `MouseLeftButtonDown` handler further up the route would not be invoked. There are two techniques you can use to ensure that your handlers would be invoked in this circumstance.

The first technique is to deliberately add the handler using the `AddHandler(RoutedEvent, Delegate, Boolean)` signature of `AddHandler`. A limitation of this approach is that this technique for attaching an event handler is only possible from code, not from markup. The simple syntax of specifying the event handler name as an event attribute value via Extensible Application Markup Language (XAML) does not enable that behavior.

The second technique works only for input events, where the tunneling and bubbling versions of the routed event are paired. For these routed events, you can add handlers to the preview/tunneling equivalent routed event instead. That routed event will tunnel through the route starting from the root, so the button class handling code would not intercept it, presuming that you attached the Preview handler at some ancestor element level in the application's element tree. If you use this approach, be cautious about marking any Preview event handled. For the example given with `PreviewMouseLeftButtonDown` being handled at the root element, if you marked the event as `Handled` in the handler implementation, you would actually suppress the `Click` event. That is typically not desirable behavior.

## See also

- [EventManager](#)
- [Preview Events](#)
- [Create a Custom Routed Event](#)
- [Routed Events Overview](#)

# Preview Events

4/8/2019 • 3 minutes to read • [Edit Online](#)

Preview events, also known as tunneling events, are routed events where the direction of the route travels from the application root towards the element that raised the event and is reported as the source in event data. Not all event scenarios support or require preview events; this topic describes the situations where preview events exist, how applications or components should handle them, and cases where creating preview events in custom components or classes might be appropriate.

## Preview Events and Input

When you handle Preview events in general, be cautious about marking the events handled in the event data. Handling a Preview event on any element other than the element that raised it (the element that is reported as the source in the event data) has the effect of not providing an element the opportunity to handle the event that it originated. Sometimes this is the desired result, particularly if the elements in question exist in relationships within the compositing of a control.

For input events specifically, Preview events also share event data instances with the equivalent bubbling event. If you use a Preview event class handler to mark the input event handled, the bubbling input event class handler will not be invoked. Or, if you use a Preview event instance handler to mark the event handled, handlers for the bubbling event will not typically be invoked. Class handlers or instance handlers can be registered or attached with an option to be invoked even if the event is marked handled, but that technique is not commonly used.

For more information about class handling and how it relates to Preview events see [Marking Routed Events as Handled, and Class Handling](#).

### Working Around Event Suppression by Controls

One scenario where Preview events are commonly used is for composited control handling of input events. Sometimes, the author of the control suppresses a certain event from originating from their control, perhaps in order to substitute a component-defined event that carries more information or implies a more specific behavior. For instance, a Windows Presentation Foundation (WPF) [Button](#) suppresses [MouseLeftButtonDown](#) and [MouseRightButtonDown](#) bubbling events raised by the [Button](#) or its composite elements in favor of capturing the mouse and raising a [Click](#) event that is always raised by the [Button](#) itself. The event and its data still continue along the route, but because the [Button](#) marks the event data as [Handled](#), only handlers for the event that specifically indicated they should act in the `handledEventsToo` case are invoked. If other elements towards the root of your application still wanted an opportunity to handle a control-suppressed event, one alternative is to attach handlers in code with `handledEventsToo` specified as `true`. But often a simpler technique is to change the routing direction you handle to be the Preview equivalent of an input event. For instance, if a control suppresses [MouseLeftButtonDown](#), try attaching a handler for [PreviewMouseLeftButtonDown](#) instead. This technique only works for base element input events such as [MouseLeftButtonDown](#). These input events use tunnel/bubble pairs, raise both events, and share the event data.

Each of these techniques has either side effects or limitations. The side effect of handling the Preview event is that handling the event at that point might disable handlers that expect to handle the bubbling event, and therefore the limitation is that it is usually not a good idea to mark the event handled while it is still on the Preview part of the route. The limitation of the `handledEventsToo` technique is that you cannot specify a `handledEventsToo` handler in XAML as an attribute, you must register the event handler in code after obtaining an object reference to the element where the handler is to be attached.

## See also

- [Marking Routed Events as Handled, and Class Handling](#)
- [Routed Events Overview](#)

# Property Change Events

11/3/2019 • 5 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) defines several events that are raised in response to a change in the value of a property. Often the property is a dependency property. The event itself is sometimes a routed event and is sometimes a standard common language runtime (CLR) event. The definition of the event varies depending on the scenario, because some property changes are more appropriately routed through an element tree, whereas other property changes are generally only of concern to the object where the property changed.

## Identifying a Property Change Event

Not all events that report a property change are explicitly identified as a property changed event, either by virtue of a signature pattern or a naming pattern. Generally, the description of the event in the SDK documentation indicates whether the event is directly tied to a property value change and provides cross-references between the property and event.

### RoutedPropertyChanged Events

Certain events use an event data type and delegate that are explicitly used for property change events. The event data type is [RoutedPropertyChangedEventArgs<T>](#), and the delegate is

[RoutedPropertyChangedEventHandler<T>](#). The event data and delegate both have a generic type parameter that is used to specify the actual type of the changing property when you define the handler. The event data contains two properties, [OldValue](#) and [NewValue](#), which are both then passed as the type argument in the event data.

The "Routed" part of the name indicates that the property changed event is registered as a routed event. The advantage of routing a property changed event is that the top level of a control can receive property changed events if properties on the child elements (the control's composite parts) change values. For instance, you might create a control that incorporates a [RangeBase](#) control such as a [Slider](#). If the value of the [Value](#) property changes on the slider part, you might want to handle that change on the parent control rather than on the part.

Because you have an old value and a new value, it might be tempting to use this event handler as a validator for the property value. However, that is not the design intention of most property changed events. Generally, the values are provided so that you can act on those values in other logic areas of your code, but actually changing the values from within the event handler is not advisable, and may cause unintentional recursion depending on how your handler is implemented.

If your property is a custom dependency property, or if you are working with a derived class where you have defined the instantiation code, there is a much better mechanism for tracking property changes that is built in to the WPF property system: the property system callbacks [CoerceValueCallback](#) and [PropertyChangedCallback](#). For more details about how you can use the WPF property system for validation and coercion, see [Dependency Property Callbacks and Validation](#) and [Custom Dependency Properties](#).

### DependencyPropertyChanged Events

Another pair of types that are part of a property changed event scenario is

[DependencyPropertyChangedEventArgs](#) and [DependencyPropertyChangedEventHandler](#). Events for these property changes are not routed; they are standard CLR events. [DependencyPropertyChangedEventArgs](#) is an unusual event data reporting type because it does not derive from [EventArgs](#);

[DependencyPropertyChangedEventArgs](#) is a structure, not a class.

Events that use [DependencyPropertyChangedEventArgs](#) and [DependencyPropertyChangedEventHandler](#) are slightly more common than [RoutedPropertyChanged](#) events. An example of an event that uses these types is [IsMouseCapturedChanged](#).

Like [RoutedPropertyChangedEventArgs<T>](#), [DependencyPropertyChangedEventArgs](#) also reports an old and new value for the property. Also, the same considerations about what you can do with the values apply; it is generally not recommended that you attempt to change the values again on the sender in response to the event.

## Property Triggers

A closely related concept to a property changed event is a property trigger. A property trigger is created within a style or template and enables you to create a conditional behavior based on the value of the property where the property trigger is assigned.

The property for a property trigger must be a dependency property. It can be (and frequently is) a read-only dependency property. A good indicator for when a dependency property exposed by a control is at least partially designed to be a property trigger is if the property name begins with "Is". Properties that have this naming are often a read-only Boolean dependency property where the primary scenario for the property is reporting control state that might have consequences to the real-time UI and is thus a property trigger candidate.

Some of these properties also have a dedicated property changed event. For instance, the property [IsMouseCaptured](#) has a property changed event [IsMouseCapturedChanged](#). The property itself is read-only, with its value adjusted by the input system, and the input system raises [IsMouseCapturedChanged](#) on each real-time change.

Compared to a true property changed event, using a property trigger to act on a property change has some limitations.

Property triggers work through an exact match logic. You specify a property and a value that indicates the specific value for which the trigger will act. For instance: `<Setter Property="IsMouseCaptured" Value="true"> ... </Setter>`. Because of this limitation, the majority of property trigger usages will be for Boolean properties, or properties that take a dedicated enumeration value, where the possible value range is manageable enough to define a trigger for each case. Or property triggers might exist only for special values, such as when an items count reaches zero, and there would be no trigger that accounts for the cases when the property value changes away from zero again (instead of triggers for all cases, you might need a code event handler here, or a default behavior that toggles back from the trigger state again when the value is nonzero).

The property trigger syntax is analogous to an "if" statement in programming. If the trigger condition is true, then the "body" of the property trigger is "executed". The "body" of a property trigger is not code, it is markup. That markup is limited to using one or more [Setter](#) elements to set other properties of the object where the style or template is being applied.

To offset the "if" condition of a property trigger that has a wide variety of possible values, it is generally advisable to set that same property value to a default by using a [Setter](#). This way, the [Trigger](#) contained setter will have precedence when the trigger condition is true, and the [Setter](#) that is not within a [Trigger](#) will have precedence whenever the trigger condition is false.

Property triggers are generally appropriate for scenarios where one or more appearance properties should change, based on the state of another property on the same element.

To learn more about property triggers, see [Styling and Templating](#).

## See also

- [Routed Events Overview](#)
- [Dependency Properties Overview](#)

# Visual Basic and WPF Event Handling

11/3/2019 • 3 minutes to read • [Edit Online](#)

For the Microsoft Visual Basic .NET language specifically, you can use the language-specific `Handles` keyword to associate event handlers with instances, instead of attaching event handlers with attributes or using the `AddHandler` method. However, the `Handles` technique for attaching handlers to instances does have some limitations, because the `Handles` syntax cannot support some of the specific routed event features of the WPF event system.

## Using "Handles" in a WPF Application

The event handlers that are connected to instances and events with `Handles` must all be defined within the partial class declaration of the instance, which is also a requirement for event handlers that are assigned through attribute values on elements. You can only specify `Handles` for an element on the page that has a `Name` property value (or `x:Name Directive` declared). This is because the `Name` in XAML creates the instance reference that is necessary to support the `Instance.Event` reference format required by the `Handles` syntax. The only element that can be used for `Handles` without a `Name` reference is the root-element instance that defines the partial class.

You can assign the same handler to multiple elements by separating `Instance.Event` references after `Handles` with commas.

You can use `Handles` to assign more than one handler to the same `Instance.Event` reference. Do not assign any importance to the order in which handlers are given in the `Handles` reference; you should assume that handlers that handle the same event can be invoked in any order.

To remove a handler that was added with `Handles` in the declaration, you can call [RemoveHandler](#).

You can use `Handles` to attach handlers for routed events, so long as you attach handlers to instances that define the event being handled in their members tables. For routed events, handlers that are attached with `Handles` follow the same routing rules as do handlers that are attached as XAML attributes, or with the common signature of `AddHandler`. This means that if the event is already marked handled (the `Handled` property in the event data is `True`), then handlers attached with `Handles` are not invoked in response to that event instance. The event could be marked handled by instance handlers on another element in the route, or by class handling either on the current element or earlier elements along the route. For input events that support paired tunnel/bubble events, the tunneling route may have marked the event pair handled. For more information about routed events, see [Routed Events Overview](#).

## Limitations of "Handles" for Adding Handlers

`Handles` cannot reference handlers for attached events. You must use the `add` accessor method for that attached event, or `typename.eventname` event attributes in XAML. For details, see [Routed Events Overview](#).

For routed events, you can only use `Handles` to assign handlers for instances where that event exists in the instance members table. However, with routed events in general, a parent element can be a listener for an event from child elements, even if the parent element does not have that event in its members table. In attribute syntax, you can specify this through a `typename.membername` attribute form that qualifies which type actually defines the event you want to handle. For instance, a parent `Page` (with no `Click` event defined) can listen for button-click events by assigning an attribute handler in the form `Button.Click`. But `Handles` does not support the `typename.membername` form, because it must support a conflicting `Instance.Event` form. For details, see [Routed Events Overview](#).

`Handles` cannot attach handlers that are invoked for events that are already marked handled. Instead, you must use code and call the `handledEventsToo` overload of [AddHandler\(RoutedEvent, Delegate, Boolean\)](#).

#### NOTE

Do not use the `Handles` syntax in Visual Basic code when you specify an event handler for the same event in XAML. In this case, the event handler is called twice.

## How WPF Implements "Handles" Functionality

When a Extensible Application Markup Language (XAML) page is compiled, the intermediate file declares `Friend WithEvents` references to every element on the page that has a `Name` property set (or `x:Name Directive` declared). Each named instance is potentially an element that can be assigned to a handler through `Handles`.

#### NOTE

Within Visual Studio, IntelliSense can show you completion for which elements are available for a `Handles` reference in a page. However, this might take one compile pass so that the intermediate file can populate all the `Friends` references.

## See also

- [AddHandler](#)
- [Marking Routed Events as Handled, and Class Handling](#)
- [Routed Events Overview](#)
- [XAML Overview \(WPF\)](#)

# Weak Event Patterns

11/3/2019 • 6 minutes to read • [Edit Online](#)

In applications, it is possible that handlers that are attached to event sources will not be destroyed in coordination with the listener object that attached the handler to the source. This situation can lead to memory leaks. Windows Presentation Foundation (WPF) introduces a design pattern that can be used to address this issue, by providing a dedicated manager class for particular events and implementing an interface on listeners for that event. This design pattern is known as the *weak event pattern*.

## Why Implement the Weak Event Pattern?

Listening for events can lead to memory leaks. The typical technique for listening to an event is to use the language-specific syntax that attaches a handler to an event on a source. For example, in C#, that syntax is:

```
source.SomeEvent += new SomeEventHandler(MyEventHandler);
```

This technique creates a strong reference from the event source to the event listener. Ordinarily, attaching an event handler for a listener causes the listener to have an object lifetime that is influenced by the object lifetime of the source (unless the event handler is explicitly removed). But in certain circumstances, you might want the object lifetime of the listener to be controlled by other factors, such as whether it currently belongs to the visual tree of the application, and not by the lifetime of the source. Whenever the source object lifetime extends beyond the object lifetime of the listener, the normal event pattern leads to a memory leak: the listener is kept alive longer than intended.

The weak event pattern is designed to solve this memory leak problem. The weak event pattern can be used whenever a listener needs to register for an event, but the listener does not explicitly know when to unregister. The weak event pattern can also be used whenever the object lifetime of the source exceeds the useful object lifetime of the listener. (In this case, *useful* is determined by you.) The weak event pattern allows the listener to register for and receive the event without affecting the object lifetime characteristics of the listener in any way. In effect, the implied reference from the source does not determine whether the listener is eligible for garbage collection. The reference is a weak reference, thus the naming of the weak event pattern and the related APIs. The listener can be garbage collected or otherwise destroyed, and the source can continue without retaining noncollectible handler references to a now destroyed object.

## Who Should Implement the Weak Event Pattern?

Implementing the weak event pattern is interesting primarily for control authors. As a control author, you are largely responsible for the behavior and containment of your control and the impact it has on applications in which it is inserted. This includes the control object lifetime behavior, in particular the handling of the described memory leak problem.

Certain scenarios inherently lend themselves to the application of the weak event pattern. One such scenario is data binding. In data binding, it is common for the source object to be completely independent of the listener object, which is a target of a binding. Many aspects of WPF data binding already have the weak event pattern applied in how the events are implemented.

## How to Implement the Weak Event Pattern

There are three ways to implement weak event pattern. The following table lists the three approaches and provides some guidance for when you should use each.

APPROACH	WHEN TO IMPLEMENT
Use an existing weak event manager class	If the event you want to subscribe to has a corresponding <a href="#">WeakEventManager</a> , use the existing weak event manager. For a list of weak event managers that are included with WPF, see the inheritance hierarchy in the <a href="#">WeakEventManager</a> class. Because the included weak event managers are limited, you will probably need to choose one of the other approaches.
Use a generic weak event manager class	Use a generic <a href="#">WeakEventManager&lt;TEventSource,TEventArgs&gt;</a> when an existing <a href="#">WeakEventManager</a> is not available, you want an easy way to implement, and you are not concerned about efficiency. The generic <a href="#">WeakEventManager&lt;TEventSource,TEventArgs&gt;</a> is less efficient than an existing or custom weak event manager. For example, the generic class does more reflection to discover the event given the event's name. Also, the code to register the event by using the generic <a href="#">WeakEventManager&lt;TEventSource,TEventArgs&gt;</a> is more verbose than using an existing or custom <a href="#">WeakEventManager</a> .
Create a custom weak event manager class	Create a custom <a href="#">WeakEventManager</a> when an existing <a href="#">WeakEventManager</a> is not available and you want the best efficiency. Using a custom <a href="#">WeakEventManager</a> to subscribe to an event will be more efficient, but you do incur the cost of writing more code at the beginning.
Use a third-party weak event manager	NuGet has <a href="#">several weak event managers</a> and many WPF frameworks also support the pattern (for instance, see <a href="#">Prism's documentation on loosely coupled event subscription</a> ).

The following sections describe how to implement the weak event pattern. For purposes of this discussion, the event to subscribe to has the following characteristics.

- The event name is `SomeEvent`.
- The event is raised by the `EventSource` class.
- The event handler has type: `SomeEventEventHandler` (or `EventHandler<SomeEventEventArgs>`).
- The event passes a parameter of type `SomeEventEventArgs` to the event handlers.

## Using an Existing Weak Event Manager Class

1. Find an existing weak event manager.

For a list of weak event managers that are included with WPF, see the inheritance hierarchy in the [WeakEventManager](#) class.

2. Use the new weak event manager instead of the normal event hookup.

For example, if your code uses the following pattern to subscribe to an event:

```
source.SomeEvent += new SomeEventEventHandler(OnSomeEvent);
```

Change it to the following pattern:

```
SomeEventWeakEventManager.AddHandler(source, OnSomeEvent);
```

Similarly, if your code uses the following pattern to unsubscribe from an event:

```
source.SomeEvent -= new SomeEventEventHandler(OnSomeEvent);
```

Change it to the following pattern:

```
SomeEventWeakEventManager.RemoveHandler(source, OnSomeEvent);
```

## Using the Generic Weak Event Manager Class

1. Use the generic [WeakEventManager<TEventArgs>](#) class instead of the normal event hookup.

When you use [WeakEventManager<TEventArgs>](#) to register event listeners, you supply the event source and [EventArgs](#) type as the type parameters to the class and call [AddHandler](#) as shown in the following code:

```
WeakEventManager<EventSource, SomeEventArgs>.AddHandler(source, "SomeEvent", source_SomeEvent);
```

## Creating a Custom Weak Event Manager Class

1. Copy the following class template to your project.

This class inherits from the [WeakEventManager](#) class.

```
class SomeEventWeakEventManager : WeakEventManager
{
    private SomeEventWeakEventManager()
    {
    }

    /// <summary>
    /// Add a handler for the given source's event.
    /// </summary>
    public static void AddHandler(EventSource source,
                                  EventHandler<SomeEventArgs> handler)
    {
        if (source == null)
            throw new ArgumentNullException("source");
        if (handler == null)
            throw new ArgumentNullException("handler");

        CurrentManager.ProtectedAddHandler(source, handler);
    }

    /// <summary>
    /// Remove a handler for the given source's event.
    /// </summary>
    public static void RemoveHandler(EventSource source,
                                     EventHandler<SomeEventArgs> handler)
    {
        if (source == null)
            throw new ArgumentNullException("source");
        if (handler == null)
            throw new ArgumentNullException("handler");
    }
}
```

```

        CurrentManager.ProtectedRemoveHandler(source, handler);
    }

    /// <summary>
    /// Get the event manager for the current thread.
    /// </summary>
    private static SomeEventWeakEventManager CurrentManager
    {
        get
        {
            Type managerType = typeof(SomeEventWeakEventManager);
            SomeEventWeakEventManager manager =
                (SomeEventWeakEventManager)GetCurrentManager(managerType);

            // at first use, create and register a new manager
            if (manager == null)
            {
                manager = new SomeEventWeakEventManager();
                SetCurrentManager(managerType, manager);
            }

            return manager;
        }
    }

    /// <summary>
    /// Return a new list to hold listeners to the event.
    /// </summary>
    protected override ListenerList NewListenerList()
    {
        return new ListenerList<SomeEventEventArgs>();
    }

    /// <summary>
    /// Listen to the given source for the event.
    /// </summary>
    protected override void StartListening(object source)
    {
        EventSource typedSource = (EventSource)source;
        typedSource.SomeEvent += new EventHandler<SomeEventEventArgs>(OnSomeEvent);
    }

    /// <summary>
    /// Stop listening to the given source for the event.
    /// </summary>
    protected override void StopListening(object source)
    {
        EventSource typedSource = (EventSource)source;
        typedSource.SomeEvent -= new EventHandler<SomeEventEventArgs>(OnSomeEvent);
    }

    /// <summary>
    /// Event handler for the SomeEvent event.
    /// </summary>
    void OnSomeEvent(object sender, SomeEventEventArgs e)
    {
        DeliverEvent(sender, e);
    }
}

```

2. Replace the `SomeEventWeakEventManager` name with your own name.
3. Replace the three names described previously with the corresponding names for your event. (`SomeEvent`, `EventSource`, and `SomeEventEventArgs`)

4. Set the visibility (public / internal / private) of the weak event manager class to the same visibility as event it manages.
5. Use the new weak event manager instead of the normal event hookup.

For example, if your code uses the following pattern to subscribe to an event:

```
source.SomeEvent += new SomeEventEventHandler(OnSomeEvent);
```

Change it to the following pattern:

```
SomeEventWeakEventManager.AddHandler(source, OnSomeEvent);
```

Similarly, if your code uses the following pattern to unsubscribe to an event:

```
source.SomeEvent -= new SomeEventEventHandler(OnSome);
```

Change it to the following pattern:

```
SomeEventWeakEventManager.RemoveHandler(source, OnSomeEvent);
```

## See also

- [WeakEventManager](#)
- [IWeakEventListener](#)
- [Routed Events Overview](#)
- [Data Binding Overview](#)

# Events How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to use events in WPF.

## In This Section

[Add an Event Handler Using Code](#)

[Handle a Routed Event](#)

[Create a Custom Routed Event](#)

[Find the Source Element in an Event Handler](#)

[Add Class Handling for a Routed Event](#)

## Reference

[RoutedEventArgs](#)

[EventManager](#)

[RoutingStrategy](#)

## Related Sections

# How to: Add an Event Handler Using Code

11/3/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to add an event handler to an element by using code.

If you want to add an event handler to a XAML element, and the markup page that contains the element has already been loaded, you must add the handler using code. Alternatively, if you are building up the element tree for an application entirely using code and not declaring any elements using XAML, you can call specific methods to add event handlers to the constructed element tree.

## Example

The following example adds a new [Button](#) to an existing page that is initially defined in XAML. A code-behind file implements an event handler method and then adds that method as a new event handler on the [Button](#).

The C# example uses the `+=` operator to assign a handler to an event. This is the same operator that is used to assign a handler in the common language runtime (CLR) event handling model. Microsoft Visual Basic does not support this operator as a means of adding event handlers. It instead requires one of two techniques:

- Use the [AddHandler](#) method, together with an `Addressof` operator, to reference the event handler implementation.
- Use the `Handles` keyword as part of the event handler definition. This technique is not shown here; see [Visual Basic and WPF Event Handling](#).

```
<TextBlock Name="text1">Start by clicking the button below</TextBlock>
<Button Name="b1" Click="MakeButton">Make new button and add handler to it</Button>
```

```
public partial class RoutedEventAddRemoveHandler {
    void MakeButton(object sender, RoutedEventArgs e)
    {
        Button b2 = new Button();
        b2.Content = "New Button";
        // Associate event handler to the button. You can remove the event
        // handler using "-=" syntax rather than "+".
        b2.Click += new RoutedEventHandler(Onb2Click);
        root.Children.Insert(root.Children.Count, b2);
        DockPanel.SetDock(b2, Dock.Top);
        text1.Text = "Now click the second button...";
        b1.IsEnabled = false;
    }
    void Onb2Click(object sender, RoutedEventArgs e)
    {
        text1.Text = "New Button (b2) Was Clicked!!";
    }
}
```

```
Public Partial Class RoutedEventAddRemoveHandler
    Private Sub MakeButton(ByVal sender As Object, ByVal e As RoutedEventArgs)
        Dim b2 As Button = New Button()
        b2.Content = "New Button"
        AddHandler b2.Click, AddressOf Onb2Click
        root.Children.Insert(root.Children.Count, b2)
        DockPanel.SetDock(b2, Dock.Top)
        text1.Text = "Now click the second button..."
        b1.IsEnabled = False
    End Sub
    Private Sub Onb2Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
        text1.Text = "New Button (b2) Was Clicked!!"
    End Sub
```

#### NOTE

Adding an event handler in the initially parsed XAML page is much simpler. Within the object element where you want to add the event handler, add an attribute that matches the name of the event that you want to handle. Then specify the value of that attribute as the name of the event handler method that you defined in the code-behind file of the XAML page. For more information, see [XAML Overview \(WPF\)](#) or [Routed Events Overview](#).

## See also

- [Routed Events Overview](#)
- [How-to Topics](#)

# How to: Handle a Routed Event

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how bubbling events work and how to write a handler that can process the routed event data.

## Example

In Windows Presentation Foundation (WPF), elements are arranged in an element tree structure. The parent element can participate in the handling of events that are initially raised by child elements in the element tree. This is possible because of event routing.

Routed events typically follow one of two routing strategies, bubbling or tunneling. This example focuses on the bubbling event and uses the [ButtonBase.Click](#) event to show how routing works.

The following example creates two [Button](#) controls and uses XAML attribute syntax to attach an event handler to a common parent element, which in this example is [StackPanel](#). Instead of attaching individual event handlers for each [Button](#) child element, the example uses attribute syntax to attach the event handler to the [StackPanel](#) parent element. This event-handling pattern shows how to use event routing as a technique for reducing the number of elements where a handler is attached. All the bubbling events for each [Button](#) route through the parent element.

Note that on the parent [StackPanel](#) element, the [Click](#) event name specified as the attribute is partially qualified by naming the [Button](#) class. The [Button](#) class is a [ButtonBase](#) derived class that has the [Click](#) event in its members listing. This partial qualification technique for attaching an event handler is necessary if the event that is being handled does not exist in the members listing of the element where the routed event handler is attached.

```
<StackPanel  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    x:Class="SDKSample.RoutedEventHandle"  
    Name="dpanel"  
    Button.Click="HandleClick"  
>  
    <StackPanel.Resources>  
        <Style TargetType="{x:Type Button}">  
            <Setter Property="Height" Value="20"/>  
            <Setter Property="Width" Value="250"/>  
            <Setter Property="HorizontalAlignment" Value="Left"/>  
        </Style>  
    </StackPanel.Resources>  
    <Button Name="Button1">Item 1</Button>  
    <Button Name="Button2">Item 2</Button>  
    <TextBlock Name="results"/>  
</StackPanel>
```

The following example handles the [Click](#) event. The example reports which element handles the event and which element raises the event. The event handler is executed when the user clicks either button.

```

public partial class RoutedEventHandle : StackPanel
{
    StringBuilder eventstr = new StringBuilder();
    void HandleClick(object sender, RoutedEventArgs args)
    {
        // Get the element that handled the event.
        FrameworkElement fe = (FrameworkElement)sender;
        eventstr.Append("Event handled by element named ");
        eventstr.Append(fe.Name);
        eventstr.Append("\n");

        // Get the element that raised the event.
        FrameworkElement fe2 = (FrameworkElement)args.Source;
        eventstr.Append("Event originated from source element of type ");
        eventstr.Append(args.Source.GetType().ToString());
        eventstr.Append(" with Name ");
        eventstr.Append(fe2.Name);
        eventstr.Append("\n");

        // Get the routing strategy.
        eventstr.Append("Event used routing strategy ");
        eventstr.Append(args.RoutedEventArgs.RoutingStrategy);
        eventstr.Append("\n");
    }

    results.Text = eventstr.ToString();
}
}

```

```

Private eventstr As New Text.StringBuilder()

Private Sub HandleClick(ByVal sender As Object, ByVal args As RoutedEventArgs)
    ' Get the element that handled the event.
    Dim fe As FrameworkElement = DirectCast(sender, FrameworkElement)
    eventstr.Append("Event handled by element named ")
    eventstr.Append(fe.Name)
    eventstr.Append(vbLf)

    ' Get the element that raised the event.
    Dim fe2 As FrameworkElement = DirectCast(args.Source, FrameworkElement)
    eventstr.Append("Event originated from source element of type ")
    eventstr.Append(args.Source.[GetType]().ToString())
    eventstr.Append(" with Name ")
    eventstr.Append(fe2.Name)
    eventstr.Append(vbLf)

    ' Get the routing strategy.
    eventstr.Append("Event used routing strategy ")
    eventstr.Append(args.RoutedEventArgs.RoutingStrategy)
    eventstr.Append(vbLf)

    results.Text = eventstr.ToString()
End Sub

```

## See also

- [RoutedEvent](#)
- [Input Overview](#)
- [Routed Events Overview](#)
- [How-to Topics](#)
- [XAML Syntax In Detail](#)

# How to: Create a Custom Routed Event

7/23/2019 • 2 minutes to read • [Edit Online](#)

For your custom event to support event routing, you need to register a [RoutedEventArgs](#) using the [RegisterRoutedEvent](#) method. This example demonstrates the basics of creating a custom routed event.

## Example

As shown in the following example, you first register a [RoutedEventArgs](#) using the [RegisterRoutedEvent](#) method. By convention, the [RoutedEventArgs](#) static field name should end with the suffix **Event**. In this example, the name of the event is `Tap` and the routing strategy of the event is [Bubble](#). After the registration call, you can provide add-and-remove common language runtime (CLR) event accessors for the event.

Note that even though the event is raised through the `OnTap` virtual method in this particular example, how you raise your event or how your event responds to changes depends on your needs.

Note also that this example basically implements an entire subclass of [Button](#); that subclass is built as a separate assembly and then instantiated as a custom class on a separate Extensible Application Markup Language (XAML) page. This is to illustrate the concept that subclassed controls can be inserted into trees composed of other controls, and that in this situation, custom events on these controls have the very same event routing capabilities as any native Windows Presentation Foundation (WPF) element does.

```
public class MyButtonSimple: Button
{
    // Create a custom routed event by first registering a RoutedEventArgsID
    // This event uses the bubbling routing strategy
    public static readonly RoutedEventArgsID TapEvent = EventManager.RegisterRoutedEvent(
        "Tap", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(MyButtonSimple));

    // Provide CLR accessors for the event
    public event RoutedEventHandler Tap
    {
        add { AddHandler(TapEvent, value); }
        remove { RemoveHandler(TapEvent, value); }
    }

    // This method raises the Tap event
    void RaiseTapEvent()
    {
        RoutedEventArgs newEventArgs = new RoutedEventArgs(MyButtonSimple.TapEvent);
        RaiseEvent(newEventArgs);
    }

    // For demonstration purposes we raise the event when the MyButtonSimple is clicked
    protected override void OnClick()
    {
        RaiseTapEvent();
    }
}
```

```

Public Class MyButtonSimple
    Inherits Button

    ' Create a custom routed event by first registering a RoutedEventID
    ' This event uses the bubbling routing strategy
    Public Shared ReadOnly TapEvent As RoutedEvent = EventManager.RegisterRoutedEvent("Tap",
        RoutingStrategy.Bubble, GetType(RoutedEventHandler), GetType(MyButtonSimple))

    ' Provide CLR accessors for the event
    Public Custom Event Tap As RoutedEventHandler
        AddHandler(ByVal value As RoutedEventHandler)
            Me.AddHandler(TapEvent, value)
        End AddHandler

        RemoveHandler(ByVal value As RoutedEventHandler)
            Me.RemoveHandler(TapEvent, value)
        End RemoveHandler

        RaiseEvent(ByVal sender As Object, ByVal e As RoutedEventArgs)
            Me.RaiseEvent(e)
        End RaiseEvent
    End Event

    ' This method raises the Tap event
    Private Sub RaiseTapEvent()
        Dim newEventArgs As New RoutedEventArgs(MyButtonSimple.TapEvent)
        MyBase.RaiseEvent(newEventArgs)
    End Sub

    ' For demonstration purposes we raise the event when the MyButtonSimple is clicked
    Protected Overrides Sub OnClick()
        Me.RaiseTapEvent()
    End Sub
End Class

```

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary"
    x:Class="SDKSample.RoutedEventCustomApp"

    >
    <Window.Resources>
        <Style TargetType="{x:Type custom:MyButtonSimple}">
            <Setter Property="Height" Value="20"/>
            <Setter Property="Width" Value="250"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="Background" Value="#808080"/>
        </Style>
    </Window.Resources>
    <StackPanel Background="LightGray">
        <custom:MyButtonSimple Name="mybtntsimple" Tap="TapHandler">Click to see Tap custom event
        work</custom:MyButtonSimple>
    </StackPanel>
</Window>

```

Tunneling events are created the same way, but with [RoutingStrategy](#) set to [Tunnel](#) in the registration call. By convention, tunneling events in WPF are prefixed with the word "Preview".

To see an example of how bubbling events work, see [Handle a Routed Event](#).

## See also

- [Routed Events Overview](#)
- [Input Overview](#)
- [Control Authoring Overview](#)

# How to: Find the Source Element in an Event Handler

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to find the source element in an event handler.

## Example

The following example shows a [Click](#) event handler that is declared in a code-behind file. When a user clicks the button that the handler is attached to, the handler changes a property value. The handler code uses the [Source](#) property of the routed event data that is reported in the event arguments to change the [Width](#) property value on the [Source](#) element.

```
<Button Click="HandleClick">Button 1</Button>
```

```
void HandleClick(object sender, RoutedEventArgs e)
{
    // You must cast the sender object as a Button element, or at least as FrameworkElement, to set Width
    Button srcButton = e.Source as Button;
    srcButton.Width = 200;
}
```

```
Private Sub HandleClick(ByVal sender As Object, ByVal e As RoutedEventArgs)
    'You must cast the object as a Button element, or at least as FrameworkElement, to set Width
    Dim srcButton As Button
    srcButton = CType(e.Source, Button)
    srcButton.Width = 200
End Sub
```

## See also

- [RoutedEventArgs](#)
- [Routed Events Overview](#)
- [How-to Topics](#)

# How to: Add Class Handling for a Routed Event

4/8/2019 • 2 minutes to read • [Edit Online](#)

Routed events can be handled either by class handlers or instance handlers on any given node in the route. Class handlers are invoked first, and can be used by class implementations to suppress events from instance handling or introduce other event specific behaviors on events that are owned by base classes. This example illustrates two closely related techniques for implementing class handlers.

## Example

This example uses a custom class based on the [Canvas](#) panel. The basic premise of the application is that the custom class introduces behaviors on its child elements, including intercepting any left mouse button clicks and marking them handled, before the child element class or any instance handlers on it will be invoked.

The [UIElement](#) class exposes a virtual method that enables class handling on the [PreviewMouseLeftButtonDown](#) event, by simply overriding the event. This is the simplest way to implement class handling if such a virtual method is available somewhere in your class' hierarchy. The following code shows the [OnPreviewMouseLeftButtonDown](#) implementation in the "MyEditContainer" that is derived from [Canvas](#). The implementation marks the event as handled in the arguments, and then adds some code to give the source element a basic visible change.

```
protected override void OnPreviewMouseRightButtonDown(System.Windows.Input.MouseButtonEventArgs e)
{
    e.Handled = true; //suppress the click event and other leftmousebuttondown responders
    MyEditContainer ec = (MyEditContainer)e.Source;
    if (ec.EditState)
        { ec.EditState = false; }
    else
        { ec.EditState = true; }
    base.OnPreviewMouseRightButtonDown(e);
}
```

```
Protected Overrides Sub OnPreviewMouseRightButtonDown(ByVal e As System.Windows.Input.MouseButtonEventArgs)
    e.Handled = True 'suppress the click event and other leftmousebuttondown responders
    Dim ec As MyEditContainer = CType(e.Source, MyEditContainer)
    If ec.EditState Then
        ec.EditState = False
    Else
        ec.EditState = True
    End If
    MyBase.OnPreviewMouseRightButtonDown(e)
End Sub
```

If no virtual is available on base classes or for that particular method, class handling can be added directly using a utility method of the [EventManager](#) class, [RegisterClassHandler](#). This method should only be called within the static initialization of classes that are adding class handling. This example adds another handler for [PreviewMouseLeftButtonDown](#), and in this case the registered class is the custom class. In contrast, when using the virtuals, the registered class is really the [UIElement](#) base class. In cases where base classes and subclass each register class handling, the subclass handlers are invoked first. The behavior in an application would be that first this handler would show its message box and then the visual change in the virtual method's handler would be shown.

```
static MyEditContainer()
{
    EventManager.RegisterClassHandler(typeof(MyEditContainer), PreviewMouseRightButtonDownEvent, new
RoutedEventHandler(LocalOnMouseRightButtonDown));
}
internal static void LocalOnMouseRightButtonDown(object sender, RoutedEventArgs e)
{
    MessageBox.Show("this is invoked before the On* class handler on UIElement");
    //e.Handled = true; //uncommenting this would cause ONLY the subclass' class handler to respond
}
```

```
Shared Sub New()
    EventManager.RegisterClassHandler(GetType(MyEditContainer), PreviewMouseRightButtonDownEvent, New
RoutedEventHandler(AddressOf LocalOnMouseRightButtonDown))
End Sub
Friend Shared Sub LocalOnMouseRightButtonDown(ByVal sender As Object, ByVal e As RoutedEventArgs)
    MessageBox.Show("this is invoked before the On* class handler on UIElement")
    'e.Handled = True //uncommenting this would cause ONLY the subclass' class handler to respond
End Sub
```

## See also

- [EventManager](#)
- [Marking Routed Events as Handled, and Class Handling](#)
- [Handle a Routed Event](#)
- [Routed Events Overview](#)

# Input (WPF)

3/5/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) includes support for several types of input. This input includes text, touch, mouse, commands, focus, touch, drag-and-drop, and digital ink. This section describes topics related to input in WPF.

## In This Section

[Input Overview](#)

[Commanding Overview](#)

[Focus Overview](#)

[Styling for Focus in Controls, and FocusVisualStyle](#)

[Walkthrough: Creating Your First Touch Application](#)

[How-to Topics](#)

[Digital Ink](#)

## Reference

[UIElement](#)

[FrameworkElement](#)

[ContentElement](#)

[FrameworkContentElement](#)

[Keyboard](#)

[Mouse](#)

[FocusManager](#)

## Related Sections

[Controls](#)

[Events](#)

# Input Overview

10/7/2019 • 25 minutes to read • [Edit Online](#)

The Windows Presentation Foundation (WPF) subsystem provides a powerful API for obtaining input from a variety of devices, including the mouse, keyboard, touch, and stylus. This topic describes the services provided by WPF and explains the architecture of the input systems.

## Input API

The primary input API exposure is found on the base element classes: [UIElement](#), [ContentElement](#), [FrameworkElement](#), and [FrameworkContentElement](#). For more information about the base elements, see [Base Elements Overview](#). These classes provide functionality for input events related to key presses, mouse buttons, mouse wheel, mouse movement, focus management, and mouse capture, to name a few. By placing the input API on the base elements, rather than treating all input events as a service, the input architecture enables the input events to be sourced by a particular object in the UI, and to support an event routing scheme whereby more than one element has an opportunity to handle an input event. Many input events have a pair of events associated with them. For example, the key down event is associated with the [KeyDown](#) and [PreviewKeyDown](#) events. The difference in these events is in how they are routed to the target element. Preview events tunnel down the element tree from the root element to the target element. Bubbling events bubble up from the target element to the root element. Event routing in WPF is discussed in more detail later in this overview and in the [Routed Events Overview](#).

### Keyboard and Mouse Classes

In addition to the input API on the base element classes, the [Keyboard](#) class and [Mouse](#) classes provide additional API for working with keyboard and mouse input.

Examples of input API on the [Keyboard](#) class are the [Modifiers](#) property, which returns the [ModifierKeys](#) currently pressed, and the [IsKeyDown](#) method, which determines whether a specified key is pressed.

The following example uses the [GetKeyStates](#) method to determine if a [Key](#) is in the down state.

```
// Uses the Keyboard.GetKeyStates to determine if a key is down.  
// A bitwise AND operation is used in the comparison.  
// e is an instance of KeyEventArgs.  
if ((Keyboard.GetKeyStates(Key.Return) & KeyStates.Down) > 0)  
{  
    btnNone.Background = Brushes.Red;  
}
```

```
' Uses the Keyboard.GetKeyStates to determine if a key is down.  
' A bitwise AND operation is used in the comparison.  
' e is an instance of KeyEventArgs.  
If (Keyboard.GetKeyStates(Key.Return) And KeyStates.Down) > 0 Then  
    btnNone.Background = Brushes.Red
```

Examples of input API on the [Mouse](#) class are [MiddleButton](#), which obtains the state of the middle mouse button, and [DirectlyOver](#), which gets the element the mouse pointer is currently over.

The following example determines whether the [LeftButton](#) on the mouse is in the [Pressed](#) state.

```
if (Mouse.LeftButton == MouseButtonState.Pressed)
{
    UpdateSampleResults("Left Button Pressed");
}
```

```
If Mouse.LeftButton = MouseButtonState.Pressed Then
    UpdateSampleResults("Left Button Pressed")
End If
```

The [Mouse](#) and [Keyboard](#) classes are covered in more detail throughout this overview.

### Stylus Input

WPF has integrated support for the [Stylus](#). The [Stylus](#) is a pen input made popular by the Tablet PC. WPF applications can treat the stylus as a mouse by using the mouse API, but WPF also exposes a stylus device abstraction that uses a model similar to the keyboard and mouse. All stylus-related APIs contain the word "Stylus".

Because the stylus can act as a mouse, applications that support only mouse input can still obtain some level of stylus support automatically. When the stylus is used in such a manner, the application is given the opportunity to handle the appropriate stylus event and then handles the corresponding mouse event. In addition, higher-level services such as ink input are also available through the stylus device abstraction. For more information about ink as input, see [Getting Started with Ink](#).

## Event Routing

A [FrameworkElement](#) can contain other elements as child elements in its content model, forming a tree of elements. In WPF, the parent element can participate in input directed to its child elements or other descendants by handing events. This is especially useful for building controls out of smaller controls, a process known as "control composition" or "compositing." For more information about element trees and how element trees relate to event routes, see [Trees in WPF](#).

Event routing is the process of forwarding events to multiple elements, so that a particular object or element along the route can choose to offer a significant response (through handling) to an event that might have been sourced by a different element. Routed events use one of three routing mechanisms: direct, bubbling, and tunneling. In direct routing, the source element is the only element notified, and the event is not routed to any other elements. However, the direct routed event still offers some additional capabilities that are only present for routed events as opposed to standard CLR events. Bubbling works up the element tree by first notifying the element that sourced the event, then the parent element, and so on. Tunneling starts at the root of the element tree and works down, ending with the original source element. For more information about routed events, see [Routed Events Overview](#).

WPF input events generally come in pairs that consists of a tunneling event and a bubbling event. Tunneling events are distinguished from bubbling events with the "Preview" prefix. For instance, [PreviewMouseMove](#) is the tunneling version of a mouse move event and [MouseMove](#) is the bubbling version of this event. This event pairing is a convention that is implemented at the element level and is not an inherent capability of the WPF event system. For details, see the WPF Input Events section in [Routed Events Overview](#).

## Handling Input Events

To receive input on an element, an event handler must be associated with that particular event. In XAML this is straightforward: you reference the name of the event as an attribute of the element that will be listening for this event. Then, you set the value of the attribute to the name of the event handler that you define, based on a delegate. The event handler must be written in code such as C# and can be included in a code-behind file.

Keyboard events occur when the operating system reports key actions that occur while keyboard focus is on an element. Mouse and stylus events each fall into two categories: events that report changes in pointer position relative to the element, and events that report changes in the state of device buttons.

### Keyboard Input Event Example

The following example listens for a left arrow key press. A [StackPanel](#) is created that has a [Button](#). An event handler to listen for the left arrow key press is attached to the [Button](#) instance.

The first section of the example creates the [StackPanel](#) and the [Button](#) and attaches the event handler for the [KeyDown](#).

```
<StackPanel>
    <Button Background="AliceBlue"
        KeyDown="OnButtonKeyDown"
        Content="Button1"/>
</StackPanel>
```

```
// Create the UI elements.
StackPanel keyboardStackPanel = new StackPanel();
Button keyboardButton1 = new Button();

// Set properties on Buttons.
keyboardButton1.Background = Brushes.AliceBlue;
keyboardButton1.Content = "Button 1";

// Attach Buttons to StackPanel.
keyboardStackPanel.Children.Add(keyboardButton1);

// Attach event handler.
keyboardButton1.KeyDown += new KeyEventHandler(OnButtonKeyDown);
```

```
' Create the UI elements.
Dim keyboardStackPanel As New StackPanel()
Dim keyboardButton1 As New Button()

' Set properties on Buttons.
keyboardButton1.Background = Brushes.AliceBlue
keyboardButton1.Content = "Button 1"

' Attach Buttons to StackPanel.
keyboardStackPanel.Children.Add(keyboardButton1)

' Attach event handler.
AddHandler keyboardButton1.KeyDown, AddressOf OnButtonKeyDown
```

The second section is written in code and defines the event handler. When the left arrow key is pressed and the [Button](#) has keyboard focus, the handler runs and the [Background](#) color of the [Button](#) is changed. If the key is pressed, but it is not the left arrow key, the [Background](#) color of the [Button](#) is changed back to its starting color.

```

private void OnButtonKeyDown(object sender, KeyEventArgs e)
{
    Button source = e.Source as Button;
    if (source != null)
    {
        if (e.Key == Key.Left)
        {
            source.Background = Brushes.LemonChiffon;
        }
        else
        {
            source.Background = Brushes.AliceBlue;
        }
    }
}

```

```

Private Sub OnButtonKeyDown(ByVal sender As Object, ByVal e As KeyEventArgs)
    Dim source As Button = TryCast(e.Source, Button)
    If source IsNot Nothing Then
        If e.Key = Key.Left Then
            source.Background = Brushes.LemonChiffon
        Else
            source.Background = Brushes.AliceBlue
        End If
    End If
End Sub

```

## Mouse Input Event Example

In the following example, the **Background** color of a **Button** is changed when the mouse pointer enters the **Button**. The **Background** color is restored when the mouse leaves the **Button**.

The first section of the example creates the **StackPanel** and the **Button** control and attaches the event handlers for the **MouseEnter** and **MouseLeave** events to the **Button**.

```

<StackPanel>
    <Button Background="AliceBlue"
        MouseEnter="OnMouseExampleMouseEnter"
        MouseLeave="OnMosueExampleMouseLeave">Button

    </Button>
</StackPanel>

```

```

// Create the UI elements.
StackPanel mouseMoveStackPanel = new StackPanel();
Button mouseMoveButton = new Button();

// Set properties on Button.
mouseMoveButton.Background = Brushes.AliceBlue;
mouseMoveButton.Content = "Button";

// Attach Buttons to StackPanel.
mouseMoveStackPanel.Children.Add(mouseMoveButton);

// Attach event handler.
mouseMoveButton.MouseEnter += new MouseEventHandler(OnMouseExampleMouseEnter);
mouseMoveButton.MouseLeave += new MouseEventHandler(OnMosueExampleMouseLeave);

```

```

' Create the UI elements.
Dim mouseMoveStackPanel As New StackPanel()
Dim mouseMoveButton As New Button()

' Set properties on Button.
mouseMoveButton.Background = Brushes.AliceBlue
mouseMoveButton.Content = "Button"

' Attach Buttons to StackPanel.
mouseMoveStackPanel.Children.Add(mouseMoveButton)

' Attach event handler.
AddHandler mouseMoveButton.MouseEnter, AddressOf OnMouseExampleMouseEnter
AddHandler mouseMoveButton.MouseLeave, AddressOf OnMosueExampleMouseLeave

```

The second section of the example is written in code and defines the event handlers. When the mouse enters the **Button**, the **Background** color of the **Button** is changed to **SlateGray**. When the mouse leaves the **Button**, the **Background** color of the **Button** is changed back to **AliceBlue**.

```

private void OnMouseExampleMouseEnter(object sender, MouseEventArgs e)
{
    // Cast the source of the event to a Button.
    Button source = e.Source as Button;

    // If source is a Button.
    if (source != null)
    {
        source.Background = Brushes.SlateGray;
    }
}

```

```

Private Sub OnMouseExampleMouseEnter(ByVal sender As Object, ByVal e As MouseEventArgs)
    ' Cast the source of the event to a Button.
    Dim source As Button = TryCast(e.Source, Button)

    ' If source is a Button.
    If source IsNot Nothing Then
        source.Background = Brushes.SlateGray
    End If
End Sub

```

```

private void OnMosueExampleMouseLeave(object sender, MouseEventArgs e)
{
    // Cast the source of the event to a Button.
    Button source = e.Source as Button;

    // If source is a Button.
    if (source != null)
    {
        source.Background = Brushes.AliceBlue;
    }
}

```

```

Private Sub OnMouseExampleMouseLeave(ByVal sender As Object, ByVal e As MouseEventArgs)
    ' Cast the source of the event to a Button.
    Dim source As Button = TryCast(e.Source, Button)

    ' If source is a Button.
    If source IsNot Nothing Then
        source.Background = Brushes.AliceBlue
    End If
End Sub

```

## Text Input

The [TextInput](#) event enables you to listen for text input in a device-independent manner. The keyboard is the primary means of text input, but speech, handwriting, and other input devices can generate text input also.

For keyboard input, WPF first sends the appropriate [KeyDown/KeyUp](#) events. If those events are not handled and the key is textual (rather than a control key such as directional arrows or function keys), then a [TextInput](#) event is raised. There is not always a simple one-to-one mapping between [KeyDown/KeyUp](#) and [TextInput](#) events because multiple keystrokes can generate a single character of text input and single keystrokes can generate multi-character strings. This is especially true for languages such as Chinese, Japanese, and Korean which use Input Method Editors (IMEs) to generate the thousands of possible characters in their corresponding alphabets.

When WPF sends a [KeyUp/KeyDown](#) event, [Key](#) is set to [Key.System](#) if the keystrokes could become part of a [TextInput](#) event (if ALT+S is pressed, for example). This allows code in a [KeyDown](#) event handler to check for [Key.System](#) and, if found, leave processing for the handler of the subsequently raised [TextInput](#) event. In these cases, the various properties of the [TextCompositionEventArgs](#) argument can be used to determine the original keystrokes. Similarly, if an IME is active, [Key](#) has the value of [Key.ImeProcessed](#), and [ImeProcessedKey](#) gives the original keystroke or keystrokes.

The following example defines a handler for the [Click](#) event and a handler for the [KeyDown](#) event.

The first segment of code or markup creates the user interface.

```

<StackPanel KeyDown="OnTextInputKeyDown">
    <Button Click="OnTextInputButtonClick"
            Content="Open" />
    <TextBox> . . . </TextBox>
</StackPanel>

```

```

// Create the UI elements.
StackPanel textInputStackPanel = new StackPanel();
Button textInputButton = new Button();
TextBox textInputTextBox = new TextBox();
textInputButton.Content = "Open";

// Attach elements to StackPanel.
textInputStackPanel.Children.Add(textInputButton);
textInputStackPanel.Children.Add(textInputTextBox);

// Attach event handlers.
textInputStackPanel.KeyDown += new KeyEventHandler(OnTextInputKeyDown);
textInputButton.Click += new RoutedEventHandler(OnTextInputButtonClick);

```

```

' Create the UI elements.
Dim textInputStackPanel As New StackPanel()
Dim textInputButton As New Button()
Dim textInputTextBox As New TextBox()
textInputButton.Content = "Open"

' Attach elements to StackPanel.
textInputStackPanel.Children.Add(textInputButton)
textInputStackPanel.Children.Add(textInputTextBox)

' Attach event handlers.
AddHandler textInputStackPanel.KeyDown, AddressOf OnTextInputKeyDown
AddHandler textInputButton.Click, AddressOf OnTextInputButtonClick

```

The second segment of code contains the event handlers.

```

private void OnTextInputKeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.O && Keyboard.Modifiers == ModifierKeys.Control)
    {
        handle();
        e.Handled = true;
    }
}

private void OnTextInputButtonClick(object sender, RoutedEventArgs e)
{
    handle();
    e.Handled = true;
}

public void handle()
{
    MessageBox.Show("Pretend this opens a file");
}

```

```

Private Sub OnTextInputKeyDown(ByVal sender As Object, ByVal e As KeyEventArgs)
    If e.Key = Key.O AndAlso Keyboard.Modifiers = ModifierKeys.Control Then
        handle()
        e.Handled = True
    End If
End Sub

Private Sub OnTextInputButtonClick(ByVal sender As Object, ByVal e As RoutedEventArgs)
    handle()
    e.Handled = True
End Sub

Public Sub handle()
    MessageBox.Show("Pretend this opens a file")
End Sub

```

Because input events bubble up the event route, the [StackPanel](#) receives the input regardless of which element has keyboard focus. The [TextBox](#) control is notified first and the [OnTextInputKeyDown](#) handler is called only if the [TextBox](#) did not handle the input. If the [PreviewKeyDown](#) event is used instead of the [KeyDown](#) event, the [OnTextInputKeyDown](#) handler is called first.

In this example, the handling logic is written two times—one time for CTRL+O, and again for button's click event. This can be simplified by using commands, instead of handling the input events directly. Commands are discussed in this overview and in [Commanding Overview](#).

# Touch and Manipulation

New hardware and API in the Windows 7 operating system provide applications the ability to receive input from multiple touches simultaneously. WPF enables applications to detect and respond to touch in a manner similar to responding to other input, such as the mouse or keyboard, by raising events when touch occurs.

WPF exposes two types of events when touch occurs: touch events and manipulation events. Touch events provide raw data about each finger on a touchscreen and its movement. Manipulation events interpret the input as certain actions. Both types of events are discussed in this section.

## Prerequisites

You need the following components to develop an application that responds to touch.

- Visual Studio 2010.
- Windows 7.
- A device, such as a touchscreen, that supports Windows Touch.

## Terminology

The following terms are used when touch is discussed.

- **Touch** is a type of user input that is recognized by Windows 7. Usually, touch is initiated by putting fingers on a touch-sensitive screen. Note that devices such as a touchpad that is common on laptop computers do not support touch if the device merely converts the finger's position and movement as mouse input.
- **Multitouch** is touch that occurs from more than one point simultaneously. Windows 7 and WPF supports multitouch. Whenever touch is discussed in the documentation for WPF, the concepts apply to multitouch.
- A **manipulation** occurs when touch is interpreted as a physical action that is applied to an object. In WPF, manipulation events interpret input as a translation, expansion, or rotation manipulation.
- A `touch device` represents a device that produces touch input, such as a single finger on a touchscreen.

## Controls that Respond to Touch

The following controls can be scrolled by dragging a finger across the control if it has content that is scrolled out of view.

- [ComboBox](#)
- [ContextMenu](#)
- [DataGrid](#)
- [ListBox](#)
- [ListView](#)
- [MenuItem](#)
- [TextBox](#)
- [ToolBar](#)
- [TreeView](#)

The [ScrollViewer](#) defines the [ScrollViewer.PanningMode](#) attached property that enables you to specify whether touch panning is enabled horizontally, vertically, both, or neither. The [ScrollViewer.PanningDeceleration](#) property specifies how quickly the scrolling slows down when the user lifts the finger from the touchscreen. The

`ScrollViewer.PanningRatio` attached property specifies the ratio of scrolling offset to translate manipulation offset.

## Touch Events

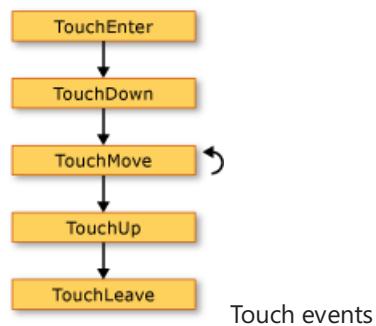
The base classes, `UIElement`, `UIElement3D`, and `ContentElement`, define events that you can subscribe to so your application will respond to touch. Touch events are useful when your application interprets touch as something other than manipulating an object. For example, an application that enables a user to draw with one or more fingers would subscribe to touch events.

All three classes define the following events, which behave similarly, regardless of the defining class.

- `TouchDown`
- `TouchMove`
- `TouchUp`
- `TouchEnter`
- `TouchLeave`
- `PreviewTouchDown`
- `PreviewTouchMove`
- `PreviewTouchUp`
- `GotTouchCapture`
- `LostTouchCapture`

Like keyboard and mouse events, the touch events are routed events. The events that begin with `Preview` are tunneling events and the events that begin with `Touch` are bubbling events. For more information about routed events, see [Routed Events Overview](#). When you handle these events, you can get the position of the input, relative to any element, by calling the `GetTouchPoint` or `GetIntermediateTouchPoints` method.

To understand the interaction among the touch events, consider the scenario where a user puts one finger on an element, moves the finger in the element, and then lifts the finger from the element. The following illustration shows the execution of the bubbling events (the tunneling events are omitted for simplicity).



Touch events

The following list describes the sequence of the events in the preceding illustration.

1. The `TouchEnter` event occurs one time when the user puts a finger on the element.
2. The `TouchDown` event occurs one time.
3. The `TouchMove` event occurs multiple times as the user moves the finger within the element.
4. The `TouchUp` event occurs one time when the user lifts the finger from the element.
5. The `TouchLeave` event occurs one time.

When more than two fingers are used, the events occur for each finger.

## Manipulation Events

For cases where an application enables a user to manipulate an object, the [UIElement](#) class defines manipulation events. Unlike the touch events that simply report the position of touch, the manipulation events report how the input can be interpreted. There are three types of manipulations, translation, expansion, and rotation. The following list describes how to invoke the three types of manipulations.

- Put a finger on an object and move the finger across the touchscreen to invoke a translation manipulation. This usually moves the object.
- Put two fingers on an object and move the fingers closer together or farther apart from one another to invoke an expansion manipulation. This usually resizes the object.
- Put two fingers on an object and rotate the fingers around each other to invoke a rotation manipulation. This usually rotates the object.

More than one type of manipulation can occur simultaneously.

When you cause objects to respond to manipulations, you can have the object appear to have inertia. This can make your objects simulate the physical world. For example, when you push a book across a table, if you push hard enough the book will continue to move after you release it. WPF enables you to simulate this behavior by raising manipulation events after the user's fingers releases the object.

For information about how to create an application that enables the user to move, resize, and rotate an object, see [Walkthrough: Creating Your First Touch Application](#).

The [UIElement](#) defines the following manipulation events.

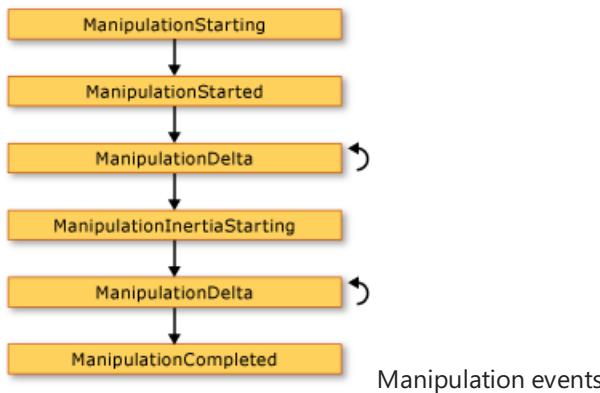
- [ManipulationStarting](#)
- [ManipulationStarted](#)
- [ManipulationDelta](#)
- [ManipulationInertiaStarting](#)
- [ManipulationCompleted](#)
- [ManipulationBoundaryFeedback](#)

By default, a [UIElement](#) does not receive these manipulation events. To receive manipulation events on a [UIElement](#), set [UIElement.IsManipulationEnabled](#) to `true`.

### The Execution Path of Manipulation Events

Consider a scenario where a user "throws" an object. The user puts a finger on the object, moves the finger across the touchscreen for a short distance, and then lifts the finger while it is moving. The result of this is that the object will move under the user's finger and continue to move after the user lifts the finger.

The following illustration shows the execution path of manipulation events and important information about each event.



The following list describes the sequence of the events in the preceding illustration.

1. The [ManipulationStarting](#) event occurs when the user places a finger on the object. Among other things, this event allows you to set the [ManipulationContainer](#) property. In the subsequent events, the position of the manipulation will be relative to the [ManipulationContainer](#). In events other than [ManipulationStarting](#), this property is read-only, so the [ManipulationStarting](#) event is the only time that you can set this property.
2. The [ManipulationStarted](#) event occurs next. This event reports the origin of the manipulation.
3. The [ManipulationDelta](#) event occurs multiple times as a user's fingers move on a touchscreen. The [DeltaManipulation](#) property of the [ManipulationDeltaEventArgs](#) class reports whether the manipulation is interpreted as movement, expansion, or translation. This is where you perform most of the work of manipulating an object.
4. The [ManipulationInertiaStarting](#) event occurs when the user's fingers lose contact with the object. This event enables you to specify the deceleration of the manipulations during inertia. This is so your object can emulate different physical spaces or attributes if you choose. For example, suppose your application has two objects that represent items in the physical world, and one is heavier than the other. You can make the heavier object decelerate faster than the lighter object.
5. The [ManipulationDelta](#) event occurs multiple times as inertia occurs. Note that this event occurs when the user's fingers move across the touchscreen and when WPF simulates inertia. In other words, [ManipulationDelta](#) occurs before and after the [ManipulationInertiaStarting](#) event. The [ManipulationDeltaEventArgs.IsInertial](#) property reports whether the [ManipulationDelta](#) event occurs during inertia, so you can check that property and perform different actions, depending on its value.
6. The [ManipulationCompleted](#) event occurs when the manipulation and any inertia ends. That is, after all the [ManipulationDelta](#) events occur, the [ManipulationCompleted](#) event occurs to signal that the manipulation is complete.

The [UIElement](#) also defines the [ManipulationBoundaryFeedback](#) event. This event occurs when the [ReportBoundaryFeedback](#) method is called in the [ManipulationDelta](#) event. The [ManipulationBoundaryFeedback](#) event enables applications or components to provide visual feedback when an object hits a boundary. For example, the [Window](#) class handles the [ManipulationBoundaryFeedback](#) event to cause the window to slightly move when its edge is encountered.

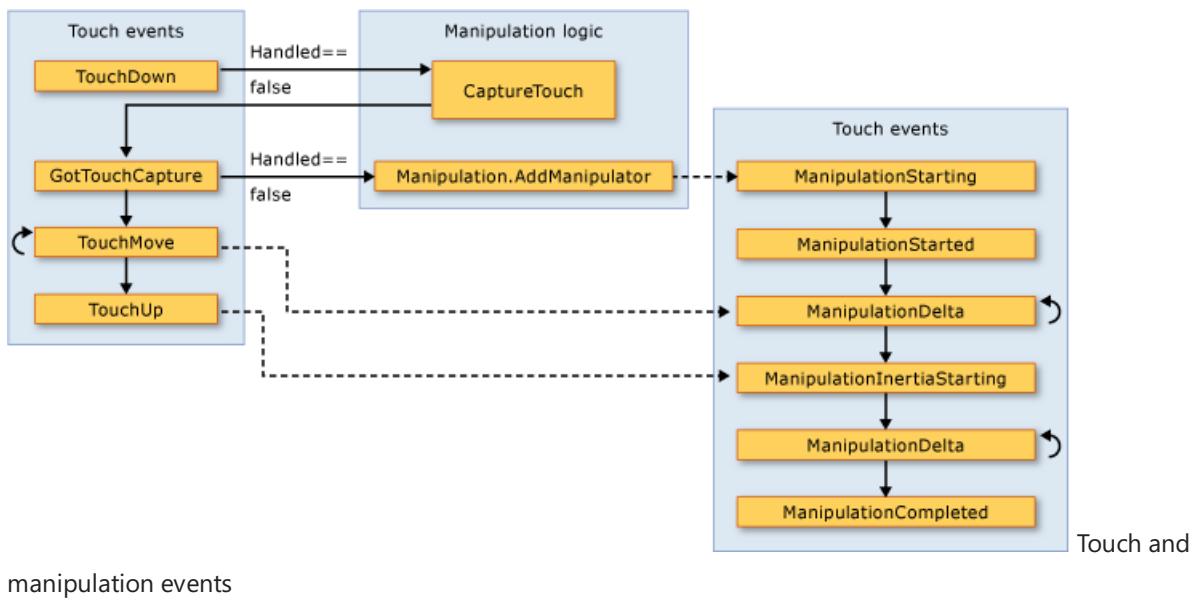
You can cancel the manipulation by calling the [Cancel](#) method on the event arguments in any manipulation event except [ManipulationBoundaryFeedback](#) event. When you call [Cancel](#), the manipulation events are no longer raised and mouse events occur for touch. The following table describes the relationship between the time the manipulation is canceled and the mouse events that occur.

THE EVENT THAT CANCEL IS CALLED IN	THE MOUSE EVENTS THAT OCCUR FOR INPUT THAT ALREADY OCCURRED
ManipulationStarting and ManipulationStarted	Mouse down events.
ManipulationDelta	Mouse down and mouse move events.
ManipulationInertiaStarting and ManipulationCompleted	Mouse down, mouse move, and mouse up events.

Note that if you call `Cancel` when the manipulation is in inertia, the method returns `false` and the input does not raise mouse events.

### The Relationship Between Touch and Manipulation Events

A `UIElement` can always receive touch events. When the `IsManipulationEnabled` property is set to `true`, a `UIElement` can receive both touch and manipulation events. If the `TouchDown` event is not handled (that is, the `Handled` property is `false`), the manipulation logic captures the touch to the element and generates the manipulation events. If the `Handled` property is set to `true` in the `TouchDown` event, the manipulation logic does not generate manipulation events. The following illustration shows the relationship between touch events and manipulation events.



manipulation events

The following list describes the relationship between the touch and manipulation events that is shown in the preceding illustration.

- When the first touch device generates a `TouchDown` event on a `UIElement`, the manipulation logic calls the `CaptureTouch` method, which generates the `GotTouchCapture` event.
- When the `GotTouchCapture` occurs, the manipulation logic calls the `Manipulation.AddManipulator` method, which generates the `ManipulationStarting` event.
- When the `TouchMove` events occur, the manipulation logic generates the `ManipulationDelta` events that occur before the `ManipulationInertiaStarting` event.
- When the last touch device on the element raises the `TouchUp` event, the manipulation logic generates the `ManipulationInertiaStarting` event.

## Focus

There are two main concepts that pertain to focus in WPF: keyboard focus and logical focus.

### Keyboard Focus

Keyboard focus refers to the element that is receiving keyboard input. There can be only one element on the whole desktop that has keyboard focus. In WPF, the element that has keyboard focus will have `IsKeyboardFocused` set to `true`. The static `Keyboard` method `FocusedElement` returns the element that currently has keyboard focus.

Keyboard focus can be obtained by tabbing to an element or by clicking the mouse on certain elements, such as a `TextBox`. Keyboard focus can also be obtained programmatically by using the `Focus` method on the `Keyboard` class. `Focus` attempts to give the specified element keyboard focus. The element returned by `Focus` is the element that currently has keyboard focus.

In order for an element to obtain keyboard focus the `Focusable` property and the `isVisible` properties must be set to `true`. Some classes, such as `Panel`, have `Focusable` set to `false` by default; therefore, you may have to set this property to `true` if you want that element to be able to obtain focus.

The following example uses `Focus` to set keyboard focus on a `Button`. The recommended place to set initial focus in an application is in the `Loaded` event handler.

```
private void OnLoaded(object sender, RoutedEventArgs e)
{
    // Sets keyboard focus on the first Button in the sample.
    Keyboard.Focus(firstButton);
}
```

```
Private Sub OnLoaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Sets keyboard focus on the first Button in the sample.
    Keyboard.Focus(firstButton)
End Sub
```

For more information about keyboard focus, see [Focus Overview](#).

## Logical Focus

Logical focus refers to the `FocusManager.FocusedElement` in a focus scope. There can be multiple elements that have logical focus in an application, but there may only be one element that has logical focus in a particular focus scope.

A focus scope is a container element that keeps track of the `FocusedElement` within its scope. When focus leaves a focus scope, the focused element will lose keyboard focus but will retain logical focus. When focus returns to the focus scope, the focused element will obtain keyboard focus. This allows for keyboard focus to be changed between multiple focus scopes but insures that the focused element within the focus scope remains the focused element when focus returns.

An element can be turned into a focus scope in Extensible Application Markup Language (XAML) by setting the `FocusManager` attached property `IsFocusScope` to `true`, or in code by setting the attached property by using the `SetIsFocusScope` method.

The following example makes a `StackPanel` into a focus scope by setting the `IsFocusScope` attached property.

```
<StackPanel Name="focusScope1"
            FocusManager.IsFocusScope="True"
            Height="200" Width="200">
    <Button Name="button1" Height="50" Width="50"/>
    <Button Name="button2" Height="50" Width="50"/>
</StackPanel>
```

```
StackPanel focusScope2 = new StackPanel();
FocusManager.SetIsFocusScope(focusScope2, true);
```

```
Dim focusScope2 As New StackPanel()
FocusManager.SetIsFocusScope(focusScope2, True)
```

Classes in WPF which are focus scopes by default are [Window](#), [Menu](#), [ToolBar](#), and [ContextMenu](#).

An element that has keyboard focus will also have logical focus for the focus scope it belongs to; therefore, setting focus on an element with the [Focus](#) method on the [Keyboard](#) class or the base element classes will attempt to give the element keyboard focus and logical focus.

To determine the focused element in a focus scope, use [GetFocusedElement](#). To change the focused element for a focus scope, use [SetFocusedElement](#).

For more information about logical focus, see [Focus Overview](#).

## Mouse Position

The WPF input API provides helpful information with regard to coordinate spaces. For example, coordinate `(0,0)` is the upper-left coordinate, but the upper-left of which element in the tree? The element that is the input target? The element you attached your event handler to? Or something else? To avoid confusion, the WPF input API requires that you specify your frame of reference when you work with coordinates obtained through the mouse. The [GetPosition](#) method returns the coordinate of the mouse pointer relative to the specified element.

## Mouse Capture

Mouse devices specifically hold a modal characteristic known as mouse capture. Mouse capture is used to maintain a transitional input state when a drag-and-drop operation is started, so that other operations involving the nominal on-screen position of the mouse pointer do not necessarily occur. During the drag, the user cannot click without aborting the drag-and-drop, which makes most mouseover cues inappropriate while the mouse capture is held by the drag origin. The input system exposes APIs that can determine mouse capture state, as well as APIs that can force mouse capture to a specific element, or clear mouse capture state. For more information on drag-and-drop operations, see [Drag and Drop Overview](#).

## Commands

Commands enable input handling at a more semantic level than device input. Commands are simple directives, such as `Cut`, `Copy`, `Paste`, or `Open`. Commands are useful for centralizing your command logic. The same command might be accessed from a [Menu](#), on a [ToolBar](#), or through a keyboard shortcut. Commands also provide a mechanism for disabling controls when the command becomes unavailable.

[RoutedCommand](#) is the WPF implementation of  [ICommand](#). When a [RoutedCommand](#) is executed, a [PreviewExecuted](#) and an [Executed](#) event are raised on the command target, which tunnel and bubble through the element tree like other input. If a command target is not set, the element with keyboard focus will be the command target. The logic that performs the command is attached to a [CommandBinding](#). When an [Executed](#) event reaches a [CommandBinding](#) for that specific command, the [ExecutedRoutedEventHandler](#) on the [CommandBinding](#) is called. This handler performs the action of the command.

For more information on commanding, see [Commanding Overview](#).

WPF provides a library of common commands which consists of [ApplicationCommands](#), [MediaCommands](#), [ComponentCommands](#), [NavigationCommands](#), and [EditingCommands](#), or you can define your own.

The following example shows how to set up a [MenuItem](#) so that when it is clicked it will invoke the [Paste](#) command on the [TextBox](#), assuming the [TextBox](#) has keyboard focus.

```
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Paste" />
    </Menu>
    <TextBox />
</StackPanel>
```

```
// Creating the UI objects
StackPanel mainStackPanel = new StackPanel();
TextBox pasteTextBox = new TextBox();
Menu stackPanelMenu = new Menu();
MenuItem pasteMenuItem = new MenuItem();

// Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem);
mainStackPanel.Children.Add(stackPanelMenu);
mainStackPanel.Children.Add(pasteTextBox);

// Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste;

// Setting the command target to the TextBox
pasteMenuItem.CommandTarget = pasteTextBox;
```

```
' Creating the UI objects
Dim mainStackPanel As New StackPanel()
Dim pasteTextBox As New TextBox()
Dim stackPanelMenu As New Menu()
Dim pasteMenuItem As New MenuItem()

' Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem)
mainStackPanel.Children.Add(stackPanelMenu)
mainStackPanel.Children.Add(pasteTextBox)

' Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste
```

For more information about commands in WPF, see [Commanding Overview](#).

## The Input System and Base Elements

Input events such as the attached events defined by the [Mouse](#), [Keyboard](#), and [Stylus](#) classes are raised by the input system and injected into a particular position in the object model based on hit testing the visual tree at run time.

Each of the events that [Mouse](#), [Keyboard](#), and [Stylus](#) define as an attached event is also re-exposed by the base element classes [UIElement](#) and [ContentElement](#) as a new routed event. The base element routed events are generated by classes handling the original attached event and reusing the event data.

When the input event becomes associated with a particular source element through its base element input event implementation, it can be routed through the remainder of an event route that is based on a combination of logical and visual tree objects, and be handled by application code. Generally, it is more convenient to handle these device-related input events using the routed events on [UIElement](#) and [ContentElement](#), because you can use more intuitive event handler syntax both in XAML and in code. You could choose to handle the attached event that initiated the process instead, but you would face several issues: the attached event may be marked

handled by the base element class handling, and you need to use accessor methods rather than true event syntax in order to attach handlers for attached events.

## What's Next

You now have several techniques to handle input in WPF. You should also have an improved understanding of the various types of input events and the routed event mechanisms used by WPF.

Additional resources are available that explain WPF framework elements and event routing in more detail. See the following overviews for more information, [Commanding Overview](#), [Focus Overview](#), [Base Elements Overview](#), [Trees in WPF](#), and [Routed Events Overview](#).

## See also

- [Focus Overview](#)
- [Commanding Overview](#)
- [Routed Events Overview](#)
- [Base Elements Overview](#)
- [Properties](#)

# Commanding Overview

10/7/2019 • 13 minutes to read • [Edit Online](#)

Commanding is an input mechanism in Windows Presentation Foundation (WPF) which provides input handling at a more semantic level than device input. Examples of commands are the **Copy**, **Cut**, and **Paste** operations found on many applications.

This overview defines what commands are in WPF, which classes are part of the commanding model, and how to use and create commands in your applications.

This topic contains the following sections:

- [What Are Commands?](#)
- [Simple Command Example in WPF](#)
- [Four Main Concepts in WPF Commanding](#)
- [Command Library](#)
- [Creating Custom Commands](#)

## What Are Commands?

Commands have several purposes. The first purpose is to separate the semantics and the object that invokes a command from the logic that executes the command. This allows for multiple and disparate sources to invoke the same command logic, and it allows the command logic to be customized for different targets. For example, the editing operations **Copy**, **Cut**, and **Paste**, which are found in many applications, can be invoked by using different user actions if they are implemented by using commands. An application might allow a user to cut selected objects or text by either clicking a button, choosing an item in a menu, or using a key combination, such as CTRL+X. By using commands, you can bind each type of user action to the same logic.

Another purpose of commands is to indicate whether an action is available. To continue the example of cutting an object or text, the action only makes sense when something is selected. If a user tries to cut an object or text without having anything selected, nothing would happen. To indicate this to the user, many applications disable buttons and menu items so that the user knows whether it is possible to perform an action. A command can indicate whether an action is possible by implementing the [CanExecute](#) method. A button can subscribe to the [CanExecuteChanged](#) event and be disabled if [CanExecute](#) returns `false` or be enabled if [CanExecute](#) returns `true`.

The semantics of a command can be consistent across applications and classes, but the logic of the action is specific to the particular object acted upon. The key combination CTRL+X invokes the **Cut** command in text classes, image classes, and Web browsers, but the actual logic for performing the **Cut** operation is defined by the application that performs the cut. A [RoutedCommand](#) enables clients to implement the logic. A text object may cut the selected text into the clipboard, while an image object may cut the selected image. When an application handles the [Executed](#) event, it has access to the target of the command and can take appropriate action depending on the target's type.

## Simple Command Example in WPF

The simplest way to use a command in WPF is to use a predefined [RoutedCommand](#) from one of the command library classes; use a control that has native support for handling the command; and use a control that has native support for invoking a command. The [Paste](#) command is one of the predefined commands in

the [ApplicationCommands](#) class. The [TextBox](#) control has built in logic for handling the [Paste](#) command. And the [MenuItem](#) class has native support for invoking commands.

The following example shows how to set up a [MenuItem](#) so that when it is clicked it will invoke the [Paste](#) command on a [TextBox](#), assuming the [TextBox](#) has keyboard focus.

```
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Paste" />
    </Menu>
    <TextBox />
</StackPanel>
```

```
// Creating the UI objects
StackPanel mainStackPanel = new StackPanel();
TextBox pasteTextBox = new TextBox();
Menu stackPanelMenu = new Menu();
MenuItem pasteMenuItem = new MenuItem();

// Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem);
mainStackPanel.Children.Add(stackPanelMenu);
mainStackPanel.Children.Add(pasteTextBox);

// Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste;

// Setting the command target to the TextBox
pasteMenuItem.CommandTarget = pasteTextBox;
```

```
' Creating the UI objects
Dim mainStackPanel As New StackPanel()
Dim pasteTextBox As New TextBox()
Dim stackPanelMenu As New Menu()
Dim pasteMenuItem As New MenuItem()

' Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem)
mainStackPanel.Children.Add(stackPanelMenu)
mainStackPanel.Children.Add(pasteTextBox)

' Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste
```

## Four Main Concepts in WPF Commanding

The routed command model in WPF can be broken up into four main concepts: the command, the command source, the command target, and the command binding:

- The *command* is the action to be executed.
- The *command source* is the object which invokes the command.
- The *command target* is the object that the command is being executed on.
- The *command binding* is the object which maps the command logic to the command.

In the previous example, the [Paste](#) command is the command, the [MenuItem](#) is the command source, the [TextBox](#) is the command target, and the command binding is supplied by the [TextBox](#) control. It is worth noting that it is not always the case that the [CommandBinding](#) is supplied by the control that is the command target

class. Quite often the [CommandBinding](#) must be created by the application developer, or the [CommandBinding](#) might be attached to an ancestor of the command target.

## Commands

Commands in WPF are created by implementing the [ICommand](#) interface. [ICommand](#) exposes two methods, [Execute](#), and [CanExecute](#), and an event, [CanExecuteChanged](#). [Execute](#) performs the actions that are associated with the command. [CanExecute](#) determines whether the command can execute on the current command target. [CanExecuteChanged](#) is raised if the command manager that centralizes the commanding operations detects a change in the command source that might invalidate a command that has been raised but not yet executed by the command binding. The WPF implementation of [ICommand](#) is the [RoutedCommand](#) class and is the focus of this overview.

The main sources of input in WPF are the mouse, the keyboard, ink, and routed commands. The more device-oriented inputs use a [RoutedEventArgs](#) to notify objects in an application page that an input event has occurred. A [RoutedCommand](#) is no different. The [Execute](#) and [CanExecute](#) methods of a [RoutedCommand](#) do not contain the application logic for the command, but rather they raise routed events that tunnel and bubble through the element tree until they encounter an object with a [CommandBinding](#). The [CommandBinding](#) contains the handlers for these events and it is the handlers that perform the command. For more information on event routing in WPF, see [Routed Events Overview](#).

The [Execute](#) method on a [RoutedCommand](#) raises the [PreviewExecuted](#) and the [Executed](#) events on the command target. The [CanExecute](#) method on a [RoutedCommand](#) raises the [CanExecute](#) and [PreviewCanExecute](#) events on the command target. These events tunnel and bubble through the element tree until they encounter an object which has a [CommandBinding](#) for that particular command.

WPF supplies a set of common routed commands spread across several classes: [MediaCommands](#), [ApplicationCommands](#), [NavigationCommands](#), [ComponentCommands](#), and [EditingCommands](#). These classes consist only of the [RoutedCommand](#) objects and not the implementation logic of the command. The implementation logic is the responsibility of the object on which the command is being executed on.

## Command Sources

A command source is the object which invokes the command. Examples of command sources are [MenuItem](#), [Button](#), and [KeyGesture](#).

Command sources in WPF generally implement the [ICommandSource](#) interface.

[ICommandSource](#) exposes three properties: [Command](#), [CommandTarget](#), and [CommandParameter](#):

- [Command](#) is the command to execute when the command source is invoked.
- [CommandTarget](#) is the object on which to execute the command. It is worth noting that in WPF the [CommandTarget](#) property on [ICommandSource](#) is only applicable when the [ICommand](#) is a [RoutedCommand](#). If the [CommandTarget](#) is set on an [ICommandSource](#) and the corresponding command is not a [RoutedCommand](#), the command target is ignored. If the [CommandTarget](#) is not set, the element with keyboard focus will be the command target.
- [CommandParameter](#) is a user-defined data type used to pass information to the handlers implementing the command.

The WPF classes that implement [ICommandSource](#) are [ButtonBase](#), [MenuItem](#), [Hyperlink](#), and [InputBinding](#). [ButtonBase](#), [MenuItem](#), and [Hyperlink](#) invoke a command when they are clicked, and an [InputBinding](#) invokes a command when the [InputGesture](#) associated with it is performed.

The following example shows how to use a [MenuItem](#) in a [ContextMenu](#) as a command source for the [Properties](#) command.

```

<StackPanel>
  <StackPanel.ContextMenu>
    <ContextMenu>
      <MenuItem Command="ApplicationCommands.Properties" />
    </ContextMenu>
  </StackPanel.ContextMenu>
</StackPanel>

```

```

StackPanel cmdSourcePanel = new StackPanel();
ContextMenu cmdSourceContextMenu = new ContextMenu();
MenuItem cmdSourceMenuItem = new MenuItem();

// Add ContextMenu to the StackPanel.
cmdSourcePanel.ContextMenu = cmdSourceContextMenu;
cmdSourcePanel.ContextMenu.Items.Add(cmdSourceMenuItem);

// Associate Command with MenuItem.
cmdSourceMenuItem.Command = ApplicationCommands.Properties;

```

```

Dim cmdSourcePanel As New StackPanel()
Dim cmdSourceContextMenu As New ContextMenu()
Dim cmdSourceMenuItem As New MenuItem()

' Add ContextMenu to the StackPanel.
cmdSourcePanel.ContextMenu = cmdSourceContextMenu
cmdSourcePanel.ContextMenu.Items.Add(cmdSourceMenuItem)

' Associate Command with MenuItem.
cmdSourceMenuItem.Command = ApplicationCommands.Properties

```

Typically, a command source will listen to the [CanExecuteChanged](#) event. This event informs the command source that the ability of the command to execute on the current command target may have changed. The command source can query the current status of the [RoutedCommand](#) by using the [CanExecute](#) method. The command source can then disable itself if the command cannot execute. An example of this is a [MenuItem](#) graying itself out when a command cannot execute.

An [InputGesture](#) can be used as a command source. Two types of input gestures in WPF are the [KeyGesture](#) and [MouseGesture](#). You can think of a [KeyGesture](#) as a keyboard shortcut, such as CTRL+C. A [KeyGesture](#) is comprised of a [Key](#) and a set of [ModifierKeys](#). A [MouseGesture](#) is comprised of a [MouseAction](#) and an optional set of [ModifierKeys](#).

In order for an [InputGesture](#) to act as a command source, it must be associated with a command. There are a few ways to accomplish this. One way is to use an [InputBinding](#).

The following example shows how to create a [KeyBinding](#) between a [KeyGesture](#) and a [RoutedCommand](#).

```

<Window.InputBindings>
  <KeyBinding Key="B"
    Modifiers="Control"
    Command="ApplicationCommands.Open" />
</Window.InputBindings>

```

```

KeyGesture OpenKeyGesture = new KeyGesture(
    Key.B,
    ModifierKeys.Control);

KeyBinding OpenCmdKeybinding = new KeyBinding(
    ApplicationCommands.Open,
    OpenKeyGesture);

this.InputBindings.Add(OpenCmdKeybinding);

```

```

Dim OpenKeyGesture As New KeyGesture(Key.B, ModifierKeys.Control)

Dim OpenCmdKeybinding As New KeyBinding(ApplicationCommands.Open, OpenKeyGesture)

Me.InputBindings.Add(OpenCmdKeybinding)

```

Another way to associate an [InputGesture](#) to a [RoutedCommand](#) is to add the [InputGesture](#) to the [InputGestureCollection](#) on the [RoutedCommand](#).

The following example shows how to add a [KeyGesture](#) to the [InputGestureCollection](#) of a [RoutedCommand](#).

```

KeyGesture OpenCmdKeyGesture = new KeyGesture(
    Key.B,
    ModifierKeys.Control);

ApplicationCommands.Open.InputGestures.Add(OpenCmdKeyGesture);

```

```

Dim OpenCmdKeyGesture As New KeyGesture(Key.B, ModifierKeys.Control)

ApplicationCommands.Open.InputGestures.Add(OpenCmdKeyGesture)

```

## CommandBinding

A [CommandBinding](#) associates a command with the event handlers that implement the command.

The [CommandBinding](#) class contains a [Command](#) property, and [PreviewExecuted](#), [Executed](#), [PreviewCanExecute](#), and [CanExecute](#) events.

[Command](#) is the command that the [CommandBinding](#) is being associated with. The event handlers which are attached to the [PreviewExecuted](#) and [Executed](#) events implement the command logic. The event handlers attached to the [PreviewCanExecute](#) and [CanExecute](#) events determine if the command can execute on the current command target.

The following example shows how to create a [CommandBinding](#) on the root [Window](#) of an application. The [CommandBinding](#) associates the [Open](#) command with [Executed](#) and [CanExecute](#) handlers.

```

<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
        Executed="OpenCmdExecuted"
        CanExecute="OpenCmdCanExecute"/>
</Window.CommandBindings>

```

```
// Creating CommandBinding and attaching an Executed and CanExecute handler
CommandBinding OpenCmdBinding = new CommandBinding(
    ApplicationCommands.Open,
    OpenCmdExecuted,
    OpenCmdCanExecute);

this.CommandBindings.Add(OpenCmdBinding);
```

```
' Creating CommandBinding and attaching an Executed and CanExecute handler
Dim OpenCmdBinding As New CommandBinding(ApplicationCommands.Open, AddressOf OpenCmdExecuted, AddressOf
OpenCmdCanExecute)

Me.CommandBindings.Add(OpenCmdBinding)
```

Next, the [ExecutedRoutedEventHandler](#) and a [CanExecuteRoutedEventHandler](#) are created. The [ExecutedRoutedEventHandler](#) opens a [MessageBox](#) that displays a string saying the command has been executed. The [CanExecuteRoutedEventHandler](#) sets the [CanExecute](#) property to `true`.

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    String command, targetobj;
    command = ((RoutedCommand)e.Command).Name;
    targetobj = ((FrameworkElement)target).Name;
    MessageBox.Show("The " + command + " command has been invoked on target object " + targetobj);
}
```

```
Private Sub OpenCmdExecuted(ByVal sender As Object, ByVal e As ExecutedRoutedEventArgs)
    Dim command, targetobj As String
    command = CType(e.Command, RoutedCommand).Name
    targetobj = CType(sender, FrameworkElement).Name
    MessageBox.Show("The " + command + " command has been invoked on target object " + targetobj)
End Sub
```

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

```
Private Sub OpenCmdCanExecute(ByVal sender As Object, ByVal e As CanExecuteRoutedEventArgs)
    e.CanExecute = True
End Sub
```

A [CommandBinding](#) is attached to a specific object, such as the root [Window](#) of the application or a control. The object that the [CommandBinding](#) is attached to defines the scope of the binding. For example, a [CommandBinding](#) attached to an ancestor of the command target can be reached by the [Executed](#) event, but a [CommandBinding](#) attached to a descendant of the command target cannot be reached. This is a direct consequence of the way a [RoutedEventArgs](#) tunnels and bubbles from the object that raises the event.

In some situations the [CommandBinding](#) is attached to the command target itself, such as with the [TextBox](#) class and the [Cut](#), [Copy](#), and [Paste](#) commands. Quite often though, it is more convenient to attach the [CommandBinding](#) to an ancestor of the command target, such as the main [Window](#) or the Application object, especially if the same [CommandBinding](#) can be used for multiple command targets. These are design decisions you will want to consider when you are creating your commanding infrastructure.

## Command Target

The command target is the element on which the command is executed. With regards to a [RoutedCommand](#), the command target is the element at which routing of the [Executed](#) and [CanExecute](#) starts. As noted previously, in WPF the [CommandTarget](#) property on [ICommandSource](#) is only applicable when the  [ICommand](#) is a [RoutedCommand](#). If the [CommandTarget](#) is set on an [ICommandSource](#) and the corresponding command is not a [RoutedCommand](#), the command target is ignored.

The command source can explicitly set the command target. If the command target is not defined, the element with keyboard focus will be used as the command target. One of the benefits of using the element with keyboard focus as the command target is that it allows the application developer to use the same command source to invoke a command on multiple targets without having to keep track of the command target. For example, if a [MenuItem](#) invokes the **Paste** command in an application that has a [TextBox](#) control and a [PasswordBox](#) control, the target can be either the [TextBox](#) or [PasswordBox](#) depending on which control has keyboard focus.

The following example shows how to explicitly set the command target in markup and in code behind.

```
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Paste"
                  CommandTarget="{Binding ElementName=mainTextBox}" />
    </Menu>
    <TextBox Name="mainTextBox"/>
</StackPanel>
```

```
// Creating the UI objects
StackPanel mainStackPanel = new StackPanel();
TextBox pasteTextBox = new TextBox();
Menu stackPanelMenu = new Menu();
MenuItem pasteMenuItem = new MenuItem();

// Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem);
mainStackPanel.Children.Add(stackPanelMenu);
mainStackPanel.Children.Add(pasteTextBox);

// Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste;

// Setting the command target to the TextBox
pasteMenuItem.CommandTarget = pasteTextBox;
```

```
' Creating the UI objects
Dim mainStackPanel As New StackPanel()
Dim pasteTextBox As New TextBox()
Dim stackPanelMenu As New Menu()
Dim pasteMenuItem As New MenuItem()

' Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem)
mainStackPanel.Children.Add(stackPanelMenu)
mainStackPanel.Children.Add(pasteTextBox)

' Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste
```

## The [CommandManager](#)

The [CommandManager](#) serves a number of command related functions. It provides a set of static methods for

adding and removing [PreviewExecuted](#), [Executed](#), [PreviewCanExecute](#), and [CanExecute](#) event handlers to and from a specific element. It provides a means to register [CommandBinding](#) and [InputBinding](#) objects onto a specific class. The [CommandManager](#) also provides a means, through the [RequerySuggested](#) event, to notify a command when it should raise the [CanExecuteChanged](#) event.

The [InvalidateRequerySuggested](#) method forces the [CommandManager](#) to raise the [RequerySuggested](#) event. This is useful for conditions that should disable/enable a command but are not conditions that the [CommandManager](#) is aware of.

## Command Library

WPF provides a set of predefined commands. The command library consists of the following classes: [ApplicationCommands](#), [NavigationCommands](#), [MediaCommands](#), [EditingCommands](#), and the [ComponentCommands](#). These classes provide commands such as [Cut](#), [BrowseBack](#) and [BrowseForward](#), [Play](#), [Stop](#), and [Pause](#).

Many of these commands include a set of default input bindings. For example, if you specify that your application handles the copy command, you automatically get the keyboard binding "CTRL+C". You also get bindings for other input devices, such as Tablet PC pen gestures and speech information.

When you reference commands in the various command libraries using XAML, you can usually omit the class name of the library class that exposes the static command property. Generally, the command names are unambiguous as strings, and the owning types exist to provide a logical grouping of commands but are not necessary for disambiguation. For instance, you can specify `Command="Cut"` rather than the more verbose `Command="ApplicationCommands.Cut"`. This is a convenience mechanism that is built in to the WPF XAML processor for commands (more precisely, it is a type converter behavior of [ICommand](#), which the WPF XAML processor references at load time).

## Creating Custom Commands

If the commands in the command library classes do not meet your needs, then you can create your own commands. There are two ways to create a custom command. The first is to start from the ground up and implement the [ICommand](#) interface. The other way, and the more common approach, is to create a [RoutedCommand](#) or a [RoutedUICommand](#).

For an example of creating a custom [RoutedCommand](#), see [Create a Custom RoutedCommand Sample](#).

## See also

- [RoutedCommand](#)
- [CommandBinding](#)
- [InputBinding](#)
- [CommandManager](#)
- [Input Overview](#)
- [Routed Events Overview](#)
- [Implement ICommandSource](#)
- [How to: Add a Command to a MenuItem](#)
- [Create a Custom RoutedCommand Sample](#)

# Focus Overview

7/16/2019 • 7 minutes to read • [Edit Online](#)

In WPF there are two main concepts that pertain to focus: keyboard focus and logical focus. Keyboard focus refers to the element that receives keyboard input and logical focus refers to the element in a focus scope that has focus. These concepts are discussed in detail in this overview. Understanding the difference in these concepts is important for creating complex applications that have multiple regions where focus can be obtained.

The major classes that participate in focus management are the [Keyboard](#) class, the [FocusManager](#) class, and the base element classes, such as [UIElement](#) and [ContentElement](#). For more information about the base elements, see the [Base Elements Overview](#).

The [Keyboard](#) class is concerned primarily with keyboard focus and the [FocusManager](#) is concerned primarily with logical focus, but this is not an absolute distinction. An element that has keyboard focus will also have logical focus, but an element that has logical focus does not necessarily have keyboard focus. This is apparent when you use the [Keyboard](#) class to set the element that has keyboard focus, for it also sets logical focus on the element.

## Keyboard Focus

Keyboard focus refers to the element that is currently receiving keyboard input. There can be only one element on the whole desktop that has keyboard focus. In WPF, the element that has keyboard focus will have [IsKeyboardFocused](#) set to `true`. The static property [FocusedElement](#) on the [Keyboard](#) class gets the element that currently has keyboard focus.

In order for an element to obtain keyboard focus, the [Focusable](#) and the [IsVisible](#) properties on the base elements must be set to `true`. Some classes, such as the [Panel](#) base class, have [Focusable](#) set to `false` by default; therefore, you must set [Focusable](#) to `true` if you want such an element to be able to obtain keyboard focus.

Keyboard focus can be obtained through user interaction with the UI, such as tabbing to an element or clicking the mouse on certain elements. Keyboard focus can also be obtained programmatically by using the [Focus](#) method on the [Keyboard](#) class. The [Focus](#) method attempts to give the specified element keyboard focus. The returned element is the element that has keyboard focus, which might be a different element than requested if either the old or new focus object block the request.

The following example uses the [Focus](#) method to set keyboard focus on a [Button](#).

```
private void OnLoaded(object sender, RoutedEventArgs e)
{
    // Sets keyboard focus on the first Button in the sample.
    Keyboard.Focus(firstButton);
}
```

```
Private Sub OnLoaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Sets keyboard focus on the first Button in the sample.
    Keyboard.Focus(firstButton)
End Sub
```

The [IsKeyboardFocused](#) property on the base element classes gets a value indicating whether the element has keyboard focus. The [IsKeyboardFocusWithin](#) property on the base element classes gets a value indicating whether the element or any one of its visual child elements has keyboard focus.

When setting initial focus at application startup, the element to receive focus must be in the visual tree of the

initial window loaded by the application, and the element must have [Focusable](#) and [IsVisible](#) set to `true`. The recommended place to set initial focus is in the [Loaded](#) event handler. A [Dispatcher](#) callback can also be used by calling [Invoke](#) or [BeginInvoke](#).

## Logical Focus

Logical focus refers to the [FocusManager.FocusedElement](#) in a focus scope. A focus scope is an element that keeps track of the [FocusedElement](#) within its scope. When keyboard focus leaves a focus scope, the focused element will lose keyboard focus but will retain logical focus. When keyboard focus returns to the focus scope, the focused element will obtain keyboard focus. This allows for keyboard focus to be changed between multiple focus scopes but ensures that the focused element in the focus scope regains keyboard focus when focus returns to the focus scope.

There can be multiple elements that have logical focus in an application, but there may only be one element that has logical focus in a particular focus scope.

An element that has keyboard focus has logical focus for the focus scope it belongs to.

An element can be turned into a focus scope in Extensible Application Markup Language (XAML) by setting the [FocusManager](#) attached property [IsFocusScope](#) to `true`. In code, an element can be turned into a focus scope by calling [SetIsFocusScope](#).

The following example makes a [StackPanel](#) into a focus scope by setting the [IsFocusScope](#) attached property.

```
<StackPanel Name="focusScope1"
            FocusManager.IsFocusScope="True"
            Height="200" Width="200">
    <Button Name="button1" Height="50" Width="50"/>
    <Button Name="button2" Height="50" Width="50"/>
</StackPanel>
```

```
StackPanel focusScope2 = new StackPanel();
FocusManager.SetIsFocusScope(focusScope2, true);
```

```
Dim focusScope2 As New StackPanel()
FocusManager.SetIsFocusScope(focusScope2, True)
```

[GetFocusScope](#) returns the focus scope for the specified element.

Classes in WPF which are focus scopes by default are [Window](#), [MenuItem](#), [ToolBar](#), and [ContextMenu](#).

[GetFocusedElement](#) gets the focused element for the specified focus scope. [SetFocusedElement](#) sets the focused element in the specified focus scope. [SetFocusedElement](#) is typically used to set the initial focused element.

The following example sets the focused element on a focus scope and gets the focused element of a focus scope.

```
// Sets the focused element in focusScope1
// focusScope1 is a StackPanel.
FocusManager.SetFocusedElement(focusScope1, button2);

// Gets the focused element for focusScope 1
IInputElement focusedElement = FocusManager.GetFocusedElement(focusScope1);
```

```

' Sets the focused element in focusScope1
' focusScope1 is a StackPanel.
FocusManager.SetFocusedElement(focusScope1, button2)

' Gets the focused element for focusScope 1
Dim focusedElement As IInputElement = FocusManager.GetFocusedElement(focusScope1)

```

## Keyboard Navigation

The [KeyboardNavigation](#) class is responsible for implementing default keyboard focus navigation when one of the navigation keys is pressed. The navigation keys are: TAB, SHIFT+TAB, CTRL+TAB, CTRL+SHIFT+TAB, UPARROW, DOWNARROW, LEFTARROW, and RIGHTARROW keys.

The navigation behavior of a navigation container can be changed by setting the attached [KeyboardNavigation](#) properties [TabNavigation](#), [ControlTabNavigation](#), and [DirectionalNavigation](#). These properties are of type [KeyboardNavigationMode](#) and the possible values are [Continue](#), [Local](#), [Contained](#), [Cycle](#), [Once](#), and [None](#). The default value is [Continue](#), which means the element is not a navigation container.

The following example creates a [Menu](#) with a number of [MenuItem](#) objects. The [TabNavigation](#) attached property is set to [Cycle](#) on the [Menu](#). When focus is changed using the tab key within the [Menu](#), focus will move from each element and when the last element is reached focus will return to the first element.

```

<Menu KeyboardNavigation.TabNavigation="Cycle">
    <MenuItem Header="Menu Item 1" />
    <MenuItem Header="Menu Item 2" />
    <MenuItem Header="Menu Item 3" />
    <MenuItem Header="Menu Item 4" />
</Menu>

```

```

Menu navigationMenu = new Menu();
MenuItem item1 = new MenuItem();
MenuItem item2 = new MenuItem();
MenuItem item3 = new MenuItem();
MenuItem item4 = new MenuItem();

navigationMenu.Items.Add(item1);
navigationMenu.Items.Add(item2);
navigationMenu.Items.Add(item3);
navigationMenu.Items.Add(item4);

KeyboardNavigation.SetTabNavigation(navigationMenu,
    KeyboardNavigationMode.Cycle);

```

```

Dim navigationMenu As New Menu()
Dim item1 As New MenuItem()
Dim item2 As New MenuItem()
Dim item3 As New MenuItem()
Dim item4 As New MenuItem()

navigationMenu.Items.Add(item1)
navigationMenu.Items.Add(item2)
navigationMenu.Items.Add(item3)
navigationMenu.Items.Add(item4)

KeyboardNavigation.SetTabNavigation(navigationMenu, KeyboardNavigationMode.Cycle)

```

# Navigating Focus Programmatically

Additional API to work with focus are [MoveFocus](#) and [PredictFocus](#).

[MoveFocus](#) changes focus to the next element in the application. A [TraversalRequest](#) is used to specify the direction. The [FocusNavigationDirection](#) passed to [MoveFocus](#) specifies the different directions focus can be moved, such as [First](#), [Last](#), [Up](#) and [Down](#).

The following example uses [MoveFocus](#) to change the focused element.

```
// Creating a FocusNavigationDirection object and setting it to a
// local field that contains the direction selected.
FocusNavigationDirection focusDirection = _focusMoveValue;

// MoveFocus takes a TraversalRequest as its argument.
TraversalRequest request = new TraversalRequest(focusDirection);

// Gets the element with keyboard focus.
UIElement elementWithFocus = Keyboard.FocusedElement as UIElement;

// Change keyboard focus.
if (elementWithFocus != null)
{
    elementWithFocus.MoveFocus(request);
}
```

```
' Creating a FocusNavigationDirection object and setting it to a
' local field that contains the direction selected.
Dim focusDirection As FocusNavigationDirection = _focusMoveValue

' MoveFocus takes a TraversalRequest as its argument.
Dim request As New TraversalRequest(focusDirection)

' Gets the element with keyboard focus.
Dim elementWithFocus As UIElement = TryCast(Keyboard.FocusedElement, UIElement)

' Change keyboard focus.
If elementWithFocus IsNot Nothing Then
    elementWithFocus.MoveFocus(request)
End If
```

[PredictFocus](#) returns the object which would receive focus if focus were to be changed. Currently, only [Up](#), [Down](#), [Left](#), and [Right](#) are supported by [PredictFocus](#).

## Focus Events

The events related to keyboard focus are [PreviewGotKeyboardFocus](#), [GotKeyboardFocus](#) and [PreviewLostKeyboardFocus](#), [LostKeyboardFocus](#). The events are defined as attached events on the [Keyboard](#) class, but are more readily accessible as equivalent routed events on the base element classes. For more information about events, see the [Routed Events Overview](#).

[GotKeyboardFocus](#) is raised when the element obtains keyboard focus. [LostKeyboardFocus](#) is raised when the element loses keyboard focus. If the [PreviewGotKeyboardFocus](#) event or the [PreviewLostKeyboardFocusEvent](#) event is handled and [Handled](#) is set to `true`, then focus will not change.

The following example attaches [GotKeyboardFocus](#) and [LostKeyboardFocus](#) event handlers to a [TextBox](#).

```

<Border BorderBrush="Black" BorderThickness="1"
        Width="200" Height="100" Margin="5">
    <StackPanel>
        <Label HorizontalAlignment="Center" Content="Type Text In This TextBox" />
        <TextBox Width="175"
                Height="50"
                Margin="5"
                TextWrapping="Wrap"
                HorizontalAlignment="Center"
                VerticalScrollBarVisibility="Auto"
                GotKeyboardFocus="TextBoxGotKeyboardFocus"
                LostKeyboardFocus="TextBoxLostKeyboardFocus"
                KeyDown="SourceTextKeyDown"/>
    </StackPanel>
</Border>

```

When the **TextBox** obtains keyboard focus, the **Background** property of the **TextBox** is changed to **LightBlue**.

```

private void TextBoxGotKeyboardFocus(object sender, KeyboardFocusChangedEventArgs e)
{
    TextBox source = e.Source as TextBox;

    if (source != null)
    {
        // Change the TextBox color when it obtains focus.
        source.Background = Brushes.LightBlue;

        // Clear the TextBox.
        source.Clear();
    }
}

```

```

Private Sub TextBoxGotKeyboardFocus(ByVal sender As Object, ByVal e As KeyboardFocusChangedEventArgs)
    Dim source As TextBox = TryCast(e.Source, TextBox)

    If source IsNot Nothing Then
        ' Change the TextBox color when it obtains focus.
        source.Background = Brushes.LightBlue

        ' Clear the TextBox.
        source.Clear()
    End If
End Sub

```

When the **TextBox** loses keyboard focus, the **Background** property of the **TextBox** is changed back to white.

```

private void TextBoxLostKeyboardFocus(object sender, KeyboardFocusChangedEventArgs e)
{
    TextBox source = e.Source as TextBox;

    if (source != null)
    {
        // Change the TextBox color when it loses focus.
        source.Background = Brushes.White;

        // Set the hit counter back to zero and updates the display.
        this.ResetCounter();
    }
}

```

```
Private Sub TextBoxLostKeyboardFocus(ByVal sender As Object, ByVal e As KeyboardFocusChangedEventArgs)
    Dim source As TextBox = TryCast(e.Source, TextBox)

    If source IsNot Nothing Then
        ' Change the TextBox color when it loses focus.
        source.Background = Brushes.White

        ' Set the hit counter back to zero and updates the display.
        Me.ResetCounter()
    End If
End Sub
```

The events related to logical focus are [GotFocus](#) and [LostFocus](#). These events are defined on the [FocusManager](#) as attached events, but the [FocusManager](#) does not expose CLR event wrappers. [UIElement](#) and [ContentElement](#) expose these events more conveniently.

## See also

- [FocusManager](#)
- [UIElement](#)
- [ContentElement](#)
- [Input Overview](#)
- [Base Elements Overview](#)

# Styling for Focus in Controls, and FocusVisualStyle

11/3/2019 • 5 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides two parallel mechanisms for changing the visual appearance of a control when it receives keyboard focus. The first mechanism is to use property setters for properties such as `IsKeyboardFocused` within the style or template that is applied to the control. The second mechanism is to provide a separate style as the value of the `FocusVisualStyle` property; the "focus visual style" creates a separate visual tree for an adorner that draws on top of the control, rather than changing the visual tree of the control or other UI element by replacing it. This topic discusses the scenarios where each of these mechanisms is appropriate.

## The Purpose of Focus Visual Style

The focus visual style feature provides a common "object model" for introducing visual user feedback based on keyboard navigation to any UI element. This is possible without applying a new template to the control, or knowing the specific template composition.

However, precisely because the focus visual style feature works without knowing the control templates, the visual feedback that can be displayed for a control using a focus visual style is necessarily limited. What the feature actually does is to overlay a different visual tree (an adorner) on top of the visual tree as created by a control's rendering through its template. You define this separate visual tree using a style that fills the `FocusVisualStyle` property.

## Default Focus Visual Style Behavior

Focus visual styles act only when the focus action was initiated by the keyboard. Any mouse action or programmatic focus change disables the mode for focus visual styles. For more information about the distinctions between focus modes, see [Focus Overview](#).

The themes for controls include a default focus visual style behavior that becomes the focus visual style for all controls in the theme. This theme style is identified by the value of the static key `FocusVisualStyleKey`. When you declare your own focus visual style at the application level, you replace this default style behavior from the themes. Alternatively, if you define the entire theme, then you should use this same key to define the style for the default behavior for your entire theme.

In the themes, the default focus visual style is generally very simple. The following is a rough approximation:

```
<Style x:Key="{x:Static SystemParameters.FocusVisualStyleKey}">
  <Setter Property="Control.Template">
    <Setter.Value>
      <ControlTemplate>
        <Rectangle StrokeThickness="1"
                  Stroke="Black"
                  StrokeDashArray="1 2"
                  SnapsToDevicePixels="true"/>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

## When to Use Focus Visual Styles

Conceptually, the appearance of focus visual styles applied to controls should be coherent from control to control.

One way to ensure coherence is to change the focus visual style only if you are composing an entire theme, where each control that is defined in the theme gets either the very same focus visual style, or some variation of a style that is visually related from control to control. Alternatively, you might use the same style (or similar styles) to style every keyboard-focusable element on a page or in a UI.

Setting [FocusVisualStyle](#) on individual control styles that are not part of a theme is not the intended usage of focus visual styles. This is because an inconsistent visual behavior between controls can lead to a confusing user experience regarding keyboard focus. If you are intending control-specific behaviors for keyboard focus that are deliberately not coherent across a theme, a much better approach is to use triggers in styles for individual input state properties, such as [IsFocused](#) or [IsKeyboardFocused](#).

Focus visual styles act exclusively for keyboard focus. As such, focus visual styles are a type of accessibility feature. If you want UI changes for any type of focus, whether via mouse, keyboard, or programmatically, then you should not use focus visual styles, and should instead use setters and triggers in styles or templates that are working from the value of general focus properties such as [IsFocused](#) or [IsKeyboardFocusWithin](#).

## How to Create a Focus Visual Style

The style you create for a focus visual style should always have the [TargetType](#) of [Control](#). The style should consist mainly of a [ControlTemplate](#). You do not specify the target type to be the type where the focus visual style is assigned to the [FocusVisualStyle](#).

Because the target type is always [Control](#), you must style by using properties that are common to all controls (using properties of the [Control](#) class and its base classes). You should create a template that will properly function as an overlay to a UI element and that will not obscure functional areas of the control. Generally, this means that the visual feedback should appear outside the control margins, or as temporary or unobtrusive effects that will not block the hit testing on the control where the focus visual style is applied. Properties that you can use in template binding that are useful for determining sizing and positioning of your overlay template include [ActualHeight](#), [ActualWidth](#), [Margin](#), and [Padding](#).

## Alternatives to Using a Focus Visual Style

For situations where using a focus visual style is not appropriate, either because you are only styling single controls or because you want greater control over the control template, there are many other accessible properties and techniques that can create visual behavior in response to changes in focus.

Triggers, setters, and event setters are all discussed in detail in [Styling and Templating](#). Routed event handling is discussed in [Routed Events Overview](#).

### **IsKeyboardFocused**

If you are specifically interested in keyboard focus, the [IsKeyboardFocused](#) dependency property can be used for a property [Trigger](#). A property trigger in either a style or template is a more appropriate technique for defining a keyboard focus behavior that is very specifically for a single control, and which might not visually match the keyboard focus behavior for other controls.

Another similar dependency property is [IsKeyboardFocusWithin](#), which might be appropriate to use if you want to visually call out that keyboard focus is somewhere within compositing or within the functional area of the control. For example, you might place an [IsKeyboardFocusWithin](#) trigger such that a panel that groups several controls appears differently, even though keyboard focus might more precisely be on an individual element within that panel.

You can also use the events [GotKeyboardFocus](#) and [LostKeyboardFocus](#) (as well as their Preview equivalents). You can use these events as the basis for an [EventSetter](#), or you can write handlers for the events in code-behind.

### **Other Focus Properties**

If you want all possible causes of changing focus to produce a visual behavior, you should base a setter or trigger

on the [IsFocused](#) dependency property, or alternatively on the [GotFocus](#) or [LostFocus](#) events used for an [EventSetter](#).

## See also

- [FocusVisualStyle](#)
- [Styling and Templating](#)
- [Focus Overview](#)
- [Input Overview](#)

# Walkthrough: Creating Your First Touch Application

5/15/2019 • 5 minutes to read • [Edit Online](#)

WPF enables applications to respond to touch. For example, you can interact with an application by using one or more fingers on a touch-sensitive device, such as a touchscreen. This walkthrough creates an application that enables the user to move, resize, or rotate a single object by using touch.

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio.
- A device that accepts touch input, such as a touchscreen, that supports Windows Touch.

Additionally, you should have a basic understanding of how to create an application in WPF, especially how to subscribe to and handle an event. For more information, see [Walkthrough: My first WPF desktop application](#).

## Creating the Application

### To create the application

1. Create a new WPF Application project in Visual Basic or Visual C# named `BasicManipulation`. For more information, see [Walkthrough: My first WPF desktop application](#).
2. Replace the contents of `MainWindow.xaml` with the following XAML.

This markup creates a simple application that contains a red `Rectangle` on a `Canvas`. The `IsManipulationEnabled` property of the `Rectangle` is set to true so that it will receive manipulation events. The application subscribes to the `ManipulationStarting`, `ManipulationDelta`, and `ManipulationInertiaStarting` events. These events contain the logic to move the `Rectangle` when the user manipulates it.

```

<Window x:Class="BasicManipulation.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Move, Size, and Rotate the Square"
    WindowState="Maximized"
    ManipulationStarting="Window_ManipulationStarting"
    ManipulationDelta="Window_ManipulationDelta"
    ManipulationInertiaStarting="Window_InertiaStarting">
<Window.Resources>

    <!--The movement, rotation, and size of the Rectangle is
        specified by its RenderTransform.-->
    <MatrixTransform x:Key="InitialMatrixTransform">
        <MatrixTransform.Matrix>
            <Matrix OffsetX="200" OffsetY="200"/>
        </MatrixTransform.Matrix>
    </MatrixTransform>

</Window.Resources>

<Canvas>
    <Rectangle Fill="Red" Name="manRect"
        Width="200" Height="200"
        RenderTransform="{StaticResource InitialMatrixTransform}"
        IsManipulationEnabled="true" />
</Canvas>
</Window>

```

3. If you are using Visual Basic, in the first line of MainWindow.xaml, replace

`x:Class="BasicManipulation.MainWindow"` with `x:Class="MainWindow"`.

4. In the `MainWindow` class, add the following `ManipulationStarting` event handler.

The `ManipulationStarting` event occurs when WPF detects that touch input begins to manipulate an object. The code specifies that the position of the manipulation should be relative to the `Window` by setting the `ManipulationContainer` property.

```

void Window_ManipulationStarting(object sender, ManipulationStartingEventArgs e)
{
    e.ManipulationContainer = this;
    e.Handled = true;
}

```

```

Private Sub Window_ManipulationStarting(ByVal sender As Object, ByVal e As
ManipulationStartingEventArgs)
    e.ManipulationContainer = Me
    e.Handled = True
End Sub

```

5. In the `MainWindow` class, add the following `ManipulationDelta` event handler.

The `ManipulationDelta` event occurs when the touch input changes position and can occur multiple times during a manipulation. The event can also occur after a finger is raised. For example, if the user drags a finger across a screen, the `ManipulationDelta` event occurs multiple times as the finger moves. When the user raises a finger from the screen, the `ManipulationDelta` event keeps occurring to simulate inertia.

The code applies the `DeltaManipulation` to the `RenderTransform` of the `Rectangle` to move it as the user moves the touch input. It also checks whether the `Rectangle` is outside the bounds of the `Window` when the event occurs during inertia. If so, the application calls the `ManipulationDeltaEventArgs.Complete` method to

end the manipulation.

```
void Window_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    // Get the Rectangle and its RenderTransform matrix.
    Rectangle rectToMove = e.OriginalSource as Rectangle;
    Matrix rectsMatrix = ((MatrixTransform)rectToMove.RenderTransform).Matrix;

    // Rotate the Rectangle.
    rectsMatrix.RotateAt(e.DeltaManipulation.Rotation,
        e.ManipulationOrigin.X,
        e.ManipulationOrigin.Y);

    // Resize the Rectangle. Keep it square
    // so use only the X value of Scale.
    rectsMatrix.ScaleAt(e.DeltaManipulation.Scale.X,
        e.DeltaManipulation.Scale.X,
        e.ManipulationOrigin.X,
        e.ManipulationOrigin.Y);

    // Move the Rectangle.
    rectsMatrix.Translate(e.DeltaManipulation.Translation.X,
        e.DeltaManipulation.Translation.Y);

    // Apply the changes to the Rectangle.
    rectToMove.RenderTransform = new MatrixTransform(rectsMatrix);

    Rect containingRect =
        new Rect(((FrameworkElement)e.ManipulationContainer).RenderSize);

    Rect shapeBounds =
        rectToMove.RenderTransform.TransformBounds(
        new Rect(rectToMove.RenderSize));

    // Check if the rectangle is completely in the window.
    // If it is not and inertia is occuring, stop the manipulation.
    if (e.IsInertial && !containingRect.Contains(shapeBounds))
    {
        e.Complete();
    }

    e.Handled = true;
}
```

```

Private Sub Window_ManipulationDelta(ByVal sender As Object, ByVal e As ManipulationDeltaEventArgs)

    ' Get the Rectangle and its RenderTransform matrix.
    Dim rectToMove As Rectangle = e.OriginalSource
    Dim rectTransform As MatrixTransform = rectToMove.RenderTransform
    Dim rectMatrix As Matrix = rectTransform.Matrix

    ' Rotate the shape
    rectMatrix.RotateAt(e.DeltaManipulation.Rotation,
        e.ManipulationOrigin.X,
        e.ManipulationOrigin.Y)

    ' Resize the Rectangle. Keep it square
    ' so use only the X value of Scale.
    rectMatrix.ScaleAt(e.DeltaManipulation.Scale.X,
        e.DeltaManipulation.Scale.X,
        e.ManipulationOrigin.X,
        e.ManipulationOrigin.Y)

    'move the center
    rectMatrix.Translate(e.DeltaManipulation.Translation.X,
        e.DeltaManipulation.Translation.Y)

    ' Apply the changes to the Rectangle.
    rectTransform = New MatrixTransform(rectMatrix)
    rectToMove.RenderTransform = rectTransform

    Dim container As FrameworkElement = e.ManipulationContainer
    Dim containingRect As New Rect(container.RenderSize)

    Dim shapeBounds As Rect = rectTransform.TransformBounds(
        New Rect(rectToMove.RenderSize))

    ' Check if the rectangle is completely in the window.
    ' If it is not and inertia is occurring, stop the manipulation.
    If e.IsInertial AndAlso Not containingRect.Contains(shapeBounds) Then
        e.Complete()
    End If

    e.Handled = True
End Sub

```

6. In the `MainWindow` class, add the following `ManipulationInertiaStarting` event handler.

The `ManipulationInertiaStarting` event occurs when the user raises all fingers from the screen. The code sets the initial velocity and deceleration for the movement, expansion, and rotation of the rectangle.

```

void Window_InertiaStarting(object sender, ManipulationInertiaStartingEventArgs e)
{

    // Decrease the velocity of the Rectangle's movement by
    // 10 inches per second every second.
    // (10 inches * 96 pixels per inch / 1000ms^2)
    e.TranslationBehavior.DesiredDeceleration = 10.0 * 96.0 / (1000.0 * 1000.0);

    // Decrease the velocity of the Rectangle's resizing by
    // 0.1 inches per second every second.
    // (0.1 inches * 96 pixels per inch / (1000ms^2))
    e.ExpansionBehavior.DesiredDeceleration = 0.1 * 96 / (1000.0 * 1000.0);

    // Decrease the velocity of the Rectangle's rotation rate by
    // 2 rotations per second every second.
    // (2 * 360 degrees / (1000ms^2))
    e.RotationBehavior.DesiredDeceleration = 720 / (1000.0 * 1000.0);

    e.Handled = true;
}

```

```

Private Sub Window_InertiaStarting(ByVal sender As Object,
                                  ByVal e As ManipulationInertiaStartingEventArgs)

    ' Decrease the velocity of the Rectangle's movement by
    ' 10 inches per second every second.
    ' (10 inches * 96 pixels per inch / 1000ms^2)
    e.TranslationBehavior.DesiredDeceleration = 10.0 * 96.0 / (1000.0 * 1000.0);

    ' Decrease the velocity of the Rectangle's resizing by
    ' 0.1 inches per second every second.
    ' (0.1 inches * 96 pixels per inch / (1000ms^2))
    e.ExpansionBehavior.DesiredDeceleration = 0.1 * 96 / (1000.0 * 1000.0);

    ' Decrease the velocity of the Rectangle's rotation rate by
    ' 2 rotations per second every second.
    ' (2 * 360 degrees / (1000ms^2))
    e.RotationBehavior.DesiredDeceleration = 720 / (1000.0 * 1000.0);

    e.Handled = True
End Sub

```

## 7. Build and run the project.

You should see a red square appear in the window.

## Testing the Application

To test the application, try the following manipulations. Note that you can do more than one of the following at the same time.

- To move the [Rectangle](#), put a finger on the [Rectangle](#) and move the finger across the screen.
- To resize the [Rectangle](#), put two fingers on the [Rectangle](#) and move the fingers closer together or farther apart from each other.
- To rotate the [Rectangle](#), put two fingers on the [Rectangle](#) and rotate the fingers around each other.

To cause inertia, quickly raise your fingers from the screen as you perform the previous manipulations. The [Rectangle](#) will continue to move, resize, or rotate for a few seconds before it stops.

## See also

- [UIElement.ManipulationStarting](#)
- [UIElement.ManipulationDelta](#)
- [UIElement.ManipulationInertiaStarting](#)

# Input and Commands How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to use the input and commanding infrastructure in Windows Presentation Foundation (WPF).

## In This Section

- [Enable a Command](#)
- [Change the Cursor Type](#)
- [Change the Color of an Element Using Focus Events](#)
- [Apply a FocusVisualStyle to a Control](#)
- [Detect When the Enter Key is Pressed](#)
- [Create a Rollover Effect Using Events](#)
- [Make an Object Follow the Mouse Pointer](#)
- [Create a RoutedCommand](#)
- [Implement ICommandSource](#)
- [Hook Up a Command to a Control with No Command Support](#)
- [Hook Up a Command to a Control with Command Support](#)

## Reference

- [UIElement](#)
- [FrameworkElement](#)
- [ContentElement](#)
- [FrameworkContentElement](#)
- [Keyboard](#)
- [Mouse](#)
- [FocusManager](#)

## Related Sections

# How to: Enable a Command

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following example demonstrates how to use commanding in Windows Presentation Foundation (WPF). The example shows how to associate a [RoutedCommand](#) to a [Button](#), create a [CommandBinding](#), and create the event handlers which implement the [RoutedCommand](#). For more information on commanding, see the [Commanding Overview](#).

## Example

The first section of code creates the user interface (UI), which consists of a [Button](#) and a [StackPanel](#), and creates a [CommandBinding](#) that associates the command handlers with the [RoutedCommand](#).

The [Command](#) property of the [Button](#) is associated with the [Close](#) command.

The [CommandBinding](#) is added to the [CommandBindingCollection](#) of the root [Window](#). The [Executed](#) and [CanExecute](#) event handlers are attached to this binding and associated with the [Close](#) command.

Without the [CommandBinding](#) there is no command logic, only a mechanism to invoke the command. When the [Button](#) is clicked, the [PreviewExecuted RoutedEvent](#) is raised on the command target followed by the [Executed RoutedEvent](#). These events traverse the element tree looking for a [CommandBinding](#) for that particular command. It is worth noting that because [RoutedEventArgs](#) tunnel and bubble through the element tree, care must be taken in where the [CommandBinding](#) is put. If the [CommandBinding](#) is on a sibling of the command target or another node that is not on the route of the [RoutedEventArgs](#), the [CommandBinding](#) will not be accessed.

```
<Window x:Class="WCSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CloseCommand"
    Name="RootWindow"
    >
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Close"
        Executed="CloseCommandHandler"
        CanExecute="CanExecuteHandler"
        />
</Window.CommandBindings>
<StackPanel Name="MainStackPanel">
    <Button Command="ApplicationCommands.Close"
        Content="Close File" />
</StackPanel>
</Window>
```

```

// Create ui elements.
StackPanel CloseCmdStackPanel = new StackPanel();
Button CloseCmdButton = new Button();
CloseCmdStackPanel.Children.Add(CloseCmdButton);

// Set Button's properties.
CloseCmdButton.Content = "Close File";
CloseCmdButton.Command = ApplicationCommands.Close;

// Create the CommandBinding.
CommandBinding CloseCommandBinding = new CommandBinding(
    ApplicationCommands.Close, CloseCommandHandler, CanExecuteHandler);

// Add the CommandBinding to the root Window.
RootWindow.CommandBindings.Add(CloseCommandBinding);

```

```

' Create ui elements.
Dim CloseCmdStackPanel As New StackPanel()
Dim CloseCmdButton As New Button()
CloseCmdStackPanel.Children.Add(CloseCmdButton)

' Set Button's properties.
CloseCmdButton.Content = "Close File"
CloseCmdButton.Command = ApplicationCommands.Close

' Create the CommandBinding.
Dim CloseCommandBinding As New CommandBinding(ApplicationCommands.Close, AddressOf CloseCommandHandler,
AddressOf CanExecuteHandler)

' Add the CommandBinding to the root Window.
RootWindow.CommandBindings.Add(CloseCommandBinding)

```

The next section of code implements the [Executed](#) and [CanExecute](#) event handlers.

The [Executed](#) handler calls a method to close the open file. The [CanExecute](#) handler calls a method to determine whether a file is open. If a file is open, [CanExecute](#) is set to `true`; otherwise, it is set to `false`.

```

// Executed event handler.
private void CloseCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
    // Calls a method to close the file and release resources.
    CloseFile();
}

// CanExecute event handler.
private void CanExecuteHandler(object sender, CanExecuteRoutedEventArgs e)
{
    // Call a method to determine if there is a file open.
    // If there is a file open, then set CanExecute to true.
    if (IsFileOpened())
    {
        e.CanExecute = true;
    }
    // if there is not a file open, then set CanExecute to false.
    else
    {
        e.CanExecute = false;
    }
}

```

```
' Executed event handler.  
Private Sub CloseCommandHandler(ByVal sender As Object, ByVal e As ExecutedRoutedEventArgs)  
    ' Calls a method to close the file and release resources.  
    CloseFile()  
End Sub  
  
' CanExecute event handler.  
Private Sub CanExecuteHandler(ByVal sender As Object, ByVal e As CanExecuteRoutedEventArgs)  
    ' Call a method to determine if there is a file open.  
    ' If there is a file open, then set CanExecute to true.  
    If IsFileOpened() Then  
        e.CanExecute = True  
    ' if there is not a file open, then set CanExecute to false.  
    Else  
        e.CanExecute = False  
    End If  
End Sub
```

## See also

- [Commanding Overview](#)

# How to: Change the Cursor Type

4/8/2019 • 3 minutes to read • [Edit Online](#)

This example shows how to change the [Cursor](#) of the mouse pointer for a specific element and for the application.

This example consists of a Extensible Application Markup Language (XAML) file and a code behind file.

## Example

The user interface is created, which consists of a [ComboBox](#) to select the desired [Cursor](#), a pair of [RadioButton](#) objects to determine if the cursor change applies to only a single element or applies to the entire application, and a [Border](#) which is the element that the new cursor is applied to.

```

<StackPanel>
    <Border Width="300">
        <StackPanel Orientation="Horizontal"
            HorizontalAlignment="Center">
            <StackPanel Margin="10">
                <Label HorizontalAlignment="Left">Cursor Type</Label>
                <ComboBox Width="100"
                    SelectionChanged="CursorTypeChanged"
                    HorizontalAlignment="Left"
                    Name="CursorSelector">
                    <ComboBoxItem Content="AppStarting" />
                    <ComboBoxItem Content="ArrowCD" />
                    <ComboBoxItem Content="Arrow" />
                    <ComboBoxItem Content="Cross" />
                    <ComboBoxItem Content="HandCursor" />
                    <ComboBoxItem Content="Help" />
                    <ComboBoxItem Content="IBeam" />
                    <ComboBoxItem Content="No" />
                    <ComboBoxItem Content="None" />
                    <ComboBoxItem Content="Pen" />
                    <ComboBoxItem Content="ScrollSE" />
                    <ComboBoxItem Content="ScrollWE" />
                    <ComboBoxItem Content="SizeAll" />
                    <ComboBoxItem Content="SizeNESW" />
                    <ComboBoxItem Content="SizeNS" />
                    <ComboBoxItem Content="SizeNWSE" />
                    <ComboBoxItem Content="SizeWE" />
                    <ComboBoxItem Content="UpArrow" />
                    <ComboBoxItem Content="WaitCursor" />
                    <ComboBoxItem Content="Custom" />
                </ComboBox>
            </StackPanel>
            <!-- The user can select different cursor types using this ComboBox -->
            <StackPanel Margin="10">
                <Label HorizontalAlignment="Left">Scope of Cursor</Label>
                <StackPanel>
                    <RadioButton Name="rbScopeElement" IsChecked="True"
                        Checked="CursorScopeSelected">Display Area Only</RadioButton>
                    <RadioButton Name="rbScopeApplication"
                        Checked="CursorScopeSelected">Entire Application</RadioButton>
                </StackPanel>
            </StackPanel>
        </StackPanel>
    </Border>
    <!-- When the mouse pointer is over this Border -->
    <!-- the selected cursor type is shown -->
    <Border Name="DisplayArea" Height="250" Width="400"
        Margin="20" Background="AliceBlue">
        <Label HorizontalAlignment="Center">
            Move Mouse Pointer Over This Area
        </Label>
    </Border>
</StackPanel>

```

The following code behind creates a [SelectionChanged](#) event handler which is called when the cursor type is changed in the [ComboBox](#). A switch statement filters on the cursor name and sets the [Cursor](#) property on the [Border](#) which is named *DisplayArea*.

If the cursor change is set to "Entire Application", the [OverrideCursor](#) property is set to the [Cursor](#) property of the [Border](#) control. This forces the cursor to change for the whole application.

```

private void CursorTypeChanged(object sender, SelectionChangedEventArgs e)
{
    ComboBox source = e.Source as ComboBox;

```

```
if (source != null)
{
    ComboBoxItem selectedCursor = source.SelectedItem as ComboBoxItem;

    // Changing the cursor of the Border control
    // by setting the Cursor property
    switch (selectedCursor.Content.ToString())
    {
        case "AppStarting":
            DisplayArea.Cursor = Cursors.AppStarting;
            break;
        case "ArrowCD":
            DisplayArea.Cursor = Cursors.ArrowCD;
            break;
        case "Arrow":
            DisplayArea.Cursor = Cursors.Arrow;
            break;
        case "Cross":
            DisplayArea.Cursor = Cursors.Cross;
            break;
        case "HandCursor":
            DisplayArea.Cursor = Cursors.Hand;
            break;
        case "Help":
            DisplayArea.Cursor = Cursors.Help;
            break;
        case "IBeam":
            DisplayArea.Cursor = Cursors.IBeam;
            break;
        case "No":
            DisplayArea.Cursor = Cursors.No;
            break;
        case "None":
            DisplayArea.Cursor = Cursors.None;
            break;
        case "Pen":
            DisplayArea.Cursor = Cursors.Pen;
            break;
        case "ScrollSE":
            DisplayArea.Cursor = Cursors.ScrollSE;
            break;
        case "ScrollWE":
            DisplayArea.Cursor = Cursors.ScrollWE;
            break;
        case "SizeAll":
            DisplayArea.Cursor = Cursors.SizeAll;
            break;
        case "SizeNESW":
            DisplayArea.Cursor = Cursors.SizeNESW;
            break;
        case "SizeNS":
            DisplayArea.Cursor = Cursors.SizeNS;
            break;
        case "SizeNWSE":
            DisplayArea.Cursor = Cursors.SizeNWSE;
            break;
        case "SizeWE":
            DisplayArea.Cursor = Cursors.SizeWE;
            break;
        case "UpArrow":
            DisplayArea.Cursor = Cursors.UpArrow;
            break;
        case "WaitCursor":
            DisplayArea.Cursor = Cursors.Wait;
            break;
        case "Custom":
            DisplayArea.Cursor = CustomCursor;
            break;
        default:
```

```
        break;  
    }  
  
    // If the cursor scope is set to the entire application  
    // Use OverrideCursor to force the cursor for all elements  
    if (cursorScopeElementOnly == false)  
    {  
        Mouse.OverrideCursor = DisplayArea.Cursor;  
    }  
}
```

```

' When the Radiobox changes, a new cursor type is set
Private Sub CursorTypeChanged(ByVal sender As Object, ByVal e As SelectionChangedEventArgs)

    Dim item As String = CType(e.Source, ComboBox).SelectedItem.Content.ToString()

    Select Case item
        Case "AppStarting"
            DisplayArea.Cursor = Cursors.AppStarting
        Case "ArrowCD"
            DisplayArea.Cursor = Cursors.ArrowCD
        Case "Arrow"
            DisplayArea.Cursor = Cursors.Arrow
        Case "Cross"
            DisplayArea.Cursor = Cursors.Cross
        Case "HandCursor"
            DisplayArea.Cursor = Cursors.Hand
        Case "Help"
            DisplayArea.Cursor = Cursors.Help
        Case "IBeam"
            DisplayArea.Cursor = Cursors.IBeam
        Case "No"
            DisplayArea.Cursor = Cursors.No
        Case "None"
            DisplayArea.Cursor = Cursors.None
        Case "Pen"
            DisplayArea.Cursor = Cursors.Pen
        Case "ScrollSE"
            DisplayArea.Cursor = Cursors.ScrollSE
        Case "ScrollWE"
            DisplayArea.Cursor = Cursors.ScrollWE
        Case "SizeAll"
            DisplayArea.Cursor = Cursors.SizeAll
        Case "SizeNESW"
            DisplayArea.Cursor = Cursors.SizeNESW
        Case "SizeNS"
            DisplayArea.Cursor = Cursors.SizeNS
        Case "SizeNWSE"
            DisplayArea.Cursor = Cursors.SizeNWSE
        Case "SizeWE"
            DisplayArea.Cursor = Cursors.SizeWE
        Case "UpArrow"
            DisplayArea.Cursor = Cursors.UpArrow
        Case "WaitCursor"
            DisplayArea.Cursor = Cursors.Wait
        Case "Custom"
            DisplayArea.Cursor = CustomCursor
    End Select

    ' if the cursor scope is set to the entire application
    ' use OverrideCursor to force the cursor for all elements
    If (cursorScopeElementOnly = False) Then
        Mouse.OverrideCursor = DisplayArea.Cursor
    End If

End Sub

```

## See also

- [Input Overview](#)

# How to: Change the Color of an Element Using Focus Events

10/7/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to change the color of an element when it gains and loses focus by using the [GotFocus](#) and [LostFocus](#) events.

This example consists of a Extensible Application Markup Language (XAML) file and a code-behind file.

## Example

The following XAML creates the user interface, which consists of two [Button](#) objects, and attaches event handlers for the [GotFocus](#) and [LostFocus](#) events to the [Button](#) objects.

```
<StackPanel>
  <StackPanel.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Height" Value="20"/>
      <Setter Property="Width" Value="250"/>
      <Setter Property="HorizontalAlignment" Value="Left"/>
    </Style>
  </StackPanel.Resources>
  <Button
    GotFocus="OnGotFocusHandler"
    LostFocus="OnLostFocusHandler">Click Or Tab To Give Keyboard Focus</Button>
  <Button
    GotFocus="OnGotFocusHandler"
    LostFocus="OnLostFocusHandler">Click Or Tab To Give Keyborad Focus</Button>
</StackPanel>
```

The following code behind creates the [GotFocus](#) and [LostFocus](#) event handlers. When the [Button](#) gains keyboard focus, the [Background](#) of the [Button](#) is changed to red. When the [Button](#) loses keyboard focus, the [Background](#) of the [Button](#) is changed back to white.

```

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

    // Raised when Button gains focus.
    // Changes the color of the Button to Red.
    private void OnGotFocusHandler(object sender, RoutedEventArgs e)
    {
        Button tb = e.Source as Button;
        tb.Background = Brushes.Red;
    }

    // Raised when Button losses focus.
    // Changes the color of the Button back to white.
    private void OnLostFocusHandler(object sender, RoutedEventArgs e)
    {
        Button tb = e.Source as Button;
        tb.Background = Brushes.White;
    }
}

```

```

Partial Public Class Window1
Inherits Window

Public Sub New()
    InitializeComponent()
End Sub

'raised when Button gains focus. Changes the color of the Button to red.
Private Sub OnGotFocusHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim tb As Button = CType(e.Source, Button)
    tb.Background = Brushes.Red
End Sub

'raised when Button loses focus. Changes the color back to white.
Private Sub OnLostFocusHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim tb As Button = CType(e.Source, Button)
    tb.Background = Brushes.White
End Sub
End Class

```

## See also

- [Input Overview](#)

# How to: Apply a FocusVisualStyle to a Control

11/3/2019 • 2 minutes to read • [Edit Online](#)

This example shows you how to create a focus visual style in resources and apply the style to a control, using the [FocusVisualStyle](#) property.

## Example

The following example defines a style that creates additional control compositing that only applies when that control is keyboard focused in the user interface (UI). This is accomplished by defining a style with a [ControlTemplate](#), then referencing that style as a resource when setting the [FocusVisualStyle](#) property.

An external rectangle resembling a border is placed outside of the rectangular area. Unless otherwise modified, the sizing of the style uses the [ActualHeight](#) and [ActualWidth](#) of the rectangular control where the focus visual style is applied. This example sets negative values for the [Margin](#) to make the border appear slightly outside the focused control.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <Page.Resources>
        <Style x:Key="MyFocusVisual">
            <Setter Property="Control.Template">
                <Setter.Value>
                    <ControlTemplate>
                        <Rectangle Margin="-2" StrokeThickness="1" Stroke="Red" StrokeDashArray="1 2"/>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>
    <StackPanel Background="Ivory" Orientation="Horizontal">
        <Canvas Width="10"/>
        <Button Width="100" Height="30" FocusVisualStyle="{DynamicResource MyFocusVisual}">
            Focus Here</Button>
        <Canvas Width="100"/>
        <Button Width="100" Height="30" FocusVisualStyle="{DynamicResource MyFocusVisual}">
            Focus Here</Button>
    </StackPanel>
</Page>
```

A [FocusVisualStyle](#) is additive to any control template style that comes either from an explicit style or a theme style; the primary style for a control can still be created by using a [ControlTemplate](#) and setting that style to the [Style](#) property.

Focus visual styles should be used consistently across a theme or a UI, rather than using a different one for each focusable element. For details, see [Styling for Focus in Controls, and FocusVisualStyle](#).

## See also

- [FocusVisualStyle](#)
- [Styling and Templating](#)
- [Styling for Focus in Controls, and FocusVisualStyle](#)

# How to: Detect When the Enter Key Pressed

10/7/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to detect when the [Enter](#) key is pressed on the keyboard.

This example consists of a Extensible Application Markup Language (XAML) file and a code-behind file.

## Example

When the user presses the [Enter](#) key in the [TextBox](#), the input in the text box appears in another area of the user interface (UI).

The following XAML creates the user interface, which consists of a [StackPanel](#), a [TextBlock](#), and a [TextBox](#).

```
<StackPanel>
    <TextBlock Width="300" Height="20">
        Type some text into the TextBox and press the Enter key.
    </TextBlock>
    <TextBox Width="300" Height="30" Name="textBox1"
        KeyDown="OnKeyDownHandler"/>
    <TextBlock Width="300" Height="100" Name="textBlock1"/>
</StackPanel>
```

The following code behind creates the [KeyDown](#) event handler. If the key that is pressed is the [Enter](#) key, a message is displayed in the [TextBlock](#).

```
private void OnKeyDownHandler(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Return)
    {
        textBlock1.Text = "You Entered: " + textBox1.Text;
    }
}
```

```
Private Sub OnKeyDownHandler(ByVal sender As Object, ByVal e As KeyEventArgs)
    If (e.Key = Key.Return) Then
        textBlock1.Text = "You Entered: " + textBox1.Text
    End If
End Sub
```

## See also

- [Input Overview](#)
- [Routed Events Overview](#)

# How to: Create a Rollover Effect Using Events

11/3/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to change the color of an element as the mouse pointer enters and leaves the area occupied by the element.

This example consists of a Extensible Application Markup Language (XAML) file and a code-behind file.

## NOTE

This example demonstrates how to use events, but the recommended way to achieve this same effect is to use a [Trigger](#) in a style. For more information, see [Styling and Templating](#).

## Example

The following XAML creates the user interface, which consists of [Border](#) around a [TextBlock](#), and attaches the [MouseEnter](#) and [MouseLeave](#) event handlers to the [Border](#).

```
<StackPanel>
    <Border MouseEnter="OnMouseEnterHandler"
            MouseLeave="OnMouseLeaveHandler"
            Name="border1" Margin="10"
            BorderThickness="1"
            BorderBrush="Black"
            VerticalAlignment="Center"
            Width="300" Height="100">
        <Label Margin="10" FontSize="14"
              HorizontalAlignment="Center">Move Cursor Over Me</Label>
    </Border>
</StackPanel>
```

The following code behind creates the [MouseEnter](#) and [MouseLeave](#) event handlers. When the mouse pointer enters the [Border](#), the background of the [Border](#) is changed to red. When the mouse pointer leaves the [Border](#), the background of the [Border](#) is changed back to white.

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

    // raised when mouse cursor enters the area occupied by the element
    void OnMouseEnterHandler(object sender, MouseEventArgs e)
    {
        border1.Background = Brushes.Red;
    }

    // raised when mouse cursor leaves the area occupied by the element
    void OnMouseLeaveHandler(object sender, MouseEventArgs e)
    {
        border1.Background = Brushes.White;
    }
}
```

```
Partial Public Class Window1
    Inherits Window

    Public Sub New()
        InitializeComponent()
    End Sub

    ' raised when mouse cursor enters the are occupied by the element
    Private Sub OnMouseEnterHandler(ByVal sender As Object, ByVal e As MouseEventArgs)
        border1.Background = Brushes.Red
    End Sub

    ' raised when mouse cursor leaves the are occupied by the element
    Private Sub OnMouseLeaveHandler(ByVal sender As Object, ByVal e As MouseEventArgs)
        border1.Background = Brushes.White
    End Sub
End Class
```

# How to: Make an Object Follow the Mouse Pointer

10/7/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to change the dimensions of an object when the mouse pointer moves on the screen.

The example includes an Extensible Application Markup Language (XAML) file that creates the user interface (UI) and a code-behind file that creates the event handler.

## Example

The following XAML creates the UI, which consists of an [Ellipse](#) inside of a [StackPanel](#), and attaches the event handler for the [MouseMove](#) event.

```
<Window x:Class="WCSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="mouseMoveWithPointer"
    Height="400"
    Width="500"
    >
<Canvas MouseMove="MouseMoveHandler"
    Background="LemonChiffon">
    <Ellipse Name="ellipse" Fill="LightBlue"
        Width="100" Height="100"/>
</Canvas>
</Window>
```

The following code behind creates the [MouseMove](#) event handler. When the mouse pointer moves, the height and the width of the [Ellipse](#) are increased and decreased.

```
// raised when the mouse pointer moves.
// Expands the dimensions of an Ellipse when the mouse moves.
private void MouseMoveHandler(object sender, MouseEventArgs e)
{
    // Get the x and y coordinates of the mouse pointer.
    System.Windows.Point position = e.GetPosition(this);
    double pX = position.X;
    double pY = position.Y;

    // Sets the Height/Width of the circle to the mouse coordinates.
    ellipse.Width = pX;
    ellipse.Height = pY;
}
```

```
' raised when the mouse pointer moves.  
' Expands the dimensions of an Ellipse when the mouse moves.  
Private Sub OnMouseMoveHandler(ByVal sender As Object, ByVal e As MouseEventArgs)  
  
    'Get the x and y coordinates of the mouse pointer.  
    Dim position As System.Windows.Point  
    position = e.GetPosition(Me)  
    Dim pX As Double  
    pX = position.X  
    Dim pY As Double  
    pY = position.Y  
  
    'Set the Height and Width of the Ellipse to the mouse coordinates.  
    ellipse1.Height = pY  
    ellipse1.Width = pX  
End Sub
```

## See also

- [Input Overview](#)

# How to: Create a RoutedCommand

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to create a custom [RoutedCommand](#) and how to implement the custom command by creating a [ExecutedRoutedEventHandler](#) and a [CanExecuteRoutedEventArgs](#) and attaching them to a [CommandBinding](#). For more information on commanding, see the [Commanding Overview](#).

## Example

The first step in creating a [RoutedCommand](#) is defining the command and instantiating it.

```
public static RoutedCommand CustomRoutedCommand = new RoutedCommand();
```

```
Public Shared CustomRoutedCommand As New RoutedCommand()
```

In order to use the command in an application, event handlers which define what the command does must be created

```
private void ExecutedCustomCommand(object sender,
    ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Custom Command Executed");
}
```

```
Private Sub ExecutedCustomCommand(ByVal sender As Object, ByVal e As ExecutedRoutedEventArgs)
    MessageBox.Show("Custom Command Executed")
End Sub
```

```
// CanExecuteRoutedEventHandler that only returns true if
// the source is a control.
private void CanExecuteCustomCommand(object sender,
    CanExecuteRoutedEventArgs e)
{
    Control target = e.Source as Control;

    if(target != null)
    {
        e.CanExecute = true;
    }
    else
    {
        e.CanExecute = false;
    }
}
```

```

' CanExecuteRoutedEventHandler that only returns true if
' the source is a control.
Private Sub CanExecuteCustomCommand(ByVal sender As Object, ByVal e As CanExecuteRoutedEventArgs)
    Dim target As Control = TryCast(e.Source, Control)

    If target IsNot Nothing Then
        e.CanExecute = True
    Else
        e.CanExecute = False
    End If
End Sub

```

Next, a [CommandBinding](#) is created which associates the command with the event handlers. The [CommandBinding](#) is created on a specific object. This object defines the scope of the [CommandBinding](#) in the element tree

```

<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:SDKSamples"
    Height="600" Width="800"
    >
<Window.CommandBindings>
    <CommandBinding Command="{x:Static custom:Window1.CustomRoutedCommand}"
                    Executed="ExecutedCustomCommand"
                    CanExecute="CanExecuteCustomCommand" />
</Window.CommandBindings>

```

```

CommandBinding customCommandBinding = new CommandBinding(
    CustomRoutedCommand, ExecutedCustomCommand, CanExecuteCustomCommand);

// attach CommandBinding to root window
this.CommandBindings.Add(customCommandBinding);

```

```

Dim customCommandBinding As New CommandBinding(CustomRoutedCommand, AddressOf ExecutedCustomCommand, AddressOf
CanExecuteCustomCommand)

' attach CommandBinding to root window
Me.CommandBindings.Add(customCommandBinding)

```

The final step is invoking the command. One way to invoke a command is to associate it with a [ICommandSource](#), such as a [Button](#).

```

<StackPanel>
    <Button Command="{x:Static custom:Window1.CustomRoutedCommand}"
            Content="CustomRoutedCommand"/>
</StackPanel>

```

```

// create the ui
StackPanel CustomCommandStackPanel = new StackPanel();
Button CustomCommandButton = new Button();
CustomCommandStackPanel.Children.Add(CustomCommandButton);

CustomCommandButton.Command = CustomRoutedCommand;

```

```
' create the ui
Dim CustomCommandStackPanel As New StackPanel()
Dim CustomCommandButton As New Button()
CustomCommandStackPanel.Children.Add(CustomCommandButton)

CustomCommandButton.Command = CustomRoutedCommand
```

When the Button is clicked, the [Execute](#) method on the custom [RoutedCommand](#) is called. The [RoutedCommand](#) raises the [PreviewExecuted](#) and [Executed](#) routed events. These events traverse the element tree looking for a [CommandBinding](#) for this particular command. If a [CommandBinding](#) is found, the [ExecutedRoutedEventHandler](#) associated with [CommandBinding](#) is called.

## See also

- [RoutedCommand](#)
- [Commanding Overview](#)

# How to: Implement ICommandSource

11/3/2019 • 5 minutes to read • [Edit Online](#)

This example shows how to create a command source by implementing [ICommandSource](#). A command source is an object that knows how to invoke a command. The [ICommandSource](#) interface exposes three members: [Command](#), [CommandParameter](#), and [CommandTarget](#). [Command](#) is the command which will be invoked. The [CommandParameter](#) is a user-defined data type which is passed from the command source to the method which handles the command. The [CommandTarget](#) is the object that the command is being executed on.

In this example, a class is created which subclasses the [Slider](#) control and implements [ICommandSource](#).

## Example

WPF provides a number of classes which implement [ICommandSource](#), such as [Button](#), [MenuItem](#), and [ListBoxItem](#). A command source defines how it invokes a command. [Button](#) and [MenuItem](#) invoke a command when they are clicked. A [ListBoxItem](#) invokes a command when it is double clicked. These classes only become a command source when their [Command](#) property is set.

For this example we will invoke the command when the slider is moved, or more accurately, when the [Value](#) property is changed.

The following is the class definition.

```
public class CommandSlider : Slider, ICommandSource
{
    public CommandSlider() : base()
    {

    }
```

```
Public Class CommandSlider
    Inherits Slider
    Implements ICommandSource
    Public Sub New()
        MyBase.New()

    End Sub
```

The next step is to implement the [ICommandSource](#) members. In this example, the properties are implemented as [DependencyProperty](#) objects. This enables the properties to use data binding. For more information about the [DependencyProperty](#) class, see the [Dependency Properties Overview](#). For more information about data binding, see the [Data Binding Overview](#).

Only the [Command](#) property is shown here.

```

// Make Command a dependency property so it can use databinding.
public static readonly DependencyProperty CommandProperty =
    DependencyProperty.Register(
        "Command",
        typeof(ICommand),
        typeof(CommandSlider),
        new PropertyMetadata((ICommand)null,
            new PropertyChangedCallback(CommandChanged)));


public ICommand Command
{
    get
    {
        return (ICommand)GetValue(CommandProperty);
    }
    set
    {
        SetValue(CommandProperty, value);
    }
}

```

```

' Make Command a dependency property so it can use databinding.
Public Shared ReadOnly CommandProperty As DependencyProperty =
    DependencyProperty.Register("Command", GetType(ICommand),
        GetType(CommandSlider),
        New PropertyMetadata(CType(Nothing, ICommand),
            New PropertyChangedCallback(AddressOf CommandChanged)))


Public ReadOnly Property Command1() As ICommand Implements ICommandSource.Command
    Get
        Return CType(GetValue(CommandProperty), ICommand)
    End Get
End Property

Public Property Command() As ICommand
    Get
        Return CType(GetValue(CommandProperty), ICommand)
    End Get
    Set(ByVal value As ICommand)
        SetValue(CommandProperty, value)
    End Set
End Property

```

The following is the [DependencyProperty](#) change callback.

```

// Command dependency property change callback.
private static void CommandChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs e)
{
    CommandSlider cs = (CommandSlider)d;
    cs.HookUpCommand((ICommand)e.OldValue, (ICommand)e.NewValue);
}

```

```

' Command dependency property change callback.
Private Shared Sub CommandChanged(ByVal d As DependencyObject, ByVal e As DependencyPropertyChangedEventArgs)
    Dim cs As CommandSlider = CType(d, CommandSlider)
    cs.HookUpCommand(CType(e.OldValue, ICommand), CType(e.NewValue, ICommand))
End Sub

```

The next step is to add and remove the command which is associated with the command source. The [Command](#) property cannot simply be overwritten when a new command is added, because the event handlers associated

with the previous command, if there was one, must be removed first.

```
// Add a new command to the Command Property.  
private void HookUpCommand(ICommand oldCommand, ICommand newCommand)  
{  
    // If oldCommand is not null, then we need to remove the handlers.  
    if (oldCommand != null)  
    {  
        RemoveCommand(oldCommand, newCommand);  
    }  
    AddCommand(oldCommand, newCommand);  
}  
  
// Remove an old command from the Command Property.  
private void RemoveCommand(ICommand oldCommand, ICommand newCommand)  
{  
    EventHandler handler = CanExecuteChanged;  
    oldCommand.CanExecuteChanged -= handler;  
}  
  
// Add the command.  
private void AddCommand(ICommand oldCommand, ICommand newCommand)  
{  
    EventHandler handler = new EventHandler(CanExecuteChanged);  
    canExecuteChangedHandler = handler;  
    if (newCommand != null)  
    {  
        newCommand.CanExecuteChanged += canExecuteChangedHandler;  
    }  
}
```

```
' Add a new command to the Command Property.  
Private Sub HookUpCommand(ByVal oldCommand As ICommand, ByVal newCommand As ICommand)  
    ' If oldCommand is not null, then we need to remove the handlers.  
    If oldCommand IsNot Nothing Then  
        RemoveCommand(oldCommand, newCommand)  
    End If  
    AddCommand(oldCommand, newCommand)  
End Sub  
  
' Remove an old command from the Command Property.  
Private Sub RemoveCommand(ByVal oldCommand As ICommand, ByVal newCommand As ICommand)  
    Dim handler As EventHandler = AddressOf CanExecuteChanged  
    RemoveHandler oldCommand.CanExecuteChanged, handler  
End Sub  
  
' Add the command.  
Private Sub AddCommand(ByVal oldCommand As ICommand, ByVal newCommand As ICommand)  
    Dim handler As New EventHandler(AddressOf CanExecuteChanged)  
    canExecuteChangedHandler = handler  
    If newCommand IsNot Nothing Then  
        AddHandler newCommand.CanExecuteChanged, canExecuteChangedHandler  
    End If  
End Sub
```

The last step is to create logic for the [CanExecuteChanged](#) handler and the [Execute](#) method.

The [CanExecuteChanged](#) event notifies the command source that the ability of the command to execute on the current command target may have changed. When a command source receives this event, it typically calls the [CanExecute](#) method on the command. If the command cannot execute on the current command target, the command source will typically disable itself. If the command can execute on the current command target, the command source will typically enable itself.

```

private void CanExecuteChanged(object sender, EventArgs e)
{
    if (this.Command != null)
    {
        RoutedCommand command = this.Command as RoutedCommand;

        // If a RoutedCommand.
        if (command != null)
        {
            if (command.CanExecute(CommandParameter, CommandTarget))
            {
                this.IsEnabled = true;
            }
            else
            {
                this.IsEnabled = false;
            }
        }
        // If a not RoutedCommand.
        else
        {
            if (Command.CanExecute(CommandParameter))
            {
                this.IsEnabled = true;
            }
            else
            {
                this.IsEnabled = false;
            }
        }
    }
}

```

```

Private Sub CanExecuteChanged(ByVal sender As Object, ByVal e As EventArgs)

If Me.Command IsNot Nothing Then
    Dim command As RoutedCommand = TryCast(Me.Command, RoutedCommand)

    ' If a RoutedCommand.
    If command IsNot Nothing Then
        If command.CanExecute(CommandParameter, CommandTarget) Then
            Me.IsEnabled = True
        Else
            Me.IsEnabled = False
        End If
        ' If a not RoutedCommand.
    Else
        If Me.Command.CanExecute(CommandParameter) Then
            Me.IsEnabled = True
        Else
            Me.IsEnabled = False
        End If
    End If
End If
End Sub

```

The last step is the [Execute](#) method. If the command is a [RoutedCommand](#), the [RoutedCommand Execute](#) method is called; otherwise, the [ICommand Execute](#) method is called.

```

// If Command is defined, moving the slider will invoke the command;
// Otherwise, the slider will behave normally.
protected override void OnValueChanged(double oldValue, double newValue)
{
    base.OnValueChanged(oldValue, newValue);

    if (this.Command != null)
    {
        RoutedCommand command = Command as RoutedCommand;

        if (command != null)
        {
            command.Execute(CommandParameter, CommandTarget);
        }
        else
        {
            ((ICommand)Command).Execute(CommandParameter);
        }
    }
}

```

```

' If Command is defined, moving the slider will invoke the command;
' Otherwise, the slider will behave normally.
Protected Overrides Sub OnValueChanged(ByVal oldValue As Double, ByVal newValue As Double)
    MyBase.OnValueChanged(oldValue, newValue)

    If Me.Command IsNot Nothing Then
        Dim command As RoutedCommand = TryCast(Me.Command, RoutedCommand)

        If command IsNot Nothing Then
            command.Execute(CommandParameter, CommandTarget)
        Else
            CType(Me.Command, ICommand).Execute(CommandParameter)
        End If
    End If
End Sub

```

## See also

- [ICommandSource](#)
- [ICommand](#)
- [RoutedCommand](#)
- [Commanding Overview](#)

# How to: Hook Up a Command to a Control with No Command Support

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to hook up a [RoutedCommand](#) to a [Control](#) which does not have built in support for the command. For a complete sample which hooks up commands to multiple sources, see the [Create a Custom RoutedCommand Sample](#) sample.

## Example

Windows Presentation Foundation (WPF) provides a library of common commands which application programmers encounter regularly. The classes which comprise the command library are: [ApplicationCommands](#), [ComponentCommands](#), [NavigationCommands](#), [MediaCommands](#), and [EditingCommands](#).

The static [RoutedCommand](#) objects which make up these classes do not supply command logic. The logic for the command is associated with the command with a [CommandBinding](#). Many controls in WPF have built in support for some of the commands in the command library. [TextBox](#), for example, supports many of the application edit commands such as [Paste](#), [Copy](#), [Cut](#), [Redo](#), and [Undo](#). The application developer does not have to do anything special to get these commands to work with these controls. If the [TextBox](#) is the command target when the command is executed, it will handle the command using the [CommandBinding](#) that is built into the control.

The following shows how to use a [Button](#) as the command source for the [Open](#) command. A [CommandBinding](#) is created that associates the specified [CanExecuteRoutedEventHandler](#) and the [CanExecuteRoutedEventArgs](#) with the [RoutedCommand](#).

First, the command source is created. A [Button](#) is used as the command source.

```
<Button Command="ApplicationCommands.Open" Name="MyButton"
        Height="50" Width="200">
    Open (KeyBindings: Ctrl+R, Ctrl+0)
</Button>
```

```
// Button used to invoke the command
Button CommandButton = new Button();
CommandButton.Command = ApplicationCommands.Open;
CommandButton.Content = "Open (KeyBindings: Ctrl-R, Ctrl-0)";
MainStackPanel.Children.Add(CommandButton);
```

```
' Button used to invoke the command
Dim CommandButton As New Button()
CommandButton.Command = ApplicationCommands.Open
CommandButton.Content = "Open (KeyBindings: Ctrl-R, Ctrl-0)"
MainStackPanel.Children.Add(CommandButton)
```

Next, the [ExecutedRoutedEventHandler](#) and the [CanExecuteRoutedEventArgs](#) are created. The [ExecutedRoutedEventArgs](#) simply opens a [MessageBox](#) to signify that the command executed. The [CanExecuteRoutedEventArgs](#) sets the [CanExecute](#) property to `true`. Normally, the can execute handler would perform more robust checks to see if the command could execute on the current command target.

```

void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    String command, targetobj;
    command = ((RoutedCommand)e.Command).Name;
    targetobj = ((FrameworkElement)target).Name;
    MessageBox.Show("The " + command + " command has been invoked on target object " + targetobj);
}
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}

```

```

Private Sub OpenCmdExecuted(ByVal sender As Object, ByVal e As ExecutedRoutedEventArgs)
    Dim command, targetobj As String
    command = CType(e.Command, RoutedCommand).Name
    targetobj = CType(sender, FrameworkElement).Name
    MessageBox.Show("The " + command + " command has been invoked on target object " + targetobj)
End Sub
Private Sub OpenCmdCanExecute(ByVal sender As Object, ByVal e As CanExecuteRoutedEventArgs)
    e.CanExecute = True
End Sub

```

Finally, a [CommandBinding](#) is created on the root [Window](#) of the application that associates the routed events handlers to the [Open](#) command.

```

<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
                    Executed="OpenCmdExecuted"
                    CanExecute="OpenCmdCanExecute"/>
</Window.CommandBindings>

```

```

// Creating CommandBinding and attaching an Executed and CanExecute handler
CommandBinding OpenCmdBinding = new CommandBinding(
    ApplicationCommands.Open,
    OpenCmdExecuted,
    OpenCmdCanExecute);

this.CommandBindings.Add(OpenCmdBinding);

```

```

' Creating CommandBinding and attaching an Executed and CanExecute handler
Dim OpenCmdBinding As New CommandBinding(ApplicationCommands.Open, AddressOf OpenCmdExecuted, AddressOf
OpenCmdCanExecute)

Me.CommandBindings.Add(OpenCmdBinding)

```

## See also

- [Commanding Overview](#)
- [Hook Up a Command to a Control with Command Support](#)

# How to: Hook Up a Command to a Control with Command Support

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to hook up a [RoutedCommand](#) to a [Control](#) which has built in support for the command. For a complete sample which hooks up commands to multiple sources, see the [Create a Custom RoutedCommand Sample](#) sample.

## Example

Windows Presentation Foundation (WPF) provides a library of common commands which application programmers encounter regularly. The classes which comprise the command library are: [ApplicationCommands](#), [ComponentCommands](#), [NavigationCommands](#), [MediaCommands](#), and [EditingCommands](#).

The static [RoutedCommand](#) objects which make up these classes do not supply command logic. The logic for the command is associated with the command with a [CommandBinding](#). Some controls have built in [CommandBindings](#) for some commands. This mechanism allows the semantics of a command to stay the same, while the actual implementation can change. A [TextBox](#), for example, handles the [Paste](#) command differently than a control designed to support images, but the basic idea of what it means to paste something stays the same. The command logic cannot be supplied by the command, but rather must be supplied by the control or the application.

Many controls in WPF do have built in support for some of the commands in the command library. [TextBox](#), for example, supports many of the application edit commands such as [Paste](#), [Copy](#), [Cut](#), [Redo](#), and [Undo](#). The application developer does not have to do anything special to get these commands to work with these controls. If the [TextBox](#) is the command target when the command is executed, it will handle the command using the [CommandBinding](#) that is built into the control.

The following shows how to use a [MenuItem](#) as the command source for the [Paste](#) command, where a [TextBox](#) is the target of the command. All the logic that defines how the [TextBox](#) performs the paste is built into the [TextBox](#) control.

A [MenuItem](#) is created and its [Command](#) property is set to the [Paste](#) command. The [CommandTarget](#) is not explicitly set to the [TextBox](#) object. When the [CommandTarget](#) is not set, the target for the command is the element which has keyboard focus. If the element which has keyboard focus does not support the [Paste](#) command or cannot currently execute the paste command (the clipboard is empty, for example) then the [MenuItem](#) would be grayed out.

```

<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MenuItemCommandTask"
    >
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Command="ApplicationCommands.Paste" Width="75" />
        </Menu>
        <TextBox BorderBrush="Black" BorderThickness="2" Margin="25"
            TextWrapping="Wrap">
            The MenuItem will not be enabled until
            this TextBox gets keyboard focus
        </TextBox>
    </DockPanel>
</Window>

```

```

// Window1 constructor
public Window1()
{
    InitializeComponent();

    // Instantiating UIElements.
    DockPanel mainPanel = new DockPanel();
    Menu mainMenu = new Menu();
    MenuItem pasteMenuItem = new MenuItem();
    TextBox mainTextBox = new TextBox();

    // Associating the MenuItem with the Paste command.
    pasteMenuItem.Command = ApplicationCommands.Paste;

    // Setting properties on the TextBox.
    mainTextBox.Text =
        "The MenuItem will not be enabled until this TextBox receives keyboard focus.";
    mainTextBox.Margin = new Thickness(25);
    mainTextBox.BorderBrush = Brushes.Black;
    mainTextBox.BorderThickness = new Thickness(2);
    mainTextBox.TextWrapping = TextWrapping.Wrap;

    // Attaching UIElements to the Window.
    this.AddChild(mainPanel);
    mainMenu.Items.Add(pasteMenuItem);
    mainPanel.Children.Add(mainMenu);
    mainPanel.Children.Add(mainTextBox);

    // Defining DockPanel layout.
    DockPanel.SetDock(mainMenu, Dock.Top);
    DockPanel.SetDock(mainTextBox, Dock.Bottom);
}

```

```
' Window1 constructor
Public Sub New()
    InitializeComponent()

    ' Instantiating UIElements.
    Dim mainPanel As New DockPanel()
    Dim mainMenu As New Menu()
    Dim pasteMenuItem As New MenuItem()
    Dim mainTextBox As New TextBox()

    ' Associating the MenuItem with the Paste command.
    pasteMenuItem.Command = ApplicationCommands.Paste

    ' Setting properties on the TextBox.
    mainTextBox.Text = "The MenuItem will not be enabled until this TextBox receives keyboard focus."
    mainTextBox.Margin = New Thickness(25)
    mainTextBox.BorderBrush = Brushes.Black
    mainTextBox.BorderThickness = New Thickness(2)
    mainTextBox.TextWrapping = TextWrapping.Wrap

    ' Attaching UIElements to the Window.
    Me.AddChild(mainPanel)
    mainMenu.Items.Add(pasteMenuItem)
    mainPanel.Children.Add(mainMenu)
    mainPanel.Children.Add(mainTextBox)

    ' Defining DockPanel layout.
    DockPanel.SetDock(mainMenu, Dock.Top)
    DockPanel.SetDock(mainTextBox, Dock.Bottom)
End Sub
```

## See also

- [Commanding Overview](#)
- [Hook Up a Command to a Control with No Command Support](#)

# Digital Ink

3/5/2019 • 2 minutes to read • [Edit Online](#)

This section discusses the use of digital ink in the WPF. Traditionally found only in the Tablet PC SDK, digital ink is now available in the core Windows Presentation Foundation. This means you can now develop full-fledged Tablet PC applications by using the power of Windows Presentation Foundation.

## In This Section

[Overviews](#)

[How-to Topics](#)

## Related Sections

[Windows Presentation Foundation](#)

# Digital Ink Overviews

3/5/2019 • 2 minutes to read • [Edit Online](#)

## In This Section

[Getting Started with Ink](#)

[Collecting Ink](#)

[Handwriting Recognition](#)

[Storing Ink](#)

[The Ink Object Model: Windows Forms and COM versus WPF](#)

[Advanced Ink Handling](#)

# Get Started with Ink in WPF

10/25/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) has an ink feature that makes it easy to incorporate digital ink into your app.

## Prerequisites

To use the following examples, first install [Visual Studio](#). It also helps to know how to write basic WPF apps. For help getting started with WPF, see [Walkthrough: My first WPF desktop application](#).

## Quick Start

This section helps you write a simple WPF application that collects ink.

### Got Ink?

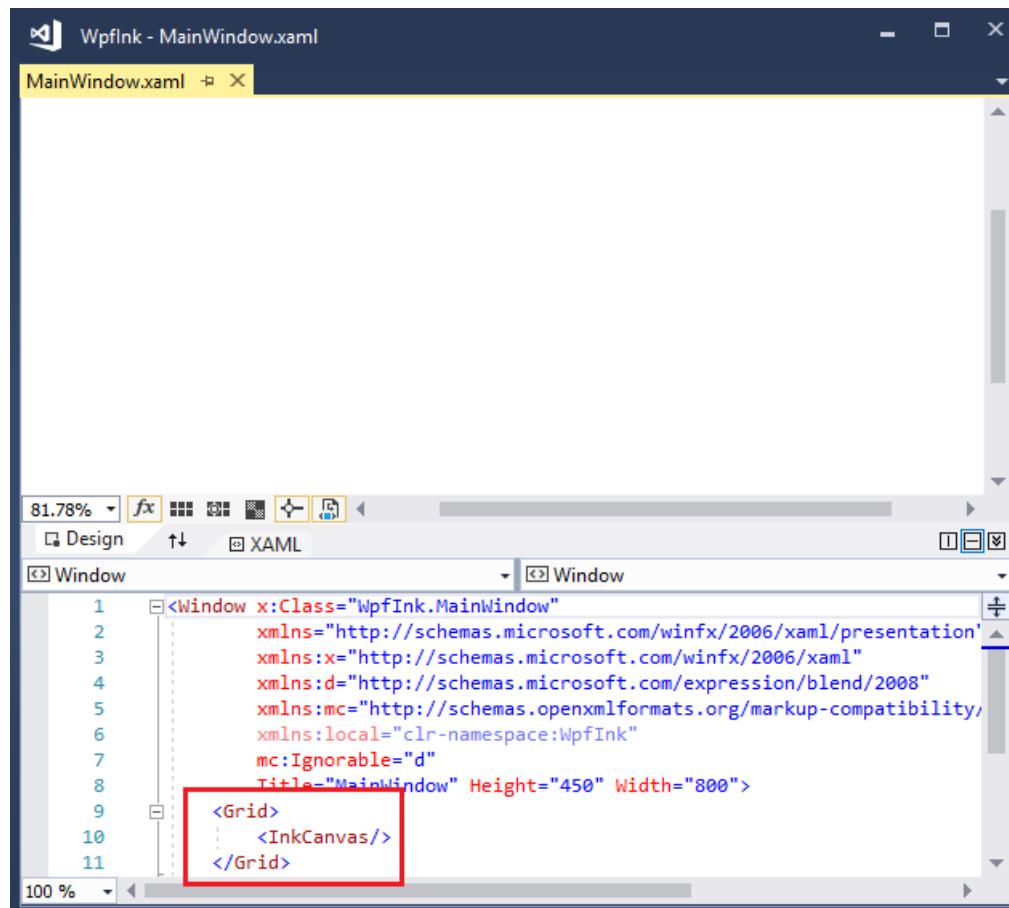
To create a WPF app that supports ink:

1. Open Visual Studio.
2. Create a new **WPF App**.

In the **New Project** dialog, expand the **Installed > Visual C# or Visual Basic > Windows Desktop** category. Then, select the **WPF App (.NET Framework)** app template. Enter a name, and then select **OK**.

Visual Studio creates the project, and *MainWindow.xaml* opens in the designer.

3. Type `<InkCanvas/>` between the `<Grid>` tags.



4. Press **F5** to launch your application in the debugger.
5. Using a stylus or mouse, write **hello world** in the window.

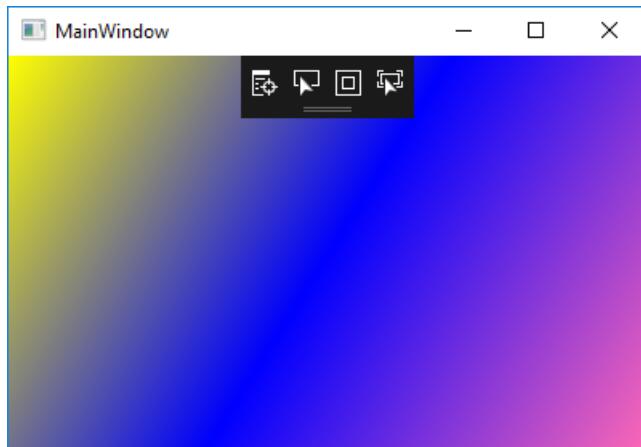
You've written the ink equivalent of a "hello world" application with only 12 keystrokes!

## Spice Up Your App

Let's take advantage of some features of the WPF. Replace everything between the opening and closing `<Window>` tags with the following markup:

```
<Page>
<InkCanvas Name="myInkCanvas" MouseRightButtonUp="RightMouseUpHandler">
    <InkCanvas.Background>
        <LinearGradientBrush>
            <GradientStop Color="Yellow" Offset="0.0" />
            <GradientStop Color="Blue" Offset="0.5" />
            <GradientStop Color="HotPink" Offset="1.0" />
        </LinearGradientBrush>
    </InkCanvas.Background>
</InkCanvas>
</Page>
```

This XAML creates a gradient brush background on your inking surface.



## Add Some Code Behind the XAML

While XAML makes it very easy to design the user interface, any real-world application needs to add code to handle events. Here is a simple example that zooms in on the ink in response to a right-click from a mouse.

1. Set the `MouseRightButtonUp` handler in your XAML:

```
<InkCanvas Name="myInkCanvas" MouseRightButtonUp="RightMouseUpHandler">
```

2. In **Solution Explorer**, expand `MainWindow.xaml` and open the code-behind file (`MainWindow.xaml.cs` or `MainWindow.xaml.vb`). Add the following event handler code:

```
private void RightMouseUpHandler(object sender,
                                System.Windows.Input.MouseEventArgs e)
{
    Matrix m = new Matrix();
    m.Scale(1.1d, 1.1d);
    ((InkCanvas)sender).Strokes.Transform(m, true);
}
```

```

Private Sub RightMouseUpHandler(ByVal sender As Object, _
                               ByVal e As System.Windows.Input.MouseButtonEventArgs)

    Dim m As New Matrix()
    m.Scale(1.1, 1.1)
    CType(sender, InkCanvas).Strokes.Transform(m, True)

End Sub

```

3. Run the application. Add some ink, and then right-click with the mouse or perform a press-and-hold equivalent with a stylus.

The display zooms in each time you click with the right mouse button.

### Use Procedural Code Instead of XAML

You can access all WPF features from procedural code. Follow these steps to create a "Hello Ink World" application for WPF that doesn't use any XAML at all.

1. Create a new console application project in Visual Studio.

In the **New Project** dialog, expand the **Installed > Visual C# or Visual Basic > Windows Desktop** category. Then, select the **Console App (.NET Framework)** app template. Enter a name, and then select **OK**.

2. Paste the following code into the Program.cs or Program.vb file:

```

using System;
using System.Windows;
using System.Windows.Controls;
class Program : Application
{
    Window win;
    InkCanvas ic;

    protected override void OnStartup(StartupEventArgs args)
    {
        base.OnStartup(args);
        win = new Window();
        ic = new InkCanvas();
        win.Content = ic;
        win.Show();
    }

    [STAThread]
    static void Main(string[] args)
    {
        new Program().Run();
    }
}

```

```

Imports System.Windows
Imports System.Windows.Controls

Class Program
    Inherits Application
    Private win As Window
    Private ic As InkCanvas

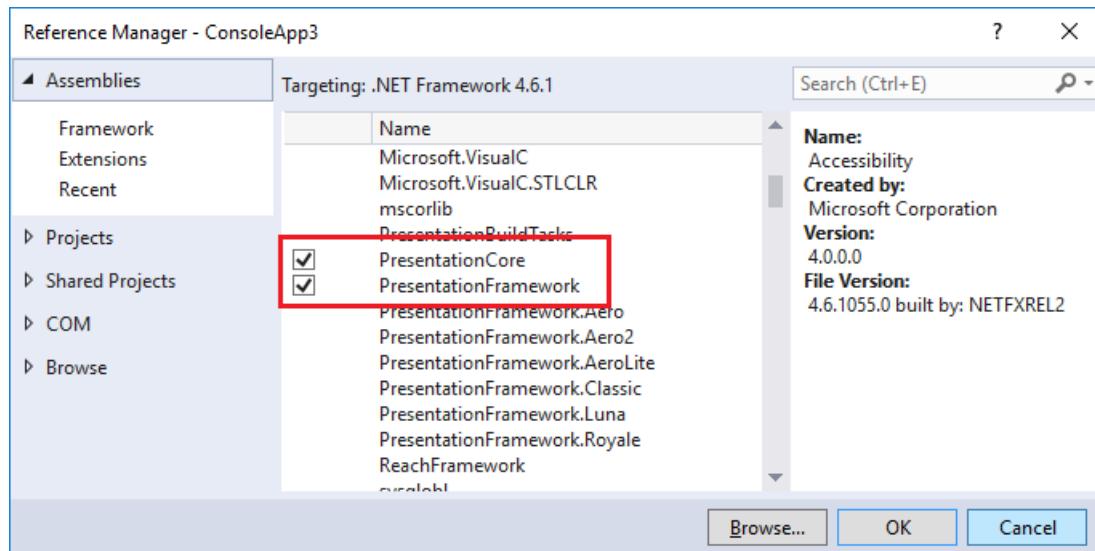
    Protected Overrides Sub OnStartup(ByVal args As StartupEventArgs)
        MyBase.OnStartup(args)
        win = New Window()
        ic = New InkCanvas()
        win.Content = ic
        win.Show()
    End Sub

End Class

Module Module1
    Sub Main()
        Dim prog As New Program()
        prog.Run()
    End Sub
End Module

```

3. Add references to the PresentationCore, PresentationFramework, and WindowsBase assemblies by right-clicking on **References** in **Solution Explorer** and choosing **Add Reference**.



4. Build the application by pressing **F5**.

## See also

- [Digital Ink](#)
- [Collecting Ink](#)
- [Handwriting Recognition](#)
- [Storing Ink](#)

# Collect Ink

11/7/2019 • 4 minutes to read • [Edit Online](#)

The [Windows Presentation Foundation](#) platform collects digital ink as a core part of its functionality. This topic discusses methods for collection of ink in Windows Presentation Foundation (WPF).

## Prerequisites

To use the following examples, you must first install Visual Studio and the Windows SDK. You should also understand how to write applications for the WPF. For more information about getting started with WPF, see [Walkthrough: My first WPF desktop application](#).

## Use the InkCanvas Element

The [System.Windows.Controls.InkCanvas](#) element provides the easiest way to collect ink in WPF. Use an **InkCanvas** element to receive and display ink input. You commonly input ink through the use of a stylus, which interacts with a digitizer to produce ink strokes. In addition, a mouse can be used in place of a stylus. The created strokes are represented as [Stroke](#) objects, and they can be manipulated both programmatically and by user input. The [InkCanvas](#) enables users to select, modify, or delete an existing [Stroke](#).

By using XAML, you can set up ink collection as easily as adding an **InkCanvas** element to your tree. The following example adds an [InkCanvas](#) to a default WPF project created in Visual Studio:

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

The **InkCanvas** element can also contain child elements, making it possible to add ink annotation capabilities to almost any type of XAML element. For example, to add inking capabilities to a text element, simply make it a child of an [InkCanvas](#):

```
<InkCanvas>
    <TextBlock>Show text here.</TextBlock>
</InkCanvas>
```

Adding support for marking up an image with ink is just as easy:

```
<InkCanvas>
    <Image Source="myPicture.jpg"/>
</InkCanvas>
```

### InkCollection Modes

The [InkCanvas](#) provides support for various input modes through its [EditMode](#) property.

### Manipulate Ink

The [InkCanvas](#) provides support for many ink editing operations. For example, [InkCanvas](#) supports back-of-pen erase, and no additional code is needed to add the functionality to the element.

#### Selection

Setting selection mode is as simple as setting the [InkCanvasEditingStyle](#) property to **Select**.

The following code sets the editing mode based on the value of a [CheckBox](#):

```
// Set the selection mode based on a checkbox
if ((bool)cbSelectionMode.IsChecked)
{
    theInkCanvas.EditingMode = InkCanvasEditingStyle.Select;
}
else
{
    theInkCanvas.EditingMode = InkCanvasEditingStyle.Ink;
}
```

```
' Set the selection mode based on a checkbox
If CBool(cbSelectionMode.IsChecked) Then
    theInkCanvas.EditingMode = InkCanvasEditingStyle.Select
Else
    theInkCanvas.EditingMode = InkCanvasEditingStyle.Ink
End If
```

### DrawingAttributes

Use the [DrawingAttributes](#) property to change the appearance of ink strokes. For instance, the [Color](#) member of [DrawingAttributes](#) sets the color of the rendered [Stroke](#).

The following example changes the color of the selected strokes to red:

```
// Get the selected strokes from the InkCanvas
StrokeCollection selection = theInkCanvas.GetSelectedStrokes();

// Check to see if any strokes are actually selected
if (selection.Count > 0)
{
    // Change the color of each stroke in the collection to red
    foreach (System.Windows.Ink.Stroke stroke in selection)
    {
        stroke.DrawingAttributes.Color = System.Windows.Media.Colors.Red;
    }
}
```

```
' Get the selected strokes from the InkCanvas
Dim selection As StrokeCollection = theInkCanvas.GetSelectedStrokes()

' Check to see if any strokes are actually selected
If selection.Count > 0 Then
    ' Change the color of each stroke in the collection to red
    Dim stroke As System.Windows.Ink.Stroke
    For Each stroke In selection
        stroke.DrawingAttributes.Color = System.Windows.Media.Colors.Red
    Next stroke
End If
```

### DefaultDrawingAttributes

The [DefaultDrawingAttributes](#) property provides access to properties such as the height, width, and color of the strokes to be created in an [InkCanvas](#). Once you change the [DefaultDrawingAttributes](#), all future strokes entered into the [InkCanvas](#) are rendered with the new property values.

In addition to modifying the [DefaultDrawingAttributes](#) in the code-behind file, you can use XAML syntax for specifying [DefaultDrawingAttributes](#) properties.

The next example demonstrates how to set the [Color](#) property. To use this code, create a new WPF project called

"HelloInkCanvas" in Visual Studio. Replace the code in the *MainWindow.xaml* file with the following code:

```
<Window x:Class="HelloInkCanvas.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:Ink="clr-namespace:System.Windows.Ink;assembly=PresentationCore"
    Title="Hello, InkCanvas!" Height="300" Width="300"
    >
<Grid>
    <InkCanvas Name="inkCanvas1" Background="Ivory">
        <InkCanvas.DefaultDrawingAttributes>
            <Ink:DrawingAttributes xmlns:ink="system-windows-ink" Color="Red" Width="5" />
        </InkCanvas.DefaultDrawingAttributes>
    </InkCanvas>

    <!-- This stack panel of buttons is a sibling to InkCanvas (not a child) but overlapping it,
        higher in z-order, so that ink is collected and rendered behind -->
    <StackPanel Name="buttonBar" VerticalAlignment="Top" Height="26" Orientation="Horizontal" Margin="5">
        <Button Click="Ink">Ink</Button>
        <Button Click="Highlight">Highlight</Button>
        <Button Click="EraseStroke">EraseStroke</Button>
        <Button Click="Select">Select</Button>
    </StackPanel>
</Grid>
</Window>
```

Next, add the following button event handlers to the code behind file, inside the *MainWindow* class:

```
// Set the EditingMode to ink input.
private void Ink(object sender, RoutedEventArgs e)
{
    inkCanvas1.EditingMode = InkCanvasEditingMode.Ink;

    // Set the DefaultDrawingAttributes for a red pen.
    inkCanvas1.DefaultDrawingAttributes.Color = Colors.Red;
    inkCanvas1.DefaultDrawingAttributes.IsHighlighter = false;
    inkCanvas1.DefaultDrawingAttributes.Height = 2;
}

// Set the EditingMode to highlighter input.
private void Highlight(object sender, RoutedEventArgs e)
{
    inkCanvas1.EditingMode = InkCanvasEditingMode.Ink;

    // Set the DefaultDrawingAttributes for a highlighter pen.
    inkCanvas1.DefaultDrawingAttributes.Color = Colors.Yellow;
    inkCanvas1.DefaultDrawingAttributes.IsHighlighter = true;
    inkCanvas1.DefaultDrawingAttributes.Height = 25;
}

// Set the EditingMode to erase by stroke.
private void EraseStroke(object sender, RoutedEventArgs e)
{
    inkCanvas1.EditingMode = InkCanvasEditingMode.EraseByStroke;
}

// Set the EditingMode to selection.
private void Select(object sender, RoutedEventArgs e)
{
    inkCanvas1.EditingMode = InkCanvasEditingMode.Select;
}
```

After copying this code, press **F5** in Visual Studio to run the program in the debugger.

Notice how the [StackPanel](#) places the buttons on top of the [InkCanvas](#). If you try to ink over the top of the buttons,

the [InkCanvas](#) collects and renders the ink behind the buttons. This is because the buttons are siblings of the [InkCanvas](#) as opposed to children. Also, the buttons are higher in the z-order, so the ink is rendered behind them.

## See also

- [DrawingAttributes](#)
- [DefaultDrawingAttributes](#)
- [System.Windows.Ink](#)

# Handwriting Recognition

8/22/2019 • 2 minutes to read • [Edit Online](#)

This section discusses the fundamentals of recognition as it pertains to digital ink in the WPF platform.

## Recognition Solutions

The following example shows how to recognize ink using the [Microsoft.Ink.InkCollector](#) class.

### NOTE

This sample requires that handwriting recognizers be installed on the system.

Create a new WPF application project in Visual Studio called **InkRecognition**. Replace the contents of the Window1.xaml file with the following XAML code. This code renders the application's user interface.

```
<Window x:Class="InkRecognition.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="InkRecognition"
    >
<Canvas Name="theRootCanvas">
    <Border
        Background="White"
        BorderBrush="Black"
        BorderThickness="2"
        Height="300"
        Width="300"
        Canvas.Top="10"
        Canvas.Left="10">
        <InkCanvas Name="theInkCanvas"></InkCanvas>
    </Border>
    <TextBox Name="textBox1"
        Height="25"
        Width="225"
        Canvas.Top="325"
        Canvas.Left="10"></TextBox>
    <Button
        Height="25"
        Width="75"
        Canvas.Top="325"
        Canvas.Left="235"
        Click="buttonClick">Recognize</Button>
        <Button x:Name="btnClear" Content="Clear Canvas" Canvas.Left="10" Canvas.Top="367" Width="75"
        Click="btnClear_Click"/>
    </Canvas>
</Window>
```

Add a reference to the Microsoft Ink assembly, Microsoft.Ink.dll, which can be found in \Program Files\Common Files\Microsoft Shared\Ink. Replace the contents of the code behind file with the following code.

```
using System.Windows;
using Microsoft.Ink;
using System.IO;

namespace InkRecognition
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>

    public partial class Window1 : Window
    {

        public Window1()
        {
            InitializeComponent();
        }

        // Recognizes handwriting by using RecognizerContext
        private void buttonClick(object sender, RoutedEventArgs e)
        {
            using (MemoryStream ms = new MemoryStream())
            {
                theInkCanvas.Strokes.Save(ms);
                var myInkCollector = new InkCollector();
                var ink = new Ink();
                ink.Load(ms.ToArray());

                using (RecognizerContext context = new RecognizerContext())
                {
                    if (ink.Strokes.Count > 0)
                    {
                        context.Strokes = ink.Strokes;
                        RecognitionStatus status;

                        var result = context.Recognize(out status);

                        if (status == RecognitionStatus.NoError)
                            textBox1.Text = result.TopString;
                        else
                            MessageBox.Show("Recognition failed");
                    }
                    else
                        MessageBox.Show("No stroke detected");
                }
            }
        }

        private void btnClear_Click(object sender, RoutedEventArgs e)
        {
            theInkCanvas.Strokes.Clear();
        }
    }
}
```

```

Imports System.Windows
Imports Microsoft.Ink
Imports System.IO

'<summary>
' Interaction logic for Window1.xaml
'</summary>

Namespace InkRecognition

    Class Window1
        Inherits Window

        Public Sub New()
            InitializeComponent()
        End Sub

        ' Recognizes handwriting by using RecognizerContext
        Private Sub buttonClick(ByVal sender As Object, ByVal e As RoutedEventArgs)

            Using ms As New MemoryStream()
                theInkCanvas.Strokes.Save(ms)
                Dim myInkCollector As InkCollector = New InkCollector()
                Dim ink As Ink = New Ink()
                ink.Load(ms.ToArray())

                Using context As New RecognizerContext()
                    If ink-strokes.Count > 0 Then
                        context.Strokes = ink-strokes
                        Dim status As RecognitionStatus

                        Dim result As RecognitionResult = context.Recognize(status)

                        If status = RecognitionStatus.NoError Then
                            textBox1.Text = result.TopString
                        Else
                            MessageBox.Show("Recognition failed")
                        End If
                    Else
                        MessageBox.Show("No stroke detected")
                    End If
                End Using
            End Using
        End Sub

        Private Sub btnClear_Click(sender As Object, e As RoutedEventArgs)
            theInkCanvas.Strokes.Clear()
        End Sub
    End Class
End Namespace

```

## See also

- [Microsoft.Ink.InkCollector](#)

# Storing Ink

4/8/2019 • 2 minutes to read • [Edit Online](#)

The [Save](#) methods provide support for storing ink as Ink Serialized Format (ISF). Constructors for the [StrokeCollection](#) class provide support for reading ink data.

## Ink Storage and Retrieval

This section discusses how to store and retrieve ink in the WPF platform.

The following example implements a button-click event handler that presents the user with a File Save dialog box and saves the ink from an [InkCanvas](#) out to a file.

```
private void buttonSaveAsClick(object sender, RoutedEventArgs e)
{
    SaveFileDialog saveFileDialog1 = new SaveFileDialog();
    saveFileDialog1.Filter = "isf files (*.isf)|*.isf";

    if (saveFileDialog1.ShowDialog() == true)
    {
        FileStream fs = new FileStream(saveFileDialog1.FileName,
                                         FileMode.Create);
        theInkCanvas.Strokes.Save(fs);
        fs.Close();
    }
}
```

```
Private Sub buttonSaveAsClick(ByVal sender As Object, ByVal e As RoutedEventArgs)

    Dim saveFileDialog1 As New SaveFileDialog()
    saveFileDialog1.Filter = "isf files (*.isf)|*.isf"

    If saveFileDialog1.ShowDialog() Then
        Dim fs As New FileStream(saveFileDialog1.FileName, FileMode.Create)
        theInkCanvas.Strokes.Save(fs)
        fs.Close()
    End If

End Sub
```

The following example implements a button-click event handler that presents the user with a File Open dialog box and reads ink from the file into an [InkCanvas](#) element.

```
private void buttonLoadClick(object sender, RoutedEventArgs e)
{
    OpenFileDialog openFileDialog1 = new OpenFileDialog();
    openFileDialog1.Filter = "isf files (*.isf)|*.isf";

    if (openFileDialog1.ShowDialog() == true)
    {
        FileStream fs = new FileStream(openFileDialog1.FileName,
                                         FileMode.Open);
        theInkCanvas.Strokes = new StrokeCollection(fs);
        fs.Close();
    }
}
```

```
Private Sub buttonLoadClick(ByVal sender As Object, ByVal e As RoutedEventArgs)

    Dim openFileDialog1 As New OpenFileDialog()
    openFileDialog1.Filter = "isf files (*.isf)|*.isf"

    If openFileDialog1.ShowDialog() Then
        Dim fs As New FileStream(openFileDialog1.FileName, FileMode.Open)
        theInkCanvas.Strokes = New StrokeCollection(fs)
        fs.Close()
    End If

End Sub
```

## See also

- [InkCanvas](#)
- [Windows Presentation Foundation](#)

# The Ink Object Model: Windows Forms and COM versus WPF

3/5/2019 • 6 minutes to read • [Edit Online](#)

There are essentially three platforms that support digital ink: the Tablet PC Windows Forms platform, the Tablet PC COM platform, and the Windows Presentation Foundation (WPF) platform. The Windows Forms and COM platforms share a similar object model, but the object model for the WPF platform is substantially different. This topic discusses the differences at a high-level so that developers that have worked with one object model can better understand the other.

## Enabling Ink in an Application

All three platforms ship objects and controls that enable an application to receive input from a tablet pen. The Windows Forms and COM platforms ship with [Microsoft.Ink.InkPicture](#), [Microsoft.Ink.InkEdit](#), [Microsoft.Ink.InkOverlay](#) and [Microsoft.Ink.InkCollector](#) classes. [Microsoft.Ink.InkPicture](#) and [Microsoft.Ink.InkEdit](#) are controls that you can add to an application to collect ink. The [Microsoft.Ink.InkOverlay](#) and [Microsoft.Ink.InkCollector](#) can be attached to an existing window to ink-enable windows and custom controls.

The WPF platform includes the [InkCanvas](#) control. You can add an [InkCanvas](#) to your application and begin collecting ink immediately. With the [InkCanvas](#), the user can copy, select, and resize ink. You can add other controls to the [InkCanvas](#), and the user can handwrite over those controls, too. You can create an ink-enabled custom control by adding an [InkPresenter](#) to it and collecting its stylus points.

The following table lists where to learn more about enabling ink in an application:

TO DO THIS...	ON THE WPF PLATFORM...	ON THE WINDOWS FORMS/COM PLATFORMS...
Add an ink-enabled control to an application	See <a href="#">Getting Started with Ink</a> .	See <a href="#">Auto Claims Form Sample</a>
Enable ink on a custom control	See <a href="#">Creating an Ink Input Control</a> .	See <a href="#">Ink Clipboard Sample</a> .

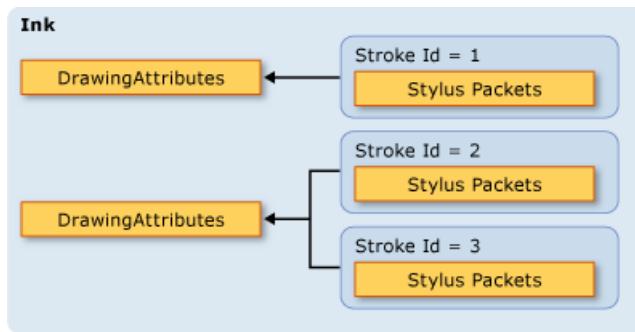
## Ink Data

On the Windows Forms and COM platforms, [Microsoft.Ink.InkCollector](#), [Microsoft.Ink.InkOverlay](#), [Microsoft.Ink.InkEdit](#), and [Microsoft.Ink.InkPicture](#) each expose a [Microsoft.Ink.Ink](#) object. The [Microsoft.Ink.Ink](#) object contains the data for one or more [Microsoft.Ink.Stroke](#) objects and exposes common methods and properties to manage and manipulate those strokes. The [Microsoft.Ink.Ink](#) object manages the lifetime of the strokes it contains; the [Microsoft.Ink.Ink](#) object creates and deletes the strokes that it owns. Each [Microsoft.Ink.Stroke](#) has an identifier that is unique within its parent [Microsoft.Ink.Ink](#) object.

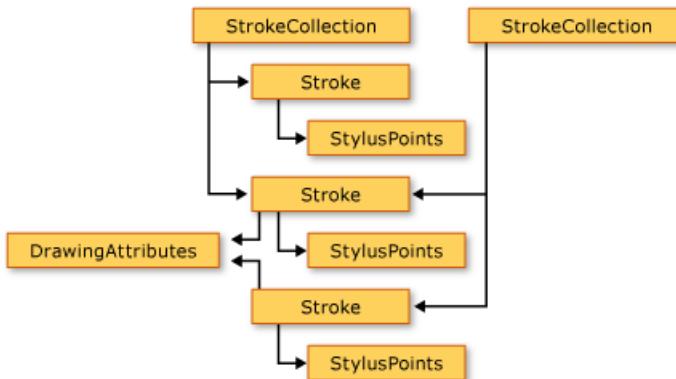
On the WPF platform, the [System.Windows.Ink.Stroke](#) class owns and manages its own lifetime. A group of [Stroke](#) objects can be collected together in a [StrokeCollection](#), which provides methods for common ink data management operations such as hit testing, erasing, transforming, and serializing the ink. A [Stroke](#) can belong to zero, one, or more [StrokeCollection](#) objects at any give time. Instead of having a [Microsoft.Ink.Ink](#) object, the [InkCanvas](#) and [InkPresenter](#) contain a [System.Windows.Ink.StrokeCollection](#).

The following pair of illustrations compares the ink data object models. On the Windows Forms and COM platforms, the [Microsoft.Ink.Ink](#) object constrains the lifetime of the [Microsoft.Ink.Stroke](#) objects, and the stylus

packets belong to the individual strokes. Two or more strokes can reference the same [Microsoft.Ink.DrawingAttributes](#) object, as shown in the following illustration.



On the WPF, each [System.Windows.Ink.Stroke](#) is a common language runtime object that exists as long as something has a reference to it. Each [Stroke](#) references a [StylusPointCollection](#) and [System.Windows.Ink.DrawingAttributes](#) object, which are also common language runtime objects.



The following table compares how to accomplish some common tasks on the WPF platform and the Windows Forms and COM platforms.

TASK	WINDOWS PRESENTATION FOUNDATION	WINDOWS FORMS AND COM
Save Ink	<code>Save</code>	<a href="#">Microsoft.Ink.Ink.Save</a>
Load Ink	Create a <a href="#">StrokeCollection</a> with the <a href="#">StrokeCollection</a> constructor.	<a href="#">Microsoft.Ink.Ink.Load</a>
Hit test	<code>HitTest</code>	<a href="#">Microsoft.Ink.Ink.HitTest</a>
Copy Ink	<code>CopySelection</code>	<a href="#">Microsoft.Ink.Ink.ClipboardCopy</a>
Paste Ink	<code>Paste</code>	<a href="#">Microsoft.Ink.Ink.ClipboardPaste</a>
Access custom properties on a collection of strokes	<code>AddPropertyData</code> (the properties are stored internally and accessed via <code>AddPropertyData</code> , <code>RemovePropertyData</code> , and <code>ContainsPropertyData</code> )	Use <a href="#">Microsoft.Ink.Ink.ExtendedProperties</a>

### Sharing ink between platforms

Although the platforms have different object models for the ink data, sharing the data between the platforms is very easy. The following examples save ink from a Windows Forms application and load the ink into a Windows Presentation Foundation application.

```
using Microsoft.Ink;
using System.Drawing;
```

```
Imports Microsoft.Ink
Imports System.Drawing
```

```
/// <summary>
/// Saves the digital ink from a Windows Forms application.
/// </summary>
/// <param name="inkToSave">An Ink object that contains the
/// digital ink.</param>
/// <returns>A MemoryStream containing the digital ink.</returns>
MemoryStream SaveInkInWinforms(Ink inkToSave)
{
    byte[] savedInk = inkToSave.Save();

    return (new MemoryStream(savedInk));

}
```

```
'/ <summary>
'/ Saves the digital ink from a Windows Forms application.
'/ </summary>
'/ <param name="inkToSave">An Ink object that contains the
'/ digital ink.</param>
'/ <returns>A MemoryStream containing the digital ink.</returns>
Function SaveInkInWinforms(ByVal inkToSave As Ink) As MemoryStream
    Dim savedInk As Byte() = inkToSave.Save()

    Return New MemoryStream(savedInk)

End Function 'SaveInkInWinforms
```

```
using System.Windows.Ink;
```

```
Imports System.Windows.Ink
```

```
/// <summary>
/// Loads digital ink into a StrokeCollection, which can be
/// used by a WPF application.
/// </summary>
/// <param name="savedInk">A MemoryStream containing the digital ink.</param>
public void LoadInkInWPF(MemoryStream inkStream)
{
    strokes = new StrokeCollection(inkStream);
}
```

```

' / <summary>
' / Loads digital ink into a StrokeCollection, which can be
' / used by a WPF application.
' / </summary>
' / <param name="savedInk">A MemoryStream containing the digital ink.</param>
Public Sub LoadInkInWPF(ByVal inkStream As MemoryStream)
    strokes = New StrokeCollection(inkStream)

End Sub

```

The following examples save ink from a Windows Presentation Foundation application and load the ink into a Windows Forms application.

```
using System.Windows.Ink;
```

```
Imports System.Windows.Ink
```

```

/// <summary>
/// Saves the digital ink from a WPF application.
/// </summary>
/// <param name="inkToSave">A StrokeCollection that contains the
/// digital ink.</param>
/// <returns>A MemoryStream containing the digital ink.</returns>
MemoryStream SaveInkInWPF(StrokeCollection strokesToSave)
{
    MemoryStream savedInk = new MemoryStream();

    strokesToSave.Save(savedInk);

    return savedInk;
}

```

```

' / <summary>
' / Saves the digital ink from a WPF application.
' / </summary>
' / <param name="inkToSave">A StrokeCollection that contains the
' / digital ink.</param>
' / <returns>A MemoryStream containing the digital ink.</returns>
Function SaveInkInWPF(ByVal strokesToSave As StrokeCollection) As MemoryStream
    Dim savedInk As New MemoryStream()

    strokesToSave.Save(savedInk)

    Return savedInk
End Function 'SaveInkInWPF

```

```
using Microsoft.Ink;
using System.Drawing;
```

```
Imports Microsoft.Ink
Imports System.Drawing
```

```

/// <summary>
/// Loads digital ink into a Windows Forms application.
/// </summary>
/// <param name="savedInk">A MemoryStream containing the digital ink.</param>
public void LoadInkInWinforms(MemoryStream savedInk)
{
    theInk = new Ink();
    theInk.Load(savedInk.ToArray());
}

```

```

'<summary>
' Loads digital ink into a Windows Forms application.
'</summary>
'<param name="savedInk">A MemoryStream containing the digital ink.</param>
Public Sub LoadInkInWinforms(ByVal savedInk As MemoryStream)
    theInk = New Ink()
    theInk.Load(savedInk.ToArray())
End Sub

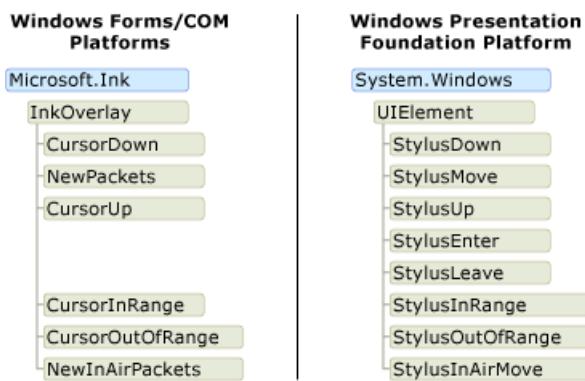
```

## Events from the Tablet Pen

The [Microsoft.Ink.InkOverlay](#), [Microsoft.Ink.InkCollector](#), and [Microsoft.Ink.InkPicture](#) on the Windows Forms and COM platforms receive events when the user inputs pen data. The [Microsoft.Ink.InkOverlay](#) or [Microsoft.Ink.InkCollector](#) is attached to a window or a control, and can subscribe to the events raised by the tablet input data. The thread on which these events occurs depends on whether the events are raised with a pen, a mouse, or programmatically. For more information about threading in relation to these events, see [General Threading Considerations](#) and [Threads on Which an Event Can Fire](#).

On the Windows Presentation Foundation platform, the [UIElement](#) class has events for pen input. This means that every control exposes the full set of stylus events. The stylus events have tunneling/bubbling event pairs and always occur on the application thread. For more information, see [Routed Events Overview](#).

The following diagram shows compares the object models for the classes that raise stylus events. The Windows Presentation Foundation object model shows only the bubbling events, not the tunneling event counterparts.



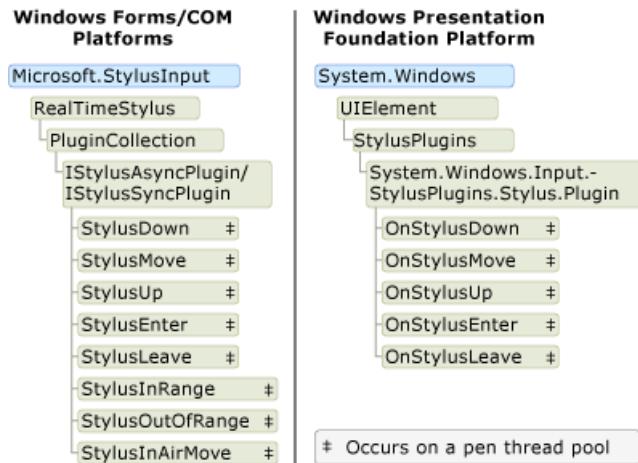
## Pen Data

All three platforms provide you with ways to intercept and manipulate the data that comes in from a tablet pen. On the Windows Forms and COM Platforms, this is achieved by creating a [Microsoft.StylusInput.RealTimeStylus](#), attaching a window or control to it, and creating a class that implements the [Microsoft.StylusInput.IStylusSyncPlugin](#) or [Microsoft.StylusInput.IStylusAsyncPlugin](#) interface. The custom plug-in is then added to the plug-in collection of the [Microsoft.StylusInput.RealTimeStylus](#). For more information about this object model, see [Architecture of the StylusInput APIs](#).

On the WPF platform, the [UIElement](#) class exposes a collection of plug-ins, similar in design to the [Microsoft.StylusInput.RealTimeStylus](#). To intercept pen data, create a class that inherits from [StylusPlugIn](#) and add the object to the [StylusPlugins](#) collection of the [UIElement](#). For more information about this interaction, see [Intercepting Input from the Stylus](#).

On all platforms, a thread pool receives the ink data via stylus events and sends it to the application thread. For more information about threading on the COM and Windows Platforms, see [Threading Considerations for the StylusInput APIs](#). For more information about threading on the Windows Presentation Software, see [The Ink Threading Model](#).

The following illustration compares the object models for the classes that receive pen data on the pen thread pool.



# Advanced Ink Handling

3/5/2019 • 2 minutes to read • [Edit Online](#)

The WPF ships with the [InkCanvas](#), and is an element you can put in your application to immediately start collecting and displaying ink. However, if the [InkCanvas](#) control does not provide a fine enough level of control, you can maintain control at a higher level by customizing your own ink collection and ink rendering classes using [System.Windows.Input.StylusPlugIns](#).

The [System.Windows.Input.StylusPlugIns](#) classes provide a mechanism for implementing low-level control over [Stylus](#) input and dynamically rendering ink. The [StylusPlugIn](#) class provides a mechanism for you to implement custom behavior and apply it to the stream of data coming from the stylus device for optimal performance. The [DynamicRenderer](#), a specialized [StylusPlugIn](#), allows you to customize dynamically rendering ink data in real-time which means that the [DynamicRenderer](#) draws digital ink immediately as [StylusPoint](#) data is generated, so it appears to "flow" from the stylus device.

## In This Section

[Custom Rendering Ink](#)

[Intercepting Input from the Stylus](#)

[Creating an Ink Input Control](#)

[The Ink Threading Model](#)

# Custom Rendering Ink

4/28/2019 • 7 minutes to read • [Edit Online](#)

The [DrawingAttributes](#) property of a stroke allows you to specify the appearance of a stroke, such as its size, color, and shape, but there may be times that you want to customize the appearance beyond what [DrawingAttributes](#) allow. You may want to customize the appearance of ink by rendering in the appearance of an air brush, oil paint, and many other effects. The Windows Presentation Foundation (WPF) allows you to custom render ink by implementing a custom [DynamicRenderer](#) and [Stroke](#) object.

This topic contains the following subsections:

- [Architecture](#)
- [Implementing a Dynamic Renderer](#)
- [Implementing Custom Strokes](#)
- [Implementing a Custom InkCanvas](#)
- [Conclusion](#)

## Architecture

Ink rendering occurs two times; when a user writes ink to an inking surface, and again after the stroke is added to the ink-enabled surface. The [DynamicRenderer](#) renders the ink when the user moves the tablet pen on the digitizer, and the [Stroke](#) renders itself once it is added to an element.

There are three classes to implement when dynamically rendering ink.

1. **DynamicRenderer:** Implement a class that derives from [DynamicRenderer](#). This class is a specialized [StylusPlugIn](#) that renders the stroke as it is drawn. The [DynamicRenderer](#) does the rendering on a separate thread, so the inking surface appears to collect ink even when the application user interface (UI) thread is blocked. For more information about the threading model, see [The Ink Threading Model](#). To customize dynamically rendering a stroke, override the [OnDraw](#) method.
2. **Stroke:** Implement a class that derives from [Stroke](#). This class is responsible for static rendering of the [StylusPoint](#) data after it has been converted into a [Stroke](#) object. Override the [DrawCore](#) method to ensure that static rendering of the stroke is consistent with dynamic rendering.
3. **InkCanvas:** Implement a class that derives from [InkCanvas](#). Assign the customized [DynamicRenderer](#) to the [DynamicRenderer](#) property. Override the [OnStrokeCollected](#) method and add a custom stroke to the [Strokes](#) property. This ensures that the appearance of the ink is consistent.

## Implementing a Dynamic Renderer

Although the [DynamicRenderer](#) class is a standard part of WPF, to perform more specialized rendering, you must create a customized dynamic renderer that derives from the [DynamicRenderer](#) and override the [OnDraw](#) method.

The following example demonstrates a customized [DynamicRenderer](#) that draws ink with a linear gradient brush effect.

```
using System;
using System.Windows.Media;
using System.Windows;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Input;
using System.Windows.Ink;
```

```
Imports System.Windows.Media
Imports System.Windows
Imports System.Windows.Input.StylusPlugIns
Imports System.Windows.Input
Imports System.Windows.Ink
```

```
// A StylusPlugin that renders ink with a linear gradient brush effect.
class CustomDynamicRenderer : DynamicRenderer
{
    [ThreadStatic]
    static private Brush brush = null;

    [ThreadStatic]
    static private Pen pen = null;

    private Point prevPoint;

    protected override void OnStylusDown(RawStylusInput rawStylusInput)
    {
        // Allocate memory to store the previous point to draw from.
        prevPoint = new Point(double.NegativeInfinity, double.NegativeInfinity);
        base.OnStylusDown(rawStylusInput);
    }

    protected override void OnDraw(DrawingContext drawingContext,
                                   StylusPointCollection stylusPoints,
                                   Geometry geometry, Brush fillBrush)
    {
        // Create a new Brush, if necessary.
        brush ??= new LinearGradientBrush(Colors.Red, Colors.Blue, 20d);

        // Create a new Pen, if necessary.
        pen ??= new Pen(brush, 2d);

        // Draw linear gradient ellipses between
        // all the StylusPoints that have come in.
        for (int i = 0; i < stylusPoints.Count; i++)
        {
            Point pt = (Point)stylusPoints[i];
            Vector v = Point.Subtract(prevPoint, pt);

            // Only draw if we are at least 4 units away
            // from the end of the last ellipse. Otherwise,
            // we're just redrawing and wasting cycles.
            if (v.Length > 4)
            {
                // Set the thickness of the stroke based
                // on how hard the user pressed.
                double radius = stylusPoints[i].PressureFactor * 10d;
                drawingContext.DrawEllipse(brush, pen, pt, radius, radius);
                prevPoint = pt;
            }
        }
    }
}
```

```

' A StylusPlugin that renders ink with a linear gradient brush effect.
Class CustomDynamicRenderer
    Inherits DynamicRenderer
    <ThreadStatic()> _
    Private Shared brush As Brush = Nothing

    <ThreadStatic()> _
    Private Shared pen As Pen = Nothing

    Private prevPoint As Point

Protected Overrides Sub OnStylusDown(ByVal rawStylusInput As RawStylusInput)
    ' Allocate memory to store the previous point to draw from.
    prevPoint = New Point(Double.NegativeInfinity, Double.NegativeInfinity)
    MyBase.OnStylusDown(rawStylusInput)

End Sub

Protected Overrides Sub OnDraw(ByVal drawingContext As DrawingContext, _
                               ByVal stylusPoints As StylusPointCollection, _
                               ByVal geometry As Geometry, _
                               ByVal fillBrush As Brush)

    ' Create a new Brush, if necessary.
    If brush Is Nothing Then
        brush = New LinearGradientBrush(Colors.Red, Colors.Blue, 20.0)
    End If

    ' Create a new Pen, if necessary.
    If pen Is Nothing Then
        pen = New Pen(brush, 2.0)
    End If

    ' Draw linear gradient ellipses between
    ' all the StylusPoints that have come in.
    Dim i As Integer
    For i = 0 To stylusPoints.Count - 1

        Dim pt As Point = CType(stylusPoints(i), Point)
        Dim v As Vector = Point.Subtract(prevPoint, pt)

        ' Only draw if we are at least 4 units away
        ' from the end of the last ellipse. Otherwise,
        ' we're just redrawing and wasting cycles.
        If v.Length > 4 Then
            ' Set the thickness of the stroke based
            ' on how hard the user pressed.
            Dim radius As Double = stylusPoints(i).PressureFactor * 10.0
            drawingContext.DrawEllipse(brush, pen, pt, radius, radius)
            prevPoint = pt
        End If
    Next i

End Sub
End Class

```

## Implementing Custom Strokes

Implement a class that derives from [Stroke](#). This class is responsible for rendering [StylusPoint](#) data after it has been converted into a [Stroke](#) object. Override the [DrawCore](#) class to do the actual drawing.

Your Stroke class can also store custom data by using the [AddPropertyData](#) method. This data is stored with the stroke data when persisted.

The [Stroke](#) class can also perform hit testing. You can also implement your own hit testing algorithm by overriding the [HitTest](#) method in the current class.

The following C# code demonstrates a custom [Stroke](#) class that renders [StylusPoint](#) data as a 3-D stroke.

```
using System;
using System.Windows.Media;
using System.Windows;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Input;
using System.Windows.Ink;

Imports System.Windows.Media
Imports System.Windows
Imports System.Windows.Input.StylusPlugIns
Imports System.Windows.Input
Imports System.Windows.Ink

// A class for rendering custom strokes
class CustomStroke : Stroke
{
    Brush brush;
    Pen pen;

    public CustomStroke(StylusPointCollection stylusPoints)
        : base(stylusPoints)
    {
        // Create the Brush and Pen used for drawing.
        brush = new LinearGradientBrush(Colors.Red, Colors.Blue, 20d);
        pen = new Pen(brush, 2d);
    }

    protected override void DrawCore(DrawingContext drawingContext,
                                    DrawingAttributes drawingAttributes)
    {
        // Allocate memory to store the previous point to draw from.
        Point prevPoint = new Point(double.NegativeInfinity,
                                    double.NegativeInfinity);

        // Draw linear gradient ellipses between
        // all the StylusPoints in the Stroke.
        for (int i = 0; i < this.StylusPoints.Count; i++)
        {
            Point pt = (Point)this.StylusPoints[i];
            Vector v = Point.Subtract(prevPoint, pt);

            // Only draw if we are at least 4 units away
            // from the end of the last ellipse. Otherwise,
            // we're just redrawing and wasting cycles.
            if (v.Length > 4)
            {
                // Set the thickness of the stroke
                // based on how hard the user pressed.
                double radius = this.StylusPoints[i].PressureFactor * 10d;
                drawingContext.DrawEllipse(brush, pen, pt, radius, radius);
                prevPoint = pt;
            }
        }
    }
}
```

```

' A class for rendering custom strokes
Class CustomStroke
    Inherits Stroke
    Private brush As Brush
    Private pen As Pen

    Public Sub New(ByVal stylusPoints As StylusPointCollection)
        MyBase.New(stylusPoints)
        ' Create the Brush and Pen used for drawing.
        brush = New LinearGradientBrush(Colors.Red, Colors.Blue, 20.0)
        pen = New Pen(brush, 2.0)

    End Sub

    Protected Overrides Sub DrawCore(ByVal drawingContext As DrawingContext, _
                                    ByVal drawingAttributes As DrawingAttributes)

        ' Allocate memory to store the previous point to draw from.
        Dim prevPoint As New Point(Double.NegativeInfinity, Double.NegativeInfinity)

        ' Draw linear gradient ellipses between
        ' all the StylusPoints in the Stroke.
        Dim i As Integer
        For i = 0 To Me.StylusPoints.Count - 1
            Dim pt As Point = CType(Me.StylusPoints(i), Point)
            Dim v As Vector = Point.Subtract(prevPoint, pt)

            ' Only draw if we are at least 4 units away
            ' from the end of the last ellipse. Otherwise,
            ' we're just redrawing and wasting cycles.
            If v.Length > 4 Then
                ' Set the thickness of the stroke
                ' based on how hard the user pressed.
                Dim radius As Double = Me.StylusPoints(i).PressureFactor * 10.0
                drawingContext.DrawEllipse(brush, pen, pt, radius, radius)
                prevPoint = pt
            End If
        Next i

    End Sub
End Class

```

## Implementing a Custom InkCanvas

The easiest way to use your customized [DynamicRenderer](#) and stroke is to implement a class that derives from [InkCanvas](#) and uses these classes. The [InkCanvas](#) has a [DynamicRenderer](#) property that specifies how the stroke is rendered when the user is drawing it.

To custom render strokes on an [InkCanvas](#) do the following:

- Create a class that derives from the [InkCanvas](#).
- Assign your customized [DynamicRenderer](#) to the [InkCanvas.DynamicRenderer](#) property.
- Override the [OnStrokeCollected](#) method. In this method, remove the original stroke that was added to the [InkCanvas](#). Then create a custom stroke, add it to the [Strokes](#) property, and call the base class with a new [InkCanvasStrokeCollectedEventArgs](#) that contains the custom stroke.

The following C# code demonstrates a custom [InkCanvas](#) class that uses a customized [DynamicRenderer](#) and collects custom strokes.

```

public class CustomRenderingInkCanvas : InkCanvas
{
    CustomDynamicRenderer customRenderer = new CustomDynamicRenderer();

    public CustomRenderingInkCanvas() : base()
    {
        // Use the custom dynamic renderer on the
        // custom InkCanvas.
        this.DynamicRenderer = customRenderer;
    }

    protected override void OnStrokeCollected(InkCanvasStrokeCollectedEventArgs e)
    {
        // Remove the original stroke and add a custom stroke.
        this.Strokes.Remove(e.Stroke);
        CustomStroke customStroke = new CustomStroke(e.Stroke.StylusPoints);
        this.Strokes.Add(customStroke);

        // Pass the custom stroke to base class' OnStrokeCollected method.
        InkCanvasStrokeCollectedEventArgs args =
            new InkCanvasStrokeCollectedEventArgs(customStroke);
        base.OnStrokeCollected(args);
    }
}

```

An [InkCanvas](#) can have more than one [DynamicRenderer](#). You can add multiple [DynamicRenderer](#) objects to the [InkCanvas](#) by adding them to the [StylusPlugins](#) property.

## Conclusion

You can customize the appearance of ink by deriving your own [DynamicRenderer](#), [Stroke](#), and [InkCanvas](#) classes. Together, these classes ensure that the appearance of the stroke is consistent when the user draws the stroke and after it is collected.

## See also

- [Advanced Ink Handling](#)

# Intercepting Input from the Stylus

8/27/2019 • 4 minutes to read • [Edit Online](#)

The [System.Windows.Input.StylusPlugIns](#) architecture provides a mechanism for implementing low-level control over [Stylus](#) input and the creation of digital ink [Stroke](#) objects. The [StylusPlugIn](#) class provides a mechanism for you to implement custom behavior and apply it to the stream of data coming from the stylus device for the optimal performance.

This topic contains the following subsections:

- [Architecture](#)
- [Implementing Stylus Plug-ins](#)
- [Adding Your Plug-in to an InkCanvas](#)
- [Conclusion](#)

## Architecture

The [StylusPlugIn](#) is the evolution of the [StylusInput](#) APIs, described in [Accessing and Manipulating Pen Input](#), in the [Microsoft Windows XP Tablet PC Edition Software Development Kit 1.7](#).

Each [UIElement](#) has a [StylusPlugIns](#) property that is a [StylusPlugInCollection](#). You can add a [StylusPlugIn](#) to an element's [StylusPlugIns](#) property to manipulate [StylusPoint](#) data as it is generated. [StylusPoint](#) data consists of all the properties supported by the system digitizer, including the [X](#) and [Y](#) point data, as well as [PressureFactor](#) data.

Your [StylusPlugIn](#) objects are inserted directly into the stream of data coming from the [Stylus](#) device when you add the [StylusPlugIn](#) to the [StylusPlugIns](#) property. The order in which plug-ins are added to the [StylusPlugIns](#) collection dictates the order in which they will receive [StylusPoint](#) data. For example, if you add a filter plug-in that restricts input to a particular region, and then add a plug-in that recognizes gestures as they are written, the plug-in that recognizes gestures will receive filtered [StylusPoint](#) data.

## Implementing Stylus Plug-ins

To implement a plug-in, derive a class from [StylusPlugIn](#). This class is applied to the stream of data as it comes in from the [Stylus](#). In this class you can modify the values of the [StylusPoint](#) data.

### Caution

If a [StylusPlugIn](#) throws or causes an exception, the application will close. You should thoroughly test controls that consume a [StylusPlugIn](#) and only use a control if you are certain the [StylusPlugIn](#) will not throw an exception.

The following example demonstrates a plug-in that restricts the stylus input by modifying the [X](#) and [Y](#) values in the [StylusPoint](#) data as it comes in from the [Stylus](#) device.

```
using System;
using System.Windows.Media;
using System.Windows;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Input;
using System.Windows.Ink;
```

```
Imports System.Windows.Media
Imports System.Windows
Imports System.Windows.Input.StylusPlugIns
Imports System.Windows.Input
Imports System.Windows.Ink
```

```
// A StylusPlugin that restricts the input area.
class FilterPlugin : StylusPlugIn
{
    protected override void OnStylusDown(RawStylusInput rawStylusInput)
    {
        // Call the base class before modifying the data.
        base.OnStylusDown(rawStylusInput);

        // Restrict the stylus input.
        Filter(rawStylusInput);
    }

    protected override void OnStylusMove(RawStylusInput rawStylusInput)
    {
        // Call the base class before modifying the data.
        base.OnStylusMove(rawStylusInput);

        // Restrict the stylus input.
        Filter(rawStylusInput);
    }

    protected override void OnStylusUp(RawStylusInput rawStylusInput)
    {
        // Call the base class before modifying the data.
        base.OnStylusUp(rawStylusInput);

        // Restrict the stylus input
        Filter(rawStylusInput);
    }

    private void Filter(RawStylusInput rawStylusInput)
    {
        // Get the StylusPoints that have come in.
        StylusPointCollection stylusPoints = rawStylusInput.GetStylusPoints();

        // Modify the (X,Y) data to move the points
        // inside the acceptable input area, if necessary.
        for (int i = 0; i < stylusPoints.Count; i++)
        {
            StylusPoint sp = stylusPoints[i];
            if (sp.X < 50) sp.X = 50;
            if (sp.X > 250) sp.X = 250;
            if (sp.Y < 50) sp.Y = 50;
            if (sp.Y > 250) sp.Y = 250;
            stylusPoints[i] = sp;
        }

        // Copy the modified StylusPoints back to the RawStylusInput.
        rawStylusInput.SetStylusPoints(stylusPoints);
    }
}
```

```

' A StylusPlugin that restricts the input area.
Class FilterPlugin
    Inherits StylusPlugIn

    Protected Overrides Sub OnStylusDown(ByVal rawStylusInput As RawStylusInput)
        ' Call the base class before modifying the data.
        MyBase.OnStylusDown(rawStylusInput)

        ' Restrict the stylus input.
        Filter(rawStylusInput)
    End Sub

    Protected Overrides Sub OnStylusMove(ByVal rawStylusInput As RawStylusInput)
        ' Call the base class before modifying the data.
        MyBase.OnStylusMove(rawStylusInput)

        ' Restrict the stylus input.
        Filter(rawStylusInput)
    End Sub

    Protected Overrides Sub OnStylusUp(ByVal rawStylusInput As RawStylusInput)
        ' Call the base class before modifying the data.
        MyBase.OnStylusUp(rawStylusInput)

        ' Restrict the stylus input
        Filter(rawStylusInput)
    End Sub

    Private Sub Filter(ByVal rawStylusInput As RawStylusInput)
        ' Get the StylusPoints that have come in.
        Dim stylusPoints As StylusPointCollection = rawStylusInput.GetStylusPoints()

        ' Modify the (X,Y) data to move the points
        ' inside the acceptable input area, if necessary.
        Dim i As Integer
        For i = 0 To stylusPoints.Count - 1
            Dim sp As StylusPoint = stylusPoints(i)
            If sp.X < 50 Then
                sp.X = 50
            End If
            If sp.X > 250 Then
                sp.X = 250
            End If
            If sp.Y < 50 Then
                sp.Y = 50
            End If
            If sp.Y > 250 Then
                sp.Y = 250
            End If
            stylusPoints(i) = sp
        Next i

        ' Copy the modified StylusPoints back to the RawStylusInput.
        rawStylusInput.SetStylusPoints(stylusPoints)
    End Sub
End Class

```

## Adding Your Plug-in to an InkCanvas

The easiest way to use your custom plug-in is to implement a class that derives from `InkCanvas` and add it to the `StylusPlugIns` property.

The following example demonstrates a custom `InkCanvas` that filters the ink.

```
public class FilterInkCanvas : InkCanvas
{
    FilterPlugin filter = new FilterPlugin();

    public FilterInkCanvas()
        : base()
    {
        this.StylusPlugIns.Add(filter);
    }
}
```

If you add a `FilterInkCanvas` to your application and run it, you will notice that the ink isn't restricted to a region until after the user completes a stroke. This is because the `InkCanvas` has a `DynamicRenderer` property, which is a `StylusPlugIn` and is already a member of the `StylusPlugIns` collection. The custom `StylusPlugIn` you added to the `StylusPlugIns` collection receives the `StylusPoint` data after `DynamicRenderer` receives data. As a result, the `StylusPoint` data will not be filtered until after the user lifts the pen to end a stroke. To filter the ink as the user draws it, you must insert the `FilterPlugin` before the `DynamicRenderer`.

The following C# code demonstrates a custom `InkCanvas` that filters the ink as it is drawn.

```
public class DynamicallyFilteredInkCanvas : InkCanvas
{
    FilterPlugin filter = new FilterPlugin();

    public DynamicallyFilteredInkCanvas()
        : base()
    {
        int dynamicRenderIndex =
            this.StylusPlugIns.IndexOf(this.DynamicRenderer);

        this.StylusPlugIns.Insert(dynamicRenderIndex, filter);
    }
}
```

## Conclusion

By deriving your own `StylusPlugIn` classes and inserting them into `StylusPlugInCollection` collections, you can greatly enhance the behavior of your digital ink. You have access to the `StylusPoint` data as it is generated, giving you the opportunity to customize the `Stylus` input. Because you have such low-level access to the `StylusPoint` data, you can implement ink collection and rendering with optimal performance for your application.

## See also

- [Advanced Ink Handling](#)
- [Accessing and Manipulating Pen Input](#)

# Creating an Ink Input Control

4/28/2019 • 7 minutes to read • [Edit Online](#)

You can create a custom control that dynamically and statically renders ink. That is, render ink as a user draws a stroke, causing the ink to appear to "flow" from the tablet pen, and display ink after it is added to the control, either via the tablet pen, pasted from the Clipboard, or loaded from a file. To dynamically render ink, your control must use a [DynamicRenderer](#). To statically render ink, you must override the stylus event methods ([OnStylusDown](#), [OnStylusMove](#), and [OnStylusUp](#)) to collect [StylusPoint](#) data, create strokes, and add them to an [InkPresenter](#) (which renders the ink on the control).

This topic contains the following subsections:

- [How to: Collect Stylus Point Data and Create Ink Strokes](#)
- [How to: Enable Your Control to Accept Input from the Mouse](#)
- [Putting it together](#)
- [Using Additional Plug-ins and DynamicRenderers](#)
- [Conclusion](#)

## How to: Collect Stylus Point Data and Create Ink Strokes

To create a control that collects and manages ink strokes do the following:

1. Derive a class from [Control](#) or one of the classes derived from [Control](#), such as [Label](#).

```
using System;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Controls;
using System.Windows;
```

```
class InkControl : Label
{
```

```
}
```

2. Add an [InkPresenter](#) to the class and set the [Content](#) property to the new [InkPresenter](#).

```
InkPresenter ip;

public InkControl()
{
    // Add an InkPresenter for drawing.
    ip = new InkPresenter();
    this.Content = ip;
}
```

3. Attach the [RootVisual](#) of the [DynamicRenderer](#) to the [InkPresenter](#) by calling the [AttachVisuals](#) method,

and add the [DynamicRenderer](#) to the [StylusPlugIns](#) collection. This allows the [InkPresenter](#) to display the ink as the stylus point data is collected by your control.

```
public InkControl()
{
    // Add a dynamic renderer that
    // draws ink as it "flows" from the stylus.
    dr = new DynamicRenderer();
    ip.AttachVisuals(dr.RootVisual, dr.DrawingAttributes);
    this.StylusPlugIns.Add(dr);

}
```

4. Override the [OnStylusDown](#) method. In this method, capture the stylus with a call to [Capture](#). By capturing the stylus, your control will continue to receive [StylusMove](#) and [StylusUp](#) events even if the stylus leaves the control's boundaries. This is not strictly mandatory, but almost always desired for a good user experience. Create a new [StylusPointCollection](#) to gather [StylusPoint](#) data. Finally, add the initial set of [StylusPoint](#) data to the [StylusPointCollection](#).

```
protected override void OnStylusDown(StylusDownEventArgs e)
{
    // Capture the stylus so all stylus input is routed to this control.
    Stylus.Capture(this);

    // Allocate memory for the StylusPointsCollection and
    // add the StylusPoints that have come in so far.
    stylusPoints = new StylusPointCollection();
    StylusPointCollection eventPoints =
        e.GetStylusPoints(this, stylusPoints.Description);

    stylusPoints.Add(eventPoints);

}
```

5. Override the [OnStylusMove](#) method and add the [StylusPoint](#) data to the [StylusPointCollection](#) object that you created earlier.

```
protected override void OnStylusMove(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    // Add the StylusPoints that have come in since the
    // last call to OnStylusMove.
    StylusPointCollection newStylusPoints =
        e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(newStylusPoints);
}
```

6. Override the [OnStylusUp](#) method and create a new [Stroke](#) with the [StylusPointCollection](#) data. Add the new [Stroke](#) you created to the [Strokes](#) collection of the [InkPresenter](#) and release stylus capture.

```

protected override void OnStylusUp(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    // Add the StylusPoints that have come in since the
    // last call to OnStylusMove.
    StylusPointCollection newStylusPoints =
        e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(newStylusPoints);

    // Create a new stroke from all the StylusPoints since OnStylusDown.
    Stroke stroke = new Stroke(stylusPoints);

    // Add the new stroke to the Strokes collection of the InkPresenter.
    ip.Strokes.Add(stroke);

    // Clear the StylusPointsCollection.
    stylusPoints = null;

    // Release stylus capture.
    Stylus.Capture(null);
}

```

## How to: Enable Your Control to Accept Input from the Mouse

If you add the preceding control to your application, run it, and use the mouse as an input device, you will notice that the strokes are not persisted. To persist the strokes when the mouse is used as the input device do the following:

1. Override the [OnMouseLeftButtonDown](#) and create a new [StylusPointCollection](#) Get the position of the mouse when the event occurred and create a [StylusPoint](#) using the point data and add the [StylusPoint](#) to the [StylusPointCollection](#).

```

protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
{
    base.OnMouseLeftButtonDown(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    // Start collecting the points.
    stylusPoints = new StylusPointCollection();
    Point pt = eGetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
}

```

2. Override the [OnMouseMove](#) method. Get the position of the mouse when the event occurred and create a [StylusPoint](#) using the point data. Add the [StylusPoint](#) to the [StylusPointCollection](#) object that you created earlier.

```

protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    // Don't collect points unless the left mouse button
    // is down.
    if (e.LeftButton == MouseButtonState.Released ||
        stylusPoints == null)
    {
        return;
    }

    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
}

```

3. Override the [OnMouseLeftButtonUp](#) method. Create a new [Stroke](#) with the [StylusPointCollection](#) data, and add the new [Stroke](#) you created to the [Strokes](#) collection of the [InkPresenter](#).

```

protected override void OnMouseLeftButtonUp(MouseEventArgs e)
{
    base.OnMouseLeftButtonUp(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    if (stylusPoints == null)
    {
        return;
    }

    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));

    // Create a stroke and add it to the InkPresenter.
    Stroke stroke = new Stroke(stylusPoints);
    stroke.DrawingAttributes = dr.DrawingAttributes;
    ip.Strokes.Add(stroke);

    stylusPoints = null;
}

```

## Putting it together

The following example is a custom control that collects ink when the user uses either the mouse or the pen.

```
using System;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Controls;
using System.Windows;
```

```
// A control for managing ink input
class InkControl : Label
{
    InkPresenter ip;
    DynamicRenderer dr;

    // The StylusPointsCollection that gathers points
    // before Stroke from is created.
    StylusPointCollection stylusPoints = null;

    public InkControl()
    {
        // Add an InkPresenter for drawing.
        ip = new InkPresenter();
        this.Content = ip;

        // Add a dynamic renderer that
        // draws ink as it "flows" from the stylus.
        dr = new DynamicRenderer();
        ip.AttachVisuals(dr.RootVisual, dr.DrawingAttributes);
        this.StylusPlugIns.Add(dr);
    }

    static InkControl()
    {
        // Allow ink to be drawn only within the bounds of the control.
        Type owner = typeof(InkControl);
        ClipToBoundsProperty.OverrideMetadata(owner,
            new FrameworkPropertyMetadata(true));
    }

    protected override void OnStylusDown(StylusDownEventArgs e)
    {
        // Capture the stylus so all stylus input is routed to this control.
        Stylus.Capture(this);

        // Allocate memory for the StylusPointsCollection and
        // add the StylusPoints that have come in so far.
        stylusPoints = new StylusPointCollection();
        StylusPointCollection eventPoints =
            e.GetStylusPoints(this, stylusPoints.Description);

        stylusPoints.Add(eventPoints);
    }

    protected override void OnStylusMove(StylusEventArgs e)
    {
        if (stylusPoints == null)
        {
            return;
        }

        // Add the StylusPoints that have come in since the
        // last call to OnStylusMove.
        StylusPointCollection newStylusPoints =
            e.GetStylusPoints(this, stylusPoints.Description);
        stylusPoints.Add(newStylusPoints);
    }
}
```

```

}

protected override void OnStylusUp(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    // Add the StylusPoints that have come in since the
    // last call to OnStylusMove.
    StylusPointCollection newStylusPoints =
        e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(newStylusPoints);

    // Create a new stroke from all the StylusPoints since OnStylusDown.
    Stroke stroke = new Stroke(stylusPoints);

    // Add the new stroke to the Strokes collection of the InkPresenter.
    ip.Strokes.Add(stroke);

    // Clear the StylusPointsCollection.
    stylusPoints = null;

    // Release stylus capture.
    Stylus.Capture(null);
}

protected override void OnMouseLeftButtonDown(MouseEventArgs e)
{
    base.OnMouseLeftButtonDown(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    // Start collecting the points.
    stylusPoints = new StylusPointCollection();
    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
}

protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    // Don't collect points unless the left mouse button
    // is down.
    if (e.LeftButton == MouseButtonState.Released ||
        stylusPoints == null)
    {
        return;
    }

    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
}

```

```

protected override void OnMouseLeftButtonUp(MouseEventArgs e)
{
    base.OnMouseLeftButtonUp(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    if (stylusPoints == null)
    {
        return;
    }

    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));

    // Create a stroke and add it to the InkPresenter.
    Stroke stroke = new Stroke(stylusPoints);
    stroke.DrawingAttributes = dr.DrawingAttributes;
    ip.Strokes.Add(stroke);

    stylusPoints = null;
}

}

```

## Using Additional Plug-ins and DynamicRenderers

Like the InkCanvas, your custom control can have custom [StylusPlugIn](#) and additional [DynamicRenderer](#) objects. Add these to the [StylusPlugIns](#) collection. The order of the [StylusPlugIn](#) objects in the [StylusPlugInCollection](#) affects the appearance of the ink when it is rendered. Suppose you have a [DynamicRenderer](#) called `dynamicRenderer` and a custom [StylusPlugIn](#) called `translatePlugin` that offsets the ink from the tablet pen. If `translatePlugin` is the first [StylusPlugIn](#) in the [StylusPlugInCollection](#), and `dynamicRenderer` is the second, the ink that "flows" will be offset as the user moves the pen. If `dynamicRenderer` is first, and `translatePlugin` is second, the ink will not be offset until the user lifts the pen.

## Conclusion

You can create a control that collects and renders ink by overriding the stylus event methods. By creating your own control, deriving your own [StylusPlugIn](#) classes, and inserting them into [StylusPlugInCollection](#), you can implement virtually any behavior imaginable with digital ink. You have access to the [StylusPoint](#) data as it is generated, giving you the opportunity to customize [Stylus](#) input and render it on the screen as appropriate for your application. Because you have such low-level access to the [StylusPoint](#) data, you can implement ink collection and render it with optimal performance for your application.

## See also

- [Advanced Ink Handling](#)
- [Accessing and Manipulating Pen Input](#)

# The Ink Threading Model

4/28/2019 • 3 minutes to read • [Edit Online](#)

One of the benefits of ink on a Tablet PC is that it feels a lot like writing with a regular pen and paper. To accomplish this, the tablet pen collects input data at a much higher rate than a mouse does and renders the ink as the user writes. The application's user interface (UI) thread is not sufficient for collecting pen data and rendering ink, because it can become blocked. To solve this, a WPF application uses two additional threads when a user writes ink.

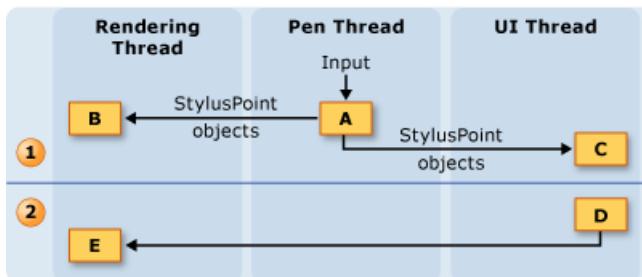
The following list describes the threads that take part in collecting and rendering digital ink:

- Pen thread - the thread that takes input from the stylus. (In reality, this is a thread pool, but this topic refers to it as a pen thread.)
  - Application user interface thread - the thread that controls the user interface of the application.
  - Dynamic rendering thread - the thread that renders the ink while the user draws a stroke. The dynamic rendering thread is different than the thread that renders other UI elements for the application, as mentioned in Window Presentation Foundation [Threading Model](#).

The inking model is the same whether the application uses the [InkCanvas](#) or a custom control similar to the one in [Creating an Ink Input Control](#). Although this topic discusses threading in terms of the [InkCanvas](#), the same concepts apply when you create a custom control.

# Threading Overview

The following diagram illustrates the threading model when a user draws a stroke:

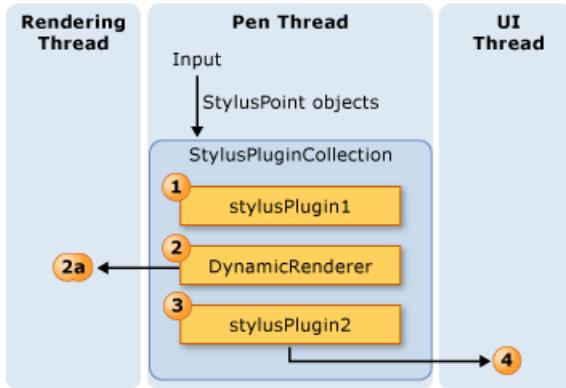


1. Actions occurring while the user draws the stroke
    - a. When the user draws a stroke, the stylus points come in on the pen thread. Stylus plug-ins, including the [DynamicRenderer](#), accept the stylus points on the pen thread and have the chance to modify them before the [InkCanvas](#) receives them.
    - b. The [DynamicRenderer](#) renders the stylus points on the dynamic rendering thread. This happens at the same time as the previous step.
    - c. The [InkCanvas](#) receives the stylus points on the UI thread.
  2. Actions occurring after the user ends the stroke
    - a. When the user finishes drawing the stroke, the [InkCanvas](#) creates a [Stroke](#) object and adds it to the [InkPresenter](#), which statically renders it.
    - b. The UI thread alerts the [DynamicRenderer](#) that the stroke is statically rendered, so the [DynamicRenderer](#) removes its visual representation of the stroke.

# Ink collection and Stylus Plug-ins

Each [UIElement](#) has a [StylusPlugInCollection](#). The [StylusPlugIn](#) objects in the [StylusPlugInCollection](#) receive and can modify the stylus points on the pen thread. The [StylusPlugIn](#) objects receive the stylus points according to their order in the [StylusPlugInCollection](#).

The following diagram illustrates the hypothetical situation where the [StylusPlugins](#) collection of a [UIElement](#) contains [stylusPlugin1](#), a [DynamicRenderer](#), and [stylusPlugin2](#), in that order.



In the previous diagram, the following behavior takes place:

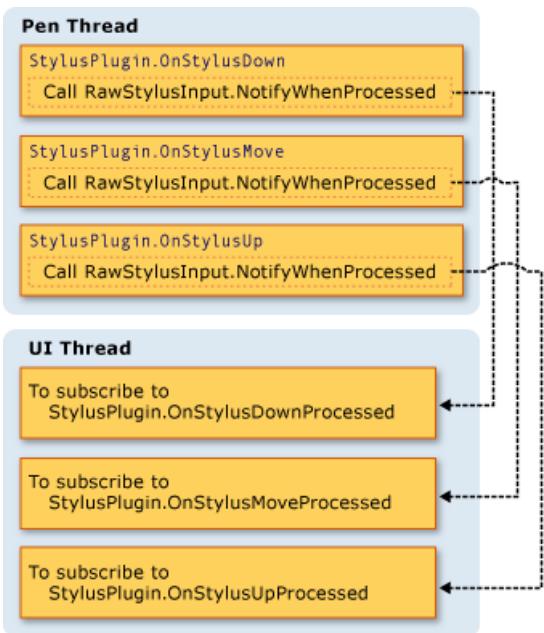
1. [stylusPlugin1](#) modifies the values for x and y.
2. [DynamicRenderer](#) receives the modified stylus points and renders them on the dynamic rendering thread.
3. [stylusPlugin2](#) receives the modified stylus points and further modifies the values for x and y.
4. The application collects the stylus points, and, when the user finishes the stroke, statically renders the stroke.

Suppose that [stylusPlugin1](#) restricts the stylus points to a rectangle and [stylusPlugin2](#) translates the stylus points to the right. In the previous scenario, the [DynamicRenderer](#) receives the restricted stylus points, but not the translated stylus points. When the user draws the stroke, the stroke is rendered within the bounds of the rectangle, but the stroke doesn't appear to be translated until the user lifts the pen.

## Performing operations with a Stylus Plug-in on the UI thread

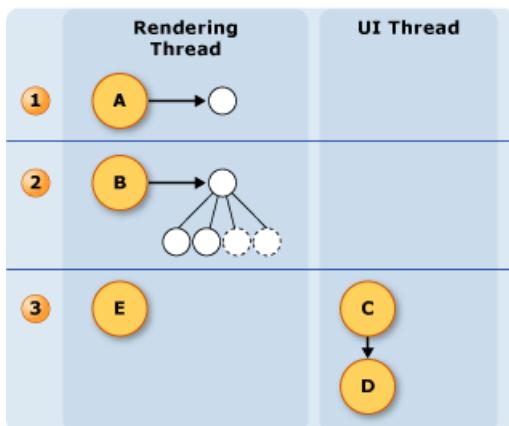
Because accurate hit-testing cannot be performed on the pen thread, some elements might occasionally receive stylus input intended for other elements. If you need to make sure the input was routed correctly before performing an operation, subscribe to and perform the operation in the [OnStylusDownProcessed](#), [OnStylusMoveProcessed](#), or [OnStylusUpProcessed](#) method. These methods are invoked by the application thread after accurate hit-testing has been performed. To subscribe to these methods, call the [NotifyWhenProcessed](#) method in the method that occurs on the pen thread.

The following diagram illustrates the relationship between the pen thread and UI thread with respect to the stylus events of a [StylusPlugIn](#).



## Rendering Ink

As the user draws a stroke, [DynamicRenderer](#) renders the ink on a separate thread so the ink appears to "flow" from the pen even when the UI thread is busy. The [DynamicRenderer](#) builds a visual tree on the dynamic rendering thread as it collects stylus points. When the user finishes the stroke, the [DynamicRenderer](#) asks to be notified when the application does the next rendering pass. After the application completes the next rendering pass, the [DynamicRenderer](#) cleans up its visual tree. The following diagram illustrates this process.



1. The user begins the stroke.
  - a. The [DynamicRenderer](#) creates the visual tree.
2. The user is drawing the stroke.
  - a. The [DynamicRenderer](#) builds the visual tree.
3. The user ends the stroke.
  - a. The [InkPresenter](#) adds the stroke to its visual tree.
  - b. The Media Integration Layer (MIL) statically renders the strokes.
  - c. The [DynamicRenderer](#) cleans up the visuals.

# Digital Ink How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

## In This Section

[Select Ink from a Custom Control](#)

[Add Custom Data to Ink Data](#)

[Erase Ink on a Custom Control](#)

[Recognize Application Gestures](#)

[Drag and Drop Ink](#)

[Data Bind to an InkCanvas](#)

[Analyze Ink with Analysis Hints](#)

[Rotate Ink](#)

[Disable the RealTimeStylus for WPF Applications](#)

# How to: Select Ink from a Custom Control

4/8/2019 • 15 minutes to read • [Edit Online](#)

By adding an [IncrementalLassoHitTester](#) to your custom control, you can enable your control so that a user can select ink with a lasso tool, similar to the way the [InkCanvas](#) selects ink with a lasso.

This example assumes you are familiar with creating an ink-enabled custom control. To create a custom control that accepts ink input, see [Creating an Ink Input Control](#).

## Example

When the user draws a lasso, the [IncrementalLassoHitTester](#) predicts which strokes will be within the lasso path's boundaries after the user completes the lasso. Strokes that are determined to be within the lasso path's boundaries can be thought of as being selected. Selected strokes can also become unselected. For example, if the user reverses direction while drawing the lasso, the [IncrementalLassoHitTester](#) may unselect some strokes.

The [IncrementalLassoHitTester](#) raises the [SelectionChanged](#) event, which enables your custom control to respond while the user is drawing the lasso. For example, you can change the appearance of strokes as the user selects and unselects them.

## Managing the Ink Mode

It is helpful to the user if the lasso appears differently than the ink on your control. To accomplish this, your custom control must keep track of whether the user is writing or selecting ink. The easiest way to do this is to declare an enumeration with two values: one to indicate that the user is writing ink and one to indicate that the user is selecting ink.

```
// Enum that keeps track of whether StrokeCollectionDemo is in ink mode
// or select mode.
public enum InkMode
{
    Ink, Select
}
```

```
' Enum that keeps track of whether StrokeCollectionDemo is in ink mode
' or select mode.
Public Enum InkMode
    Ink
    [Select]
End Enum 'InkMode
```

Next, add two [DrawingAttributes](#) to the class: one to use when the user writes ink, one to use when the user selects ink. In the constructor, initialize the [DrawingAttributes](#) and attach both [AttributeChanged](#) events to the same event handler. Then set the [DrawingAttributes](#) property of the [DynamicRenderer](#) to the ink [DrawingAttributes](#).

```
DrawingAttributes inkDA;
DrawingAttributes selectDA;
```

```
Private inkDA As DrawingAttributes  
Private selectDA As DrawingAttributes
```

```
// In the constructor.  
// Selection drawing attributes use dark gray ink.  
selectDA = new DrawingAttributes();  
selectDA.Color = Colors.DarkGray;  
  
// ink drawing attributes use default attributes  
inkDA = new DrawingAttributes();  
inkDA.Width = 5;  
inkDA.Height = 5;  
  
inkDA.AttributeChanged += new PropertyDataChangedEventHandler(DrawingAttributesChanged);  
selectDA.AttributeChanged += new PropertyDataChangedEventHandler(DrawingAttributesChanged);
```

```
' In the constructor.  
' Selection drawing attributes use dark gray ink.  
selectDA = New DrawingAttributes()  
selectDA.Color = Colors.DarkGray  
  
' ink drawing attributes use default attributes  
inkDA = New DrawingAttributes()  
inkDA.Width = 5  
inkDA.Height = 5  
  
AddHandler inkDA.AttributeChanged, _  
    AddressOf DrawingAttributesChanged  
  
AddHandler selectDA.AttributeChanged, _  
    AddressOf DrawingAttributesChanged
```

Add a property that exposes the selection mode. When the user changes the selection mode, set the [DrawingAttributes](#) property of the [DynamicRenderer](#) to the appropriate [DrawingAttributes](#) object and then reattach the [RootVisual](#) Property to the [InkPresenter](#).

```

// Property to indicate whether the user is inputting or
// selecting ink.
public InkMode Mode
{
    get
    {
        return mode;
    }

    set
    {
        mode = value;

        // Set the DrawingAttributes of the DynamicRenderer
        if (mode == InkMode.Ink)
        {
            renderer.DrawingAttributes = inkDA;
        }
        else
        {
            renderer.DrawingAttributes = selectDA;
        }

        // Reattach the visual of the DynamicRenderer to the InkPresenter.
        presenter.DetachVisuals(renderer.RootVisual);
        presenter.AttachVisuals(renderer.RootVisual, renderer.DrawingAttributes);
    }
}

```

```

' Property to indicate whether the user is inputting or
' selecting ink.
Public Property Mode() As InkMode
    Get
        Return Mode
    End Get

    Set(ByVal value As InkMode)
        modeState = value

        ' Set the DrawingAttributes of the DynamicRenderer
        If modeState = InkMode.Ink Then
            renderer.DrawingAttributes = inkDA
        Else
            renderer.DrawingAttributes = selectDA
        End If

        ' Reattach the visual of the DynamicRenderer to the InkPresenter.
        presenter.DetachVisuals(renderer.RootVisual)
        presenter.AttachVisuals(renderer.RootVisual, renderer.DrawingAttributes)
    End Set
End Property

```

Expose the [DrawingAttributes](#) as properties so applications can determine the appearance of the ink strokes and selection strokes.

```

// Property to allow the user to change the pen's DrawingAttributes.
public DrawingAttributes InkDrawingAttributes
{
    get
    {
        return inkDA;
    }
}

// Property to allow the user to change the Selector's newStroke DrawingAttributes.
public DrawingAttributes SelectDrawingAttributes
{
    get
    {
        return selectDA;
    }
}

```

```

' Property to allow the user to change the pen's DrawingAttributes.
Public ReadOnly Property InkDrawingAttributes() As DrawingAttributes
    Get
        Return inkDA
    End Get
End Property

' Property to allow the user to change the Selector's newStroke DrawingAttributes.
Public ReadOnly Property SelectDrawingAttributes() As DrawingAttributes
    Get
        Return selectDA
    End Get
End Property

```

When a property of a [DrawingAttributes](#) object changes, the [RootVisual](#) must be reattached to the [InkPresenter](#). In the event handler for the [AttributeChanged](#) event, reattach the [RootVisual](#) to the [InkPresenter](#).

```

void DrawingAttributesChanged(object sender, PropertyDataChangedEventArgs e)
{
    // Reattach the visual of the DynamicRenderer to the InkPresenter
    // whenever the DrawingAttributes change.
    presenter.DetachVisuals(renderer.RootVisual);
    presenter.AttachVisuals(renderer.RootVisual, renderer.DrawingAttributes);

}

```

```

Private Sub DrawingAttributesChanged(ByVal sender As Object, _
                                     ByVal e As PropertyDataChangedEventArgs)

    ' Reattach the visual of the DynamicRenderer to the InkPresenter
    ' whenever the DrawingAttributes change.
    presenter.DetachVisuals(renderer.RootVisual)
    presenter.AttachVisuals(renderer.RootVisual, _
                           renderer.DrawingAttributes)

End Sub

```

## Using the IncrementalLassoHitTester

Create and initialize a [StrokeCollection](#) that contains the selected strokes.

```
// StylusPointCollection that collects the stylus points from the stylus events.  
StylusPointCollection stylusPoints;
```

```
' StylusPointCollection that collects the stylus points from the stylus events.  
Private stylusPoints As StylusPointCollection
```

When the user starts to draw a stroke, either ink or the lasso, unselect any selected strokes. Then, if the user is drawing a lasso, create an [IncrementalLassoHitTester](#) by calling [GetIncrementalLassoHitTester](#), subscribe to the [SelectionChanged](#) event, and call [AddPoints](#). This code can be a separate method and called from the [OnStylusDown](#) and [OnMouseDown](#) methods.

```
private void InitializeHitTester(StylusPointCollection collectedPoints)  
{  
    // Deselect any selected strokes.  
    foreach (Stroke selectedStroke in selectedStrokes)  
    {  
        selectedStroke.DrawingAttributes.Color = inkDA.Color;  
    }  
    selectedStrokes.Clear();  
  
    if (mode == InkMode.Select)  
    {  
        // Remove the previously drawn lasso, if it exists.  
        if (lassoPath != null)  
        {  
            presenter.Strokes.Remove(lassoPath);  
            lassoPath = null;  
        }  
  
        selectionTester =  
            presenter.Strokes.GetIncrementalLassoHitTester(80);  
        selectionTester.SelectionChanged +=  
            new LassoSelectionChangedEventHandler(selectionTester_SelectionChanged);  
        selectionTester.AddPoints(collectedPoints);  
    }  
}
```

```
Private Sub InitializeHitTester(ByVal collectedPoints As StylusPointCollection)  
  
    ' Deselect any selected strokes.  
    Dim selectedStroke As Stroke  
    For Each selectedStroke In selectedStrokes  
        selectedStroke.DrawingAttributes.Color = inkDA.Color  
    Next selectedStroke  
    selectedStrokes.Clear()  
  
    If modeState = InkMode.Select Then  
        ' Remove the previously drawn lasso, if it exists.  
        If Not (lassoPath Is Nothing) Then  
            presenter.Strokes.Remove(lassoPath)  
            lassoPath = Nothing  
        End If  
  
        selectionTester = presenter.Strokes.GetIncrementalLassoHitTester(80)  
        AddHandler selectionTester.SelectionChanged, AddressOf selectionTester_SelectionChanged  
        selectionTester.AddPoints(collectedPoints)  
    End If  
  
End Sub
```

Add the stylus points to the [IncrementalLassoHitTester](#) while the user draws the lasso. Call the following method from the [OnStylusMove](#), [OnStylusUp](#), [OnMouseMove](#), and [OnMouseLeftButtonUp](#) methods.

```
private void AddPointsToHitTester(StylusPointCollection collectedPoints)
{
    if (mode == InkMode.Select &&
        selectionTester != null &&
        selectionTester.IsValid)
    {
        // When the control is selecting strokes, add the
        // stylus packetList to selectionTester.
        selectionTester.AddPoints(collectedPoints);

    }
}
```

```
Private Sub AddPointsToHitTester(ByVal collectedPoints As StylusPointCollection)

    If modeState = InkMode.Select AndAlso _
        Not selectionTester Is Nothing AndAlso _
        selectionTester.IsValid Then

        ' When the control is selecting strokes, add the
        ' stylus packetList to selectionTester.
        selectionTester.AddPoints(collectedPoints)
    End If

End Sub
```

Handle the [IncrementalLassoHitTester.SelectionChanged](#) event to respond when the user selects and unselects strokes. The [LassoSelectionChangedEventArgs](#) class has the [SelectedStrokes](#) and [DeselectedStrokes](#) properties that get the strokes that were selected and unselected, respectively.

```
void selectionTester_SelectionChanged(object sender,
    LassoSelectionChangedEventArgs args)
{
    // Change the color of all selected strokes to red.
    foreach (Stroke selectedStroke in args.SelectedStrokes)
    {
        selectedStroke.DrawingAttributes.Color = Colors.Red;
        selectedStrokes.Add(selectedStroke);

    }

    // Change the color of all unselected strokes to
    // their original color.
    foreach (Stroke unselectedStroke in args.DeselectedStrokes)
    {
        unselectedStroke.DrawingAttributes.Color = inkDA.Color;
        selectedStrokes.Remove(unselectedStroke);
    }
}
```

```

Private Sub selectionTester_SelectionChanged(ByVal sender As Object, _
                                         ByVal args As LassoSelectionChangedEventArgs)

    ' Change the color of all selected strokes to red.
    Dim selectedStroke As Stroke
    For Each selectedStroke In args.SelectedStrokes
        selectedStroke.DrawingAttributes.Color = Colors.Red
        selectedStrokes.Add(selectedStroke)
    Next selectedStroke

    ' Change the color of all unselected strokes to
    ' their original color.
    Dim unselectedStroke As Stroke
    For Each unselectedStroke In args.DeselectedStrokes
        unselectedStroke.DrawingAttributes.Color = inkDA.Color
        selectedStrokes.Remove(unselectedStroke)
    Next unselectedStroke

End Sub

```

When the user finishes drawing the lasso, unsubscribe from the [SelectionChanged](#) event and call [EndHitTesting](#).

```

if (mode == InkMode.Select && lassoPath == null)
{
    // Add the lasso to the InkPresenter and add the packetList
    // to selectionTester.
    lassoPath = newStroke;
    lassoPath.DrawingAttributes = selectDA.Clone();
    presenter.Strokes.Add(lassoPath);
    selectionTester.SelectionChanged -= new LassoSelectionChangedEventHandler
        (selectionTester_SelectionChanged);
    selectionTester.EndHitTesting();
}

```

```

If modeState = InkMode.Select AndAlso lassoPath Is Nothing Then
    ' Add the lasso to the InkPresenter and add the packetList
    ' to selectionTester.
    lassoPath = newStroke
    lassoPath.DrawingAttributes = selectDA.Clone()
    presenter.Strokes.Add(lassoPath)
    RemoveHandler selectionTester.SelectionChanged, _
        AddressOf selectionTester_SelectionChanged
    selectionTester.EndHitTesting()
End If

```

## Putting it All Together.

The following example is a custom control that enables a user to select ink with a lasso.

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Input;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Ink;

// Enum that keeps track of whether StrokeCollectionDemo is in ink mode
// or select mode.
public enum InkMode
{
    ...
}

```

```

    ink, Select
}

// This control allows the user to input and select ink. When the
// user selects ink, the lasso remains visible until they erase, or clip
// the selected strokes, or clear the selection. When the control is
// in selection mode, strokes that are selected turn red.
public class InkSelector : Label
{
    InkMode mode;

    DrawingAttributes inkDA;
    DrawingAttributes selectDA;

    InkPresenter presenter;
    IncrementalLassoHitTester selectionTester;
    StrokeCollection selectedStrokes = new StrokeCollection();

    // StylusPointCollection that collects the stylus points from the stylus events.
    StylusPointCollection stylusPoints;
    // Stroke that represents the lasso.
    Stroke lassoPath;

    DynamicRenderer renderer;

    public InkSelector()
    {
        mode = InkMode.Ink;

        // Use an InkPresenter to display the strokes on the custom control.
        presenter = new InkPresenter();
        this.Content = presenter;

        // In the constructor.
        // Selection drawing attributes use dark gray ink.
        selectDA = new DrawingAttributes();
        selectDA.Color = Colors.DarkGray;

        // ink drawing attributes use default attributes
        inkDA = new DrawingAttributes();
        inkDA.Width = 5;
        inkDA.Height = 5;

        inkDA.AttributeChanged += new PropertyDataChangedEventHandler(DrawingAttributesChanged);
        selectDA.AttributeChanged += new PropertyDataChangedEventHandler(DrawingAttributesChanged);

        // Add a DynmaicRenderer to the control so ink appears
        // to "flow" from the tablet pen.
        renderer = new DynamicRenderer();
        renderer.DrawingAttributes = inkDA;
        this.StylusPlugIns.Add(renderer);
        presenter.AttachVisuals(renderer.RootVisual,
            renderer.DrawingAttributes);
    }

    static InkSelector()
    {
        // Allow ink to be drawn only within the bounds of the control.
        Type owner = typeof(InkSelector);
        ClipToBoundsProperty.OverrideMetadata(owner,
            new FrameworkPropertyMetadata(true));
    }

    // Prepare to collect stylus packets. If Mode is set to Select,
    // get the IncrementalHitTester from the InkPresenter's newStroke
    // StrokeCollection and subscribe to its StrokeHitChanged event.
    protected override void OnStylusDown(StylusDownEventArgs e)
    {

```

```

base.OnStylusDown(e);

Stylus.Capture(this);

// Create a new StylusPointCollection using the StylusPointDescription
// from the stylus points in the StylusDownEventArgs.
stylusPoints = new StylusPointCollection();
StylusPointCollection eventPoints = e.GetStylusPoints(this, stylusPoints.Description);

stylusPoints.Add(eventPoints);

InitializeHitTester(eventPoints);

}

protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
{
    base.OnMouseLeftButtonDown(e);

    Mouse.Capture(this);

    if (e.StylusDevice != null)
    {
        return;
    }

    Point pt = eGetPosition(this);

    StylusPointCollection collectedPoints = new StylusPointCollection(new Point[] { pt });

    stylusPoints = new StylusPointCollection();

    stylusPoints.Add(collectedPoints);

    InitializeHitTester(collectedPoints);

}

private void InitializeHitTester(StylusPointCollection collectedPoints)
{
    // Deselect any selected strokes.
    foreach (Stroke selectedStroke in selectedStrokes)
    {
        selectedStroke.DrawingAttributes.Color = inkDA.Color;
    }
    selectedStrokes.Clear();

    if (mode == InkMode.Select)
    {
        // Remove the previously drawn lasso, if it exists.
        if (lassoPath != null)
        {
            presenter.Strokes.Remove(lassoPath);
            lassoPath = null;
        }

        selectionTester =
            presenter.Strokes.GetIncrementalLassoHitTester(80);
        selectionTester.SelectionChanged +=
            new LassoSelectionChangedEventHandler(selectionTester_SelectionChanged);
        selectionTester.AddPoints(collectedPoints);
    }
}

// Collect the stylus packets as the stylus moves.
protected override void OnStylusMove(StylusEventArgs e)
{
    if (stylusPoints == null)
    {

```

```

        return;
    }

    StylusPointCollection collectedPoints = e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(collectedPoints);
    AddPointsToHitTester(collectedPoints);

}

protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);

    if (e.StylusDevice != null)
    {
        return;
    }

    if (e.LeftButton == MouseButtons.Released)
    {
        return;
    }

    stylusPoints ??= new StylusPointCollection();

    Point pt = e.GetPosition(this);

    StylusPointCollection collectedPoints = new StylusPointCollection(new Point[] { pt });

    stylusPoints.Add(collectedPoints);

    AddPointsToHitTester(collectedPoints);

}

private void AddPointsToHitTester(StylusPointCollection collectedPoints)
{
    if (mode == InkMode.Select &&
        selectionTester != null &&
        selectionTester.IsValid)
    {
        // When the control is selecting strokes, add the
        // stylus packetList to selectionTester.
        selectionTester.AddPoints(collectedPoints);

    }
}

// When the user lifts the stylus, create a Stroke from the
// collected stylus points and add it to the InkPresenter.
// When the control is selecting strokes, add the
// point data to the IncrementalHitTester.
protected override void OnStylusUp(StylusEventArgs e)
{
    stylusPoints ??= new StylusPointCollection();
    StylusPointCollection collectedPoints =
        e.GetStylusPoints(this, stylusPoints.Description);

    stylusPoints.Add(collectedPoints);
    AddPointsToHitTester(collectedPoints);
    AddStrokeToPresenter();
    stylusPoints = null;

    Stylus.Capture(null);
}

protected override void OnMouseLeftButtonUp(MouseEventArgs e)

```

```

{
    base.OnMouseLeftButtonUp(e);

    if (e.StylusDevice != null) return;

    if (stylusPoints == null) stylusPoints = new StylusPointCollection();

    Point pt = e.GetPosition(this);

    StylusPointCollection collectedPoints = new StylusPointCollection(new Point[] { pt });

    stylusPoints.Add(collectedPoints);
    AddPointsToHitTester(collectedPoints);
    AddStrokeToPresenter();

    stylusPoints = null;

    Mouse.Capture(null);

}

private void AddStrokeToPresenter()
{
    Stroke newStroke = new Stroke(stylusPoints);

    if (mode == InkMode.Ink)
    {
        // Add the stroke to the InkPresenter.
        newStroke.DrawingAttributes = inkDA.Clone();
        presenter.Strokes.Add(newStroke);
    }

    if (mode == InkMode.Select && lassoPath == null)
    {
        // Add the lasso to the InkPresenter and add the packetList
        // to selectionTester.
        lassoPath = newStroke;
        lassoPath.DrawingAttributes = selectDA.Clone();
        presenter.Strokes.Add(lassoPath);
        selectionTester.SelectionChanged -= new LassoSelectionChangedEventHandler
            (selectionTester_SelectionChanged);
        selectionTester.EndHitTesting();
    }
}

void selectionTester_SelectionChanged(object sender,
    LassoSelectionChangedEventArgs args)
{
    // Change the color of all selected strokes to red.
    foreach (Stroke selectedStroke in args.SelectedStrokes)
    {
        selectedStroke.DrawingAttributes.Color = Colors.Red;
        selectedStrokes.Add(selectedStroke);
    }

    // Change the color of all unselected strokes to
    // their original color.
    foreach (Stroke unselectedStroke in args.DeselectedStrokes)
    {
        unselectedStroke.DrawingAttributes.Color = inkDA.Color;
        selectedStrokes.Remove(unselectedStroke);
    }
}

// Property to indicate whether the user is inputting or
// selecting ink.
public InkMode Mode

```

```

    {
        get
        {
            return mode;
        }

        set
        {
            mode = value;

            // Set the DrawingAttributes of the DynamicRenderer
            if (mode == InkMode.Ink)
            {
                renderer.DrawingAttributes = inkDA;
            }
            else
            {
                renderer.DrawingAttributes = selectDA;
            }

            // Reattach the visual of the DynamicRenderer to the InkPresenter.
            presenter.DetachVisuals(renderer.RootVisual);
            presenter.AttachVisuals(renderer.RootVisual, renderer.DrawingAttributes);
        }
    }

    void DrawingAttributesChanged(object sender, PropertyDataChangedEventArgs e)
    {
        // Reattach the visual of the DynamicRenderer to the InkPresenter
        // whenever the DrawingAttributes change.
        presenter.DetachVisuals(renderer.RootVisual);
        presenter.AttachVisuals(renderer.RootVisual, renderer.DrawingAttributes);
    }

    // Property to allow the user to change the pen's DrawingAttributes.
    public DrawingAttributes InkDrawingAttributes
    {
        get
        {
            return inkDA;
        }
    }

    // Property to allow the user to change the Selector's newStroke DrawingAttributes.
    public DrawingAttributes SelectDrawingAttributes
    {
        get
        {
            return selectDA;
        }
    }
}

```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Media
Imports System.Windows.Input
Imports System.Windows.Input.StylusPlugIns
Imports System.Windows.Ink

' Enum that keeps track of whether StrokeCollectionDemo is in ink mode
' or select mode.
Public Enum InkMode
    Ink
    [Select]

```

```

End Enum 'InkMode

' This control allows the user to input and select ink. When the
' user selects ink, the lasso remains visible until they erase, or clip
' the selected strokes, or clear the selection. When the control is
' in selection mode, strokes that are selected turn red.
Public Class InkSelector
    Inherits Label

    Private modeState As InkMode

    Private inkDA As DrawingAttributes
    Private selectDA As DrawingAttributes

    Private presenter As InkPresenter
    Private selectionTester As IncrementalLassoHitTester
    Private selectedStrokes As New StrokeCollection()

    ' StylusPointCollection that collects the stylus points from the stylus events.
    Private stylusPoints As StylusPointCollection

    ' Stroke that represents the lasso.
    Private lassoPath As Stroke

    Private renderer As DynamicRenderer

    Public Sub New()
        modeState = InkMode.Ink

        ' Use an InkPresenter to display the strokes on the custom control.
        presenter = New InkPresenter()
        Me.Content = presenter

        ' In the constructor.
        ' Selection drawing attributes use dark gray ink.
        selectDA = New DrawingAttributes()
        selectDA.Color = Colors.DarkGray

        ' ink drawing attributes use default attributes
        inkDA = New DrawingAttributes()
        inkDA.Width = 5
        inkDA.Height = 5

        AddHandler inkDA.AttributeChanged, _
                    AddressOf DrawingAttributesChanged

        AddHandler selectDA.AttributeChanged, _
                    AddressOf DrawingAttributesChanged

        ' Add a DynmaicRenderer to the control so ink appears
        ' to "flow" from the tablet pen.
        renderer = New DynamicRenderer()
        renderer.DrawingAttributes = inkDA
        Me.StylusPlugIns.Add(renderer)
        presenter.AttachVisuals(renderer.RootVisual, _
                              renderer.DrawingAttributes)

    End Sub

    Shared Sub New()
        ' Allow ink to be drawn only within the bounds of the control.
        Dim owner As Type = GetType(InkSelector)
        ClipToBoundsProperty.OverrideMetadata(owner, _
                                              New FrameworkPropertyMetadata(True))

    End Sub

```

```

' Prepare to collect stylus packets. If Mode is set to Select,
' get the IncrementalHitTester from the InkPresenter'newStroke
' StrokeCollection and subscribe to its StrokeHitChanged event.
Protected Overrides Sub OnStylusDown(ByVal e As StylusDownEventArgs)
    MyBase.OnStylusDown(e)

    Stylus.Capture(Me)

    ' Create a new StylusPointCollection using the StylusPointDescription
    ' from the stylus points in the StylusDownEventArgs.
    stylusPoints = New StylusPointCollection()
    Dim eventPoints As StylusPointCollection = e.GetStylusPoints(Me, stylusPoints.Description)

    stylusPoints.Add(eventPoints)

    InitializeHitTester(eventPoints)

End Sub

Protected Overrides Sub OnMouseLeftButtonDown(ByVal e As MouseButtonEventArgs)
    MyBase.OnMouseLeftButtonDown(e)

    Mouse.Capture(Me)

    If Not (e.StylusDevice Is Nothing) Then
        Return
    End If

    Dim pt As Point = eGetPosition(Me)

    Dim collectedPoints As New StylusPointCollection(New Point() {pt})

    stylusPoints = New StylusPointCollection()

    stylusPoints.Add(collectedPoints)

    InitializeHitTester(collectedPoints)

End Sub

Private Sub InitializeHitTester(ByVal collectedPoints As StylusPointCollection)

    ' Deselect any selected strokes.
    Dim selectedStroke As Stroke
    For Each selectedStroke In selectedStrokes
        selectedStroke.DrawingAttributes.Color = inkDA.Color
    Next selectedStroke
    selectedStrokes.Clear()

    If modeState = InkMode.Select Then
        ' Remove the previously drawn lasso, if it exists.
        If Not (lassoPath Is Nothing) Then
            presenter.Strokes.Remove(lassoPath)
            lassoPath = Nothing
        End If

        selectionTester = presenter.Strokes.GetIncrementalLassoHitTester(80)
        AddHandler selectionTester.SelectionChanged, AddressOf selectionTester_SelectionChanged
        selectionTester.AddPoints(collectedPoints)
    End If

End Sub

' Collect the stylus packets as the stylus moves.
Protected Overrides Sub OnStylusMove(ByVal e As StylusEventArgs)

    If stylusPoints Is Nothing Then

```

```

    If stylusPoints Is Nothing Then
        Return
    End If

    Dim collectedPoints As StylusPointCollection = e.GetStylusPoints(Me, stylusPoints.Description)
    stylusPoints.Add(collectedPoints)
    AddPointsToHitTester(collectedPoints)

End Sub

Protected Overrides Sub OnMouseMove(ByVal e As MouseEventArgs)

    MyBase.OnMouseMove(e)

    If Not (e.StylusDevice Is Nothing) Then
        Return
    End If

    If e.LeftButton = MouseButtons.Released Then
        Return
    End If

    If stylusPoints Is Nothing Then
        stylusPoints = New StylusPointCollection()
    End If

    Dim pt As Point = e.GetPosition(Me)

    Dim collectedPoints As New StylusPointCollection(New Point() {pt})
    stylusPoints.Add(collectedPoints)

    AddPointsToHitTester(collectedPoints)

End Sub

Private Sub AddPointsToHitTester(ByVal collectedPoints As StylusPointCollection)

    If modeState = InkMode.Select AndAlso _
        Not selectionTester Is Nothing AndAlso _
        selectionTester.IsValid Then

        ' When the control is selecting strokes, add the
        ' stylus packetList to selectionTester.
        selectionTester.AddPoints(collectedPoints)
    End If

End Sub

' When the user lifts the stylus, create a Stroke from the
' collected stylus points and add it to the InkPresenter.
' When the control is selecting strokes, add the
' point data to the IncrementalHitTester.

Protected Overrides Sub OnStylusUp(ByVal e As StylusEventArgs)

    If stylusPoints Is Nothing Then
        stylusPoints = New StylusPointCollection()
    End If

    Dim collectedPoints As StylusPointCollection = _
        e.GetStylusPoints(Me, stylusPoints.Description)

    stylusPoints.Add(collectedPoints)
    AddPointsToHitTester(collectedPoints)
    AddStrokeToPresenter()

    stylusPoints = Nothing

```

```

    Stylus.Capture(Nothing)

End Sub

Protected Overrides Sub OnMouseLeftButtonUp(ByVal e As MouseButtonEventArgs)
    MyBase.OnMouseLeftButtonUp(e)

    If Not (e.StylusDevice Is Nothing) Then
        Return
    End If
    If stylusPoints Is Nothing Then
        stylusPoints = New StylusPointCollection()
    End If
    Dim pt As Point = e.GetPosition(Me)

    Dim collectedPoints As New StylusPointCollection(New Point() {pt})

    stylusPoints.Add(collectedPoints)
    AddPointsToHitTester(collectedPoints)
    AddStrokeToPresenter()

    stylusPoints = Nothing

    Mouse.Capture(Nothing)

End Sub

Private Sub AddStrokeToPresenter()
    Dim newStroke As New Stroke(stylusPoints)

    If modeState = InkMode.Ink Then
        ' Add the stroke to the InkPresenter.
        newStroke.DrawingAttributes = inkDA.Clone()
        presenter.Strokes.Add(newStroke)
    End If

    If modeState = InkMode.Select AndAlso lassoPath Is Nothing Then
        ' Add the lasso to the InkPresenter and add the packetList
        ' to selectionTester.
        lassoPath = newStroke
        lassoPath.DrawingAttributes = selectDA.Clone()
        presenter.Strokes.Add(lassoPath)
        RemoveHandler selectionTester.SelectionChanged, _
            AddressOf selectionTester_SelectionChanged
        selectionTester.EndHitTesting()
    End If
End Sub

Private Sub selectionTester_SelectionChanged(ByVal sender As Object, _
    ByVal args As LassoSelectionChangedEventArgs)

    ' Change the color of all selected strokes to red.
    Dim selectedStroke As Stroke
    For Each selectedStroke In args.SelectedStrokes
        selectedStroke.DrawingAttributes.Color = Colors.Red
        selectedStrokes.Add(selectedStroke)
    Next selectedStroke

    ' Change the color of all unselected strokes to
    ' their original color.
    Dim unselectedStroke As Stroke
    For Each unselectedStroke In args.DeselectedStrokes
        unselectedStroke.DrawingAttributes.Color = inkDA.Color
        selectedStrokes.Remove(unselectedStroke)
    Next unselectedStroke
End Sub

```

```

    Next unselectedStroke

End Sub

' Property to indicate whether the user is inputting or
' selecting ink.
Public Property Mode() As InkMode
    Get
        Return Mode
    End Get

    Set(ByVal value As InkMode)
        modeState = value

        ' Set the DrawingAttributes of the DynamicRenderer
        If modeState = InkMode.Ink Then
            renderer.DrawingAttributes = inkDA
        Else
            renderer.DrawingAttributes = selectDA
        End If

        ' Reattach the visual of the DynamicRenderer to the InkPresenter.
        presenter.DetachVisuals(renderer.RootVisual)
        presenter.AttachVisuals(renderer.RootVisual, renderer.DrawingAttributes)
    End Set
End Property
Private Sub DrawingAttributesChanged(ByVal sender As Object, _
                                     ByVal e As PropertyDataChangedEventArgs)

    ' Reattach the visual of the DynamicRenderer to the InkPresenter
    ' whenever the DrawingAttributes change.
    presenter.DetachVisuals(renderer.RootVisual)
    presenter.AttachVisuals(renderer.RootVisual, _
                           renderer.DrawingAttributes)

End Sub

' Property to allow the user to change the pen's DrawingAttributes.
Public ReadOnly Property InkDrawingAttributes() As DrawingAttributes
    Get
        Return inkDA
    End Get
End Property

' Property to allow the user to change the Selector's DrawingAttributes.
Public ReadOnly Property SelectDrawingAttributes() As DrawingAttributes
    Get
        Return selectDA
    End Get
End Property

End Class

```

## See also

- [IncrementalLassoHitTester](#)
- [StrokeCollection](#)
- [StylusPointCollection](#)
- [Creating an Ink Input Control](#)

# How to: Add Custom Data to Ink Data

4/28/2019 • 2 minutes to read • [Edit Online](#)

You can add custom data to ink that will be saved when the ink is saved as ink serialized format (ISF). You can save the custom data to the [DrawingAttributes](#), the [StrokeCollection](#), or the [Stroke](#). Being able to save custom data on three objects gives you the ability to decide the best place to save the data. All three classes use similar methods to store and access custom data.

Only the following types can be saved as custom data:

- [Boolean](#)
- [Boolean\[\]](#)
- [Byte](#)
- [Byte\[\]](#)
- [Char](#)
- [Char\[\]](#)
- [DateTime](#)
- [DateTime\[\]](#)
- [Decimal](#)
- [Decimal\[\]](#)
- [Double](#)
- [Double\[\]](#)
- [Int16](#)
- [Int16\[\]](#)
- [Int32](#)
- [Int32\[\]](#)
- [Int64](#)
- [Int64\[\]](#)
- [Single](#)
- [Single\[\]](#)
- [String](#)
- [UInt16](#)
- [UInt16\[\]](#)
- [UInt32](#)
- [UInt32\[\]](#)

- [UInt64](#)
- [UInt64\[\]](#)

## Example

The following example demonstrates how to add and retrieve custom data from a [StrokeCollection](#).

```
Guid timestamp = new Guid("12345678-9012-3456-7890-123456789012");

// Add a timestamp to the StrokeCollection.
private void AddTimestamp()
{
    inkCanvas1.Strokes.AddPropertyData(timestamp, DateTime.Now);
}

// Get the timestamp of the StrokeCollection.
private void GetTimestamp()
{
    if (inkCanvas1.Strokes.ContainsPropertyData(timestamp))
    {
        object date = inkCanvas1.Strokes.GetPropertyData(timestamp);

        if (date is DateTime)
        {
            MessageBox.Show("This StrokeCollection's timestamp is " +
                ((DateTime)date).ToString());
        }
    }
    else
    {
        MessageBox.Show(
            "The StrokeCollection does not have a timestamp.");
    }
}
```

The following example creates an application that displays an [InkCanvas](#) and two buttons. The button, `switchAuthor`, enables two pens to be used by two different authors. The button `changePenColors` changes the color of each stroke on the [InkCanvas](#) according to the author. The application defines two [DrawingAttributes](#) objects and adds a custom property to each one that indicates which author drew the [Stroke](#). When the user clicks `changePenColors`, the application changes the appearance of the stroke according to the value of the custom property.

```

<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Adding Custom Data to Ink" Height="500" Width="700"
    >
    <DockPanel Name="root">

        <StackPanel Background="DarkSlateBlue">
            <Button Name="switchAuthor" Click="switchAuthor_click" >
                Switch to student's pen
            </Button>
            <Button Name="changePenColors" Click="changeColor_click" >
                Change the color of the pen ink
            </Button>
        </StackPanel>
        <InkCanvas Name="inkCanvas1">
        </InkCanvas>
    </DockPanel>
</Window>

```

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Ink;

/// <summary>
/// Interaction logic for Window1.xaml
/// </summary>

public partial class Window1 : Window
{
    Guid authorGuid = new Guid("12345678-9012-3456-7890-123456789012");
    DrawingAttributes teachersDA = new DrawingAttributes();
    DrawingAttributes studentsDA = new DrawingAttributes();
    string teacher = "teacher";
    string student = "student";
    bool useStudentPen = false;

    public Window1()
    {
        InitializeComponent();

        teachersDA.Color = Colors.Red;
        teachersDA.Width = 5;
        teachersDA.Height = 5;
        teachersDA.AddPropertyData(authorGuid, teacher);

        studentsDA.Color = Colors.Blue;
        studentsDA.Width = 5;
        studentsDA.Height = 5;
        studentsDA.AddPropertyData(authorGuid, student);

        inkCanvas1.DefaultDrawingAttributes = teachersDA;
    }

    // Switch between using the 'pen' DrawingAttributes and the
    // 'highlighter' DrawingAttributes.
    void switchAuthor_click(Object sender, RoutedEventArgs e)
    {
        useStudentPen = !useStudentPen;
    }
}

```

```
if (useStudentPen)
{
    switchAuthor.Content = "Use teacher's pen";
    inkCanvas1.DefaultDrawingAttributes = studentsDA;
}
else
{
    switchAuthor.Content = "Use student's pen";
    inkCanvas1.DefaultDrawingAttributes = teachersDA;

}

}

// Change the color of the ink that on the InkCanvas that used the pen.
void changeColor_click(Object sender, RoutedEventArgs e)
{
    foreach (Stroke s in inkCanvas1.Strokes)
    {
        if (s.DrawingAttributes.ContainsPropertyData(authorGuid))
        {
            object data = s.DrawingAttributes.GetPropertyData(authorGuid);

            if ((data is string) && ((string)data == teacher))
            {
                s.DrawingAttributes.Color = Colors.Black;
            }
            if ((data is string) && ((string)data == student))
            {
                s.DrawingAttributes.Color = Colors.Green;
            }
        }
    }
}
```

# How to: Erase Ink on a Custom Control

3/5/2019 • 4 minutes to read • [Edit Online](#)

The [IncrementalStrokeHitTester](#) determines whether the currently drawn stroke intersects another stroke. This is useful for creating a control that enables a user to erase parts of a stroke, the way a user can on an [InkCanvas](#) when the [EditingMode](#) is set to [EraseByPoint](#).

## Example

The following example creates a custom control that enables the user to erase parts of strokes. This example creates a control that contains ink when it is initialized. To create a control that collects ink, see [Creating an Ink Input Control](#).

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.IO;

// This control initializes with ink already on it and allows the
// user to erase the ink with the tablet pen or mouse.
public class InkEraser : Label
{
    IncrementalStrokeHitTester eraseTester;

    InkPresenter presenter;

    string strokesString =
        @"ALwHAxIEETLgIgERYQBGwIAJAFhAEbAgAkAQUBOBkgMgkA9P8CAekiOkUzCQD4n"
        + "wIBWiA6RTgIAP4DAAAAGh8RAACAPx8JEQAAAAAAAPA/CiUhh/A6N4HR0AivFX8Vs"
        + "IfsiuyLSaIeDSLwabiHm0GgUDi+KZkAcjsQh/A9t4IC5VJpfLhaIxxyXih7Dncnh"
        + "+6e7qODwoER1PAw8EpGoJAg61IKjCYXBVD4DAIHf67KIHAojoB4fwMteBn+RKB"
        + "lziaIfwWTeCwePqbM8WgIeeQCDQOFRvcIAKNA+H8B8XgQhkUbjQTTnGuaZns3l4h"
        + "/DWt4a0YKBB0A94D6HRCAlGnp5CS8LEXMLB1t0gYIAKUB0H8KnXhU71Mold+tcbi"
        + "kChkqu2EtPYxp9bmYCH8HDhg4ZhMiWryMHH+4Jt8nleX8c0/D/AkYwxJGiHkkQgm"
        + "Ch9CqcFhMDQCBwlAwuR2eAACmgdh/EpF41A6XMUFhMXgMHgVdxBFpRkpZII5EINA"
        + "OA64M+J4Lw1CIOAh/B2x4PS4bQodAopEI5IJBki4waEx2Qy+dy+ayHgleEmmHH8c"
        + "e3MZOCGw5Twd3CwsHAwMCgRAEAgElKwOHZKBApagIfxezeL0uN02N8IzwaGEpNIJ"
        + "ZxHnElYoj0GfyuU6FgmhpIlgIfwYgeDHeaI1vjt0ZgcHgHAYb9hUCgEFgsPm1xnM"
        + "ZkYhsnYJgZeZh4uAgCgnSBlOjv40AgwCmkgh/GrR41X4dGoRJL9EKra5KY7IZ3C"
        + "4fj/M06olSoU8kkehUbh8jkMdCH8IJXhAXhMCk8JuNlmNyh0YiEumUwn2wMRxyHw"
        + "2TzWmzeb020zGKxMITwIhnjzb44zRhGEKRhCM4zrr6sQKXRWH8kuXkmPj0DiXC"
        + "gcJbC9HZZgkKgUG4bLh3YrJHAYw2Cah/CiN4Tq7DOZr4BB/AFtdOWW5P2h1Wkzv"
        + "14+YwqXF8d5fZ7ih51QkbB4LqrLAYDBIDABA4B04nAIACAvIIfy4BeXA2DRSrQ1L"
        + "oHhsYQ/KMxlsv18rn8Xkcdg+G9NVaUWimUDYk9Ah/BoF4M0YBCqZPYqk8dwLf7hD"
        + "YNBJFLKBnQzTqNubWshl9VoMireFYZYQEBUGsDAwKEjYuDQKBgICBgcCAgIOAg4nI"
        + "80ACloSh/Bf14Gf/I0t6FXF8F4ToPCzz1Pwp4+B+DhmQ0847rfDeCcG8eKh/EZV"
        + "419eZt8A9nUF8VzxaUe5grl7YrPaHfpRKJNx4yHmUuj1vicwmMBEAjUVgKB61A=";

    public InkEraser()
    {
        presenter = new InkPresenter();
        this.Content = presenter;

        // Create a StrokeCollection the string and add it to
        StrokeCollectionConverter converter =
            new StrokeCollectionConverter();

        if (converter.CanConvertFrom(typeof(string)))
    }
```

```

        {
            StrokeCollection newStrokes =
                converter.ConvertFrom(strokesString) as StrokeCollection;
            presenter.Strokes.Clear();
            presenter.Strokes.Add(newStrokes);
        }

    }

protected override void OnStylusDown(StylusEventArgs e)
{
    base.OnStylusDown(e);
    StylusPointCollection points = e.GetStylusPoints(this);

    InitializeEraserHitTester(points);

}

protected override void OnMouseLeftButtonDown(MouseEventArgs e)
{
    base.OnMouseLeftButtonDown(e);

    if (e.StylusDevice != null)
    {
        return;
    }

    Point pt = e.GetPosition(this);

    StylusPointCollection collectedPoints = new StylusPointCollection(new Point[] { pt });

    InitializeEraserHitTester(collectedPoints);
}

// Prepare to collect stylus packets. Get the
// IncrementalHitTester from the InkPresenter's
// StrokeCollection and subscribe to its StrokeHitChanged event.
private void InitializeEraserHitTester(StylusPointCollection points)
{
    EllipseStylusShape eraserTip = new EllipseStylusShape(3, 3, 0);
    eraseTester =
        presenter.Strokes.GetIncrementalStrokeHitTester(eraserTip);
    eraseTester.StrokeHit += new StrokeHitEventHandler(eraseTester_StrokeHit);
    eraseTester.AddPoints(points);
}

protected override void OnStylusMove(StylusEventArgs e)
{
    StylusPointCollection points = e.GetStylusPoints(this);

    AddPointsToEraserHitTester(points);
}

protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);

    if (e.StylusDevice != null)
    {
        return;
    }

    if (e.LeftButton == MouseButtonState.Released)
    {
        return;
    }

    Point pt = e.GetPosition(this);
}

```

```

        StylusPointCollection collectedPoints = new StylusPointCollection(new Point[] { pt });

        AddPointsToEraserHitTester(collectedPoints);
    }

    // Collect the StylusPackets as the stylus moves.
    private void AddPointsToEraserHitTester(StylusPointCollection points)
    {
        if (eraseTester.IsValid)
        {
            eraseTester.AddPoints(points);
        }
    }

    // Unsubscribe from the StrokeHitChanged event when the
    // user lifts the stylus.
    protected override void OnStylusUp(StylusEventArgs e)
    {
        StylusPointCollection points = e.GetStylusPoints(this);

        StopEraseHitTesting(points);
    }

    protected override void OnMouseLeftButtonUp(MouseEventArgs e)
    {
        base.OnMouseLeftButtonUp(e);

        if (e.StylusDevice != null)
        {
            return;
        }

        Point pt = e.GetPosition(this);

        StylusPointCollection collectedPoints = new StylusPointCollection(new Point[] { pt });

        StopEraseHitTesting(collectedPoints);
    }

    private void StopEraseHitTesting(StylusPointCollection points)
    {
        eraseTester.AddPoints(points);
        eraseTester.StrokeHit -= new
            StrokeHitEventHandler(eraseTester_StrokeHit);
        eraseTester.EndHitTesting();
    }

    // When the stylus intersects a stroke, erase that part of
    // the stroke. When the stylus dissects a stroke, the
    // Stroke.Erase method returns a StrokeCollection that contains
    // the two new strokes.
    void eraseTester_StrokeHit(object sender,
        StrokeHitEventArgs args)
    {
        StrokeCollection eraseResult =
            args.GetPointEraseResults();
        StrokeCollection strokesToReplace = new StrokeCollection();
        strokesToReplace.Add(args.HitStroke);

        // Replace the old stroke with the new one.
        if (eraseResult.Count > 0)
        {
            presenter.Strokes.Replace(strokesToReplace, eraseResult);
        }
        else
        {
            presenter.Strokes.Remove(strokesToReplace);
        }
    }
}

```

```

    presenter.Strokes.Remove(strokesReplace),
}
}
}

```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Ink
Imports System.Windows.Input
Imports System.Windows.Media
Imports System.IO

' This control initializes with ink already on it and allows the
' user to erase the ink with the tablet pen or mouse.

Public Class InkEraser
    Inherits Label
    Private eraseTester As IncrementalStrokeHitTester

    Private presenter As InkPresenter

    ' The base-64 encoded string that contains ink data
    ' in ink serialized format (ISF).
    Private strokesString As String = _
        "ALwHAxdIEETLgIgERYQBGwIAJAFGhAEbAgAkAQUBOBkgMgkA9P8CAekiOkUzCQD4n" _
        & "wIBWiA6RTgIAP4DAAAAGh8RAACAPx8JEQAAAAAAAPA/CiUh/A6N4HR0AivFX8Vs" _
        & "IfsiuyLSaIeDSLwabiHm0GgUDi+KZkACjsQh/A9t4IC5VJpfLhaIxxyxXIh7Dncnh" _
        & "+6e7qODwoERlPAw8EpGoJAg61IKjCYXBYDA4DAIHf67KIHAAojB4fwMteBn+RKB" _
        & "lziaIfwWTeCwePqbM8WgIeeQCDQ0FRvcIAKNA+H8B8XgQHkUbjQTTnGuaZns314h" _
        & "/DWt4a0YKBOA94D6HRCAiGnp5CS8LEXMLB1t0gYIAKUBOH8KnXhU71Mold+tcbi" _
        & "kChkqu2EtPYxp9bmYCH8HDhg4ZhMiwRyMHH+4Jt8nleX8c0/D/AkYwxJGiHkkQgm" _
        & "Ch9CqcFhMDQCBwWAwuR2eAACmgdh/EpF41A6XMUfhMXgMHgVDxBFpRKpZII5EINA" _
        & "OA64M+J4Lw1CIoAh/B2x4PS4bQodAopEI5IJBki4waEx20y+dy+ayHgleEemmHH8c" _
        & "e3MZOCGw5Twd3CwsHAwMcgRAEagElKw0HZKBApaGIfxezeL0uN02N8IzwaGEpNIJ" _
        & "ZxHnElYoj0GfyuU6FgmhpIlgIfwYgeDHeaI1vj0tZgcHgHAYb9hUCgEFgsPm1xnM" _
        & "ZkYhsnYJgZeZh4uAgCgnSBlOjv40AgwCmkgh/GrR41X4dGoRJL9EKra5HKY7IZ3C" _
        & "4fj/M06olSoU8kkehUbh8jkMdCH8IJXhAXhMCk8JuNmNyh0YiEumUwn2wMRxyHw" _
        & "2TzWmzeb020zGKxMITwIhnrzbb44zRhGEKRhCM4zrr6sQKXRWH8kuXkmPj0DiXC" _
        & "gcJbC9HZZgkKgUG4bLh3YrwJHAYw2CAh/CiN4Tq7DOZr4BB/AFtdOWW5P2h1Wkzv" _
        & "14+YwqXF8d5fZ7ih51Qkb4LqrLAYDBIDABA4B04nAICApvIIfy4BeXA2DRSrQll" _
        & "oHhsYQ/KMX1svl8rn8Xkcdg+G9NVaUwimUDYk9Ah/Bof4M0YBCqZPYqk8dwLf7hd" _
        & "YNBJFLKBnqZTqNubWsh19VoM1reFYZYQEBGUsDAwKEjYuDQKBgICBgcAgIOAg4nI" _
        & "80ACloSh/BF14Gf/I0t6FXfF8F4ToPCzz1PwP4+B+DhmQ0847rfDeCcG8eKh/EZV" _
        & "4i9eZt8A9nUF8VzaUe5grl7YrPaHfpRKJNx4yHmUuj1vicwmMBEAjUVgKB61A="

Public Sub New()
    presenter = New InkPresenter()
    Me.Content = presenter

    ' Create a StrokeCollection the string and add it to
    Dim converter As New StrokeCollectionConverter()

    If converter.CanConvertFrom(GetType(String)) Then
        Dim newStrokes As StrokeCollection = converter.ConvertFrom(strokesString)

        presenter.Strokes.Clear()
        presenter.Strokes.Add(newStrokes)
    End If

End Sub

Protected Overrides Sub OnStylusDown(ByVal e As StylusDownEventArgs)

    MyBase.OnStylusDown(e)
    Dim points As StylusPointCollection = e.GetStylusPoints(Me)

    InitializeEraserHitTester(points)

```

```

End Sub

Protected Overrides Sub OnMouseLeftButtonDown(ByVal e As MouseButtonEventArgs)
    MyBase.OnMouseLeftButtonDown(e)

    If Not (e.StylusDevice Is Nothing) Then
        Return
    End If

    Dim pt As Point = e.GetPosition(Me)

    Dim collectedPoints As New StylusPointCollection(New Point() {pt})

    InitializeEraserHitTester(collectedPoints)

End Sub

' Get the IncrementalHitTester from the InkPresenter's
' StrokeCollection and subscribe to its StrokeHitChanged event.
Private Sub InitializeEraserHitTester(ByVal points As StylusPointCollection)

    Dim eraserTip As New EllipseStylusShape(3, 3, 0)
    eraseTester = presenter.Strokes.GetIncrementalStrokeHitTester(eraserTip)
    AddHandler eraseTester.StrokeHit, AddressOf eraseTester_StrokeHit
    eraseTester.AddPoints(points)

End Sub

Protected Overrides Sub OnStylusMove(ByVal e As StylusEventArgs)
    Dim points As StylusPointCollection = e.GetStylusPoints(Me)

    AddPointsToEraserHitTester(points)

End Sub

Protected Overrides Sub OnMouseMove(ByVal e As MouseEventArgs)
    MyBase.OnMouseMove(e)

    If Not (e.StylusDevice Is Nothing) Then
        Return
    End If

    If e.LeftButton = MouseButtons.Released Then
        Return
    End If

    Dim pt As Point = e.GetPosition(Me)

    Dim collectedPoints As New StylusPointCollection(New Point() {pt})

    AddPointsToEraserHitTester(collectedPoints)

End Sub

' Collect the StylusPackets as the stylus moves.
Private Sub AddPointsToEraserHitTester(ByVal points As StylusPointCollection)

    If eraseTester.IsValid Then
        eraseTester.AddPoints(points)
    End If

End Sub

```

\* Unsubscribe from the StrokeHitChanged event when the

```

    ' OnStylusUp from the StylusEventChanged event when the
    ' user lifts the stylus.
Protected Overrides Sub OnStylusUp(ByVal e As StylusEventArgs)

    Dim points As StylusPointCollection = e.GetStylusPoints(Me)

    StopEraseHitTesting(points)

End Sub

Protected Overrides Sub OnMouseLeftButtonUp(ByVal e As MouseButtonEventArgs)

    MyBase.OnMouseLeftButtonUp(e)

    If Not (e.StylusDevice Is Nothing) Then
        Return
    End If

    Dim pt As Point = eGetPosition(Me)

    Dim collectedPoints As New StylusPointCollection(New Point() {pt})

    StopEraseHitTesting(collectedPoints)

End Sub

Private Sub StopEraseHitTesting(ByVal points As StylusPointCollection)

    eraseTester.AddPoints(points)
    RemoveHandler eraseTester.StrokeHit, AddressOf eraseTester_StrokeHit
    eraseTester.EndHitTesting()

End Sub

' When the stylus intersects a stroke, erase that part of
' the stroke. When the stylus dissects a stroke, the
' Stroke.Erase method returns a StrokeCollection that contains
' the two new strokes.
Private Sub eraseTester_StrokeHit(ByVal sender As Object, ByVal args As StrokeHitEventArgs)

    Dim eraseResult As StrokeCollection = args.GetPointEraseResults()
    Dim strokesToReplace As New StrokeCollection()
    strokesToReplace.Add(args.HitStroke)

    ' Replace the old stroke with the new one.
    If eraseResult.Count > 0 Then
        presenter.Strokes.Replace(strokesToReplace, eraseResult)
    Else
        presenter.Strokes.Remove(strokesToReplace)
    End If

End Sub
End Class

```

# How To: Recognize Application Gestures

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following example demonstrates how to erase ink when a user makes a [ScratchOut](#) gesture on an [InkCanvas](#). This example assumes an [InkCanvas](#), called `inkCanvas1`, is declared in the XAML file.

## Example

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Ink;
using System.Collections.ObjectModel;

public partial class Window1 : Window
{
    public Window1()
    {
        InitializeComponent();

        if (inkCanvas1.IsGestureRecognizerAvailable)
        {
            inkCanvas1.EditingMode = InkCanvasEditingMode.InkAndGesture;
            inkCanvas1.Gesture += new InkCanvasGestureEventHandler(inkCanvas1_Gesture);
            inkCanvas1.SetEnabledGestures(
                new ApplicationGesture[] { ApplicationGesture.ScratchOut });
        }
    }

    void inkCanvas1_Gesture(object sender, InkCanvasGestureEventArgs e)
    {
        ReadOnlyCollection<GestureRecognitionResult> gestureResults =
            e.GetGestureRecognitionResults();

        // Check the first recognition result for a gesture.
        if ((gestureResults[0].RecognitionConfidence ==
            RecognitionConfidence.Strong) &&
            (gestureResults[0].ApplicationGesture ==
            ApplicationGesture.ScratchOut))
        {
            StrokeCollection hitStrokes = inkCanvas1.Strokes.HitTest(
                e.Strokes.GetBounds(), 10);

            if (hitStrokes.Count > 0)
            {
                inkCanvas1.Strokes.Remove(hitStrokes);
            }
        }
    }
}
```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Ink
Imports System.Collections.ObjectModel

Class Window1
    Inherits Window

    Public Sub New()
        InitializeComponent()

        If inkCanvas1.IsGestureRecognizerAvailable Then
            inkCanvas1.EditingMode = InkCanvasEditingMode.InkAndGesture
            AddHandler inkCanvas1.Gesture, AddressOf inkCanvas1_Gesture
            inkCanvas1.SetEnabledGestures(New ApplicationGesture() {ApplicationGesture.ScratchOut})
        End If

    End Sub

    Private Sub inkCanvas1_Gesture(ByVal sender As Object, ByVal e As InkCanvasGestureEventArgs)

        Dim gestureResults As ReadOnlyCollection(Of GestureRecognitionResult) = _
            e.GetGestureRecognitionResults()

        ' Check the first recognition result for a gesture.
        If gestureResults(0).RecognitionConfidence = _
            RecognitionConfidence.Strong AndAlso _
            gestureResults(0).ApplicationGesture = _
            ApplicationGesture.ScratchOut Then

            Dim hitStrokes As StrokeCollection = _
                inkCanvas1.Strokes.HitTest(e.Strokes.GetBounds(), 10)

            If hitStrokes.Count > 0 Then
                inkCanvas1.Strokes.Remove(hitStrokes)
            End If
        End If

    End Sub
End Class

```

## See also

- [ApplicationGesture](#)
- [InkCanvas](#)
- [Gesture](#)

# How to: Drag and Drop Ink

3/5/2019 • 3 minutes to read • [Edit Online](#)

## Example

The following example creates an application that enables the user to drag selected strokes from one [InkCanvas](#) to the other.

```
<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="InkDragDropSample" Height="500" Width="700"
    >
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <InkCanvas Name="ic1" AllowDrop="True"
        Grid.Column="0" Grid.Row="0"
        Margin="10,10,10,10" Background="AliceBlue"
        PreviewMouseDown="InkCanvas_PreviewMouseDown"
        Drop="InkCanvas_Drop"/>

    <InkCanvas Name="ic2" AllowDrop="True"
        Grid.Column="1" Grid.Row="0"
        Margin="10,10,10,10" Background="Beige"
        PreviewMouseDown="InkCanvas_PreviewMouseDown"
        Drop="InkCanvas_Drop"/>

    <CheckBox Grid.Row="1"
        Checked="switchToSelect" Unchecked="switchToInk">
        Select Mode
    </CheckBox>
</Grid>
</Window>
```

```
using System;
using System.IO;
using System.Windows;
using System.Windows.Ink;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Input;
using System.Windows.Media;

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

    void InkCanvas_PreviewMouseDown(object sender, MouseEventArgs e)
```

```

    {
        InkCanvas ic = (InkCanvas)sender;

        Point pt = e.GetPosition(ic);

        // If the user is moving selected strokes, prepare the strokes to be
        // moved to another InkCanvas.
        if (ic.HitTestSelection(pt) ==
            InkCanvasSelectionHitResult.Selection)
        {
            StrokeCollection selectedStrokes = ic.GetSelectedStrokes();
            StrokeCollection strokesToMove = selectedStrokes.Clone();

            // Remove the offset of the selected strokes so they
            // are positioned when the strokes are dropped.
            Rect inkBounds = strokesToMove.GetBounds();
            TranslateStrokes(strokesToMove, -inkBounds.X, -inkBounds.Y);

            // Perform drag and drop.
            MemoryStream ms = new MemoryStream();
            strokesToMove.Save(ms);
            DataObject dataObject = new DataObject(
                StrokeCollection.InkSerializedFormat, ms);

            DragDropEffects effects =
                DragDrop.DoDragDrop(ic, dataObject,
                    DragDropEffects.Move);

            if ((effects & DragDropEffects.Move) ==
                DragDropEffects.Move)
            {
                // Remove the selected strokes
                // from the current InkCanvas.
                ic.Strokes.Remove(selectedStrokes);
            }
        }
    }

    void InkCanvas_Drop(object sender, DragEventArgs e)
    {
        // Get the strokes that were moved.
        InkCanvas ic = (InkCanvas)sender;
        MemoryStream ms = (MemoryStream)e.Data.GetData(
            StrokeCollection.InkSerializedFormat);
        ms.Position = 0;
        StrokeCollection strokes = new StrokeCollection(ms);

        // Translate the strokes to the position at which
        // they were dropped.
        Point pt = e.GetPosition(ic);
        TranslateStrokes(strokes, pt.X, pt.Y);

        // Add the strokes to the InkCanvas and keep them selected.
        ic.Strokes.Add(strokes);
        ic.Select(strokes);
    }

    // Helper method that translates the specified strokes.
    void TranslateStrokes(StrokeCollection strokes,
        double x, double y)
    {
        Matrix mat = new Matrix();
        mat.Translate(x, y);
        strokes.Transform(mat, false);
    }

    void switchToSelect(object sender, RoutedEventArgs e)
    {
        ic1.EditingMode = InkCanvasEditingMode.Select;
    }
}

```

```

        ic1.EditingMode = InkCanvasEditingStyle.Select,
        ic2.EditingMode = InkCanvasEditingStyle.Select;
    }

    void switchToInk(object sender, RoutedEventArgs e)
    {
        ic1.EditingMode = InkCanvasEditingStyle.Ink;
        ic2.EditingMode = InkCanvasEditingStyle.Ink;
    }
}

```

```

Imports System.IO
Imports System.Windows
Imports System.Windows.Ink
Imports System.Windows.Controls
Imports System.Windows.Data
Imports System.Windows.Input
Imports System.Windows.Media

Class Window1
    Inherits Window

    Public Sub New()
        InitializeComponent()
    End Sub

    Private Sub InkCanvas_PreviewMouseDown(ByVal sender As Object, _
                                         ByVal e As MouseButtonEventArgs)

        Dim ic As InkCanvas = CType(sender, InkCanvas)

        Dim pt As Point = e.GetPosition(ic)

        ' If the user is moving selected strokes, prepare the strokes to be
        ' moved to another InkCanvas.
        If ic.HitTestSelection(pt) = InkCanvasSelectionHitResult.Selection Then

            Dim selectedStrokes As StrokeCollection = _
                ic.GetSelectedStrokes()

            Dim strokesToMove As StrokeCollection = _
                selectedStrokes.Clone()

            ' Remove the offset of the selected strokes so they
            ' are positioned when the strokes are dropped.
            Dim inkBounds As Rect = strokesToMove.GetBounds()
            TranslateStrokes(strokesToMove, -inkBounds.X, -inkBounds.Y)

            ' Perform drag and drop.
            Dim ms As New MemoryStream()
            strokesToMove.Save(ms)

            Dim dataObject As New DataObject _
                (StrokeCollection.InkSerializedFormat, ms)

            Dim effects As DragDropEffects = _
                DragDrop.DoDragDrop(ic, dataObject, DragDropEffects.Move)

            If (effects And DragDropEffects.Move) = DragDropEffects.Move Then

                ' Remove the selected strokes from the current InkCanvas.
                ic.Strokes.Remove(selectedStrokes)
            End If
        End If
    End Sub

```

```

Private Sub InkCanvas_Drop(ByVal sender As Object, _
    ByVal e As DragEventArgs)

    ' Get the strokes that were moved.
    Dim ic As InkCanvas = CType(sender, InkCanvas)
    Dim ms As MemoryStream = CType(e.Data.GetData( _
        StrokeCollection.InkSerializedFormat), _
        MemoryStream)

    ms.Position = 0
    Dim strokes As New StrokeCollection(ms)

    ' Translate the strokes to the position at which
    ' they were dropped.
    Dim pt As Point = e.GetPosition(ic)
    TranslateStrokes(strokes, pt.X, pt.Y)

    ' Add the strokes to the InkCanvas and keep them selected.
    ic.Strokes.Add(strokes)
    ic.Select(strokes)

End Sub

' Helper method that translates the specified strokes.
Sub TranslateStrokes(ByVal strokes As StrokeCollection, _
    ByVal x As Double, ByVal y As Double)

    Dim mat As New Matrix()
    mat.Translate(x, y)
    strokes.Transform(mat, False)

End Sub

Private Sub switchToSelect(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)

    ic1.EditingMode = InkCanvasEditingStyle.Select
    ic2.EditingMode = InkCanvasEditingStyle.Select

End Sub

Private Sub switchToInk(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)

    ic1.EditingMode = InkCanvasEditingStyle.Ink
    ic2.EditingMode = InkCanvasEditingStyle.Ink

End Sub

End Class

```

# How to: Data Bind to an InkCanvas

3/5/2019 • 2 minutes to read • [Edit Online](#)

## Example

The following example demonstrates how to bind the [Strokes](#) property of an [InkCanvas](#) to another [InkCanvas](#).

```
<InkCanvas Background="LightGray"
    Canvas.Top="0" Canvas.Left="0"
    Height="400" Width="200" Name="ic"/>

<!-- Bind the Strokes of the second InkCanvas to the first InkCanvas
    and mirror the strokes along the Y axis.-->
<InkCanvas Background="LightBlue"
    Canvas.Top="0" Canvas.Left="200"
    Height="400" Width="200"
    Strokes="{Binding ElementName=ic, Path=Strokes}">
    <InkCanvas.LayoutTransform>
        <ScaleTransform ScaleX="-1" ScaleY="1" />
    </InkCanvas.LayoutTransform>
</InkCanvas>
```

The following example demonstrates how to bind the [DefaultDrawingAttributes](#) property to a data source.

```
<Canvas.Resources>
    <!--Define an array containing some DrawingAttributes.-->
    <x:Array x:Key="MyDrawingAttributes" x:Type="{x:Type DrawingAttributes}">
        <DrawingAttributes Color="Black" FitToCurve="true" Width="3" Height="3"/>
        <DrawingAttributes Color="Blue" FitToCurve="false" Width="5" Height="5"/>
        <DrawingAttributes Color="Red" FitToCurve="true" Width="7" Height="7"/>
    </x:Array>

    <!--Create a DataTemplate to display the DrawingAttributes shown above-->
    <DataTemplate DataType="{x:Type DrawingAttributes}" >
        <Border Width="80" Height="{Binding Path=Height}">
            <Border.Background>
                <SolidColorBrush Color="{Binding Path=Color}" />
            </Border.Background>
        </Border>
    </DataTemplate>
</Canvas.Resources>
```

```
<!--Bind the InkCanvas' DefaultDrawingAtributes to
    a Listbox, called lbDrawingAttributes.-->
<InkCanvas Name="inkCanvas1" Background="LightGreen"
    Canvas.Top="400" Canvas.Left="0"
    Height="400" Width="400"
    DefaultDrawingAttributes="{Binding
        ElementName=lbDrawingAttributes, Path=SelectedItem}">
</InkCanvas>

<!--Use the array, MyDrawingAttributes, to populate a ListBox-->
<ListBox Name="lbDrawingAttributes"
    Canvas.Top="400" Canvas.Left="450"
    Height="100" Width="100"
    ItemsSource="{StaticResource MyDrawingAttributes}" />
```

The following example declares two `InkCanvas` objects in XAML and establishes data binding between them and other data sources. The first `InkCanvas`, called `ic`, is bound to two data sources. The `EditMode` and `DefaultDrawingAttributes` properties on `ic` are bound to `ListBox` objects, which are in turn bound to arrays defined in the XAML. The `EditMode`, `DefaultDrawingAttributes`, and `Strokes` properties of the second `InkCanvas` are bound to the first `InkCanvas`, `ic`.

```

<Canvas>
    <Canvas.Resources>
        <!--Define an array containing the InkEditingMode Values.-->
        <x:Array x:Key="MyEditingModes" x:Type="{x:Type InkCanvasEditingStyle}">
            <x:Static Member="InkCanvasEditingStyle.Ink"/>
            <x:Static Member="InkCanvasEditingStyle.Select"/>
            <x:Static Member="InkCanvasEditingStyle.EraseByPoint"/>
            <x:Static Member="InkCanvasEditingStyle.EraseByStroke"/>
        </x:Array>

        <!--Define an array containing some DrawingAttributes.-->
        <x:Array x:Key="MyDrawingAttributes"
            x:Type="{x:Type DrawingAttributes}">
            <DrawingAttributes Color="Black" FitToCurve="true"
                Width="3" Height="3"/>
            <DrawingAttributes Color="Blue" FitToCurve="false"
                Width="5" Height="5"/>
            <DrawingAttributes Color="Red" FitToCurve="true"
                Width="7" Height="7"/>
        </x:Array>

        <!--Create a DataTemplate to display the
            DrawingAttributes shown above-->
        <DataTemplate DataType="{x:Type DrawingAttributes}" >
            <Border Width="80" Height="{Binding Path=Height}">
                <Border.Background>
                    <SolidColorBrush Color="{Binding Path=Color}"/>
                </Border.Background>
            </Border>
        </DataTemplate>
    </Canvas.Resources>

        <!--Bind the first InkCavas' DefaultDrawingAtributes to a
            Listbox, called lbDrawingAttributes, and its EditingMode to
            a ListBox called lbEditMode.-->
    <InkCanvas Name="ic" Background="LightGray"
        Canvas.Top="0" Canvas.Left="0"
        Height="400" Width="200"
        DefaultDrawingAttributes="{Binding
            ElementName=lbDrawingAttributes, Path=SelectedItem}"
        EditingMode=
            "{Binding ElementName=lbEditMode, Path=SelectedItem}"
        >
    </InkCanvas>

        <!--Bind the Strokes, DefaultDrawingAtributes, and, EditingMode properties of
            the second InkCavas the first InkCanvas.-->
    <InkCanvas Background="LightBlue"
        Canvas.Top="0" Canvas.Left="200"
        Height="400" Width="200"
        Strokes="{Binding ElementName=ic, Path=Strokes}"
        DefaultDrawingAttributes="{Binding
            ElementName=ic, Path=DefaultDrawingAttributes}"
        EditingMode="{Binding ElementName=ic, Path=EditMode}">

        <InkCanvas.LayoutTransform>
            <ScaleTransform ScaleX="-1" ScaleY="1" />
        </InkCanvas.LayoutTransform>
    </InkCanvas>

```

```
<!--Use the array, MyEditingModes, to populate a ListBox-->
<ListBox Name="lbEditingMode"
    Canvas.Top="0" Canvas.Left="450"
    Height="100" Width="100"
    ItemsSource="{StaticResource MyEditingModes}" />

<!--Use the array, MyDrawingAttributes, to populate a ListBox-->
<ListBox Name="lbDrawingAttributes"
    Canvas.Top="150" Canvas.Left="450"
    Height="100" Width="100"
    ItemsSource="{StaticResource MyDrawingAttributes}" />

</Canvas>
```

# How to: Analyze Ink with Analysis Hints

3/5/2019 • 5 minutes to read • [Edit Online](#)

An [System.Windows.Ink.AnalysisHintNode](#) provides a hint for the [System.Windows.Ink.InkAnalyzer](#) to which it is attached. The hint applies to the area specified by the [System.Windows.Ink.ContextNode.Location%2A](#) property of the [System.Windows.Ink.AnalysisHintNode](#) and provides extra context to the ink analyzer to improve recognition accuracy. The [System.Windows.Ink.InkAnalyzer](#) applies this context information when analyzing ink obtained from within the hint's area.

## Example

The following example is an application that uses multiple [System.Windows.Ink.AnalysisHintNode](#) objects on a form that accepts ink input. The application uses the [System.Windows.Ink.AnalysisHintNode.Factoid%2A](#) property to provide context information for each entry on the form. The application uses background analysis to analyze the ink and clears the form of all ink five seconds after the user stops adding ink.

```
<Window x:Class="FormAnalyzer"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="FormAnalyzer"
    SizeToContent="WidthAndHeight"
    >
    <StackPanel Orientation="Vertical">
        <InkCanvas Name="xaml_writingCanvas" Height="500" Width="840"
            StrokeCollected="RestartAnalysis" >
            <Grid>
                <Grid.Resources>
                    <Style TargetType="{x:Type Label}">
                        <Setter Property="FontSize" Value="20"/>
                        <Setter Property="FontFamily" Value="Arial"/>
                    </Style>

                    <Style TargetType="{x:Type TextBlock}">
                        <Setter Property="FontSize" Value="18"/>
                        <Setter Property="VerticalAlignment" Value="Center"/>
                    </Style>
                </Grid.Resources>

                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="100"/></ColumnDefinition>
                    <ColumnDefinition Width="160"/></ColumnDefinition>
                    <ColumnDefinition Width="160"/></ColumnDefinition>
                    <ColumnDefinition Width="100"/></ColumnDefinition>
                    <ColumnDefinition Width="160"/></ColumnDefinition>
                    <ColumnDefinition Width="160"/></ColumnDefinition>
                </Grid.ColumnDefinitions>

                <Grid.RowDefinitions>
                    <RowDefinition Height="100"/></RowDefinition>
                    <RowDefinition Height="100"/></RowDefinition>
                    <RowDefinition Height="100"/></RowDefinition>
                    <RowDefinition Height="100"/></RowDefinition>
                    <RowDefinition Height="100"/></RowDefinition>
                </Grid.RowDefinitions>

                <Label Grid.Row="0" Grid.Column="0">Title</Label>
                <Label Grid.Row="1" Grid.Column="0">Director</Label>
                <Label Grid.Row="2" Grid.Column="0">Starring</Label>
                <Label Grid.Row="3" Grid.Column="0">Rating</Label>
            </Grid>
        </InkCanvas>
    </StackPanel>
</Window>
```

```

<Label Grid.Row="3" Grid.Column="3">Year</Label>
<Label Grid.Row="4" Grid.Column="0">Genre</Label>

<TextBlock Name="xaml_blockTitle"
    Grid.Row="0" Grid.Column="1"
    Grid.ColumnSpan="5"/>
<TextBlock Name="xaml_blockDirector"
    Grid.Row="1" Grid.Column="1"
    Grid.ColumnSpan="5"/>
<TextBlock Name="xaml_blockStarring"
    Grid.Row="2" Grid.Column="1"
    Grid.ColumnSpan="5"/>
<TextBlock Name="xaml_blockRating"
    Grid.Row="3" Grid.Column="1"
    Grid.ColumnSpan="2"/>
<TextBlock Name="xaml_blockYear"
    Grid.Row="3" Grid.Column="4"
    Grid.ColumnSpan="2"/>
<TextBlock Name="xaml_blockGenre"
    Grid.Row="4" Grid.Column="1"
    Grid.ColumnSpan="5"/>

<Line Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="6"
StrokeThickness="2" Stroke="Black"
X1="0" Y1="100" X2="840" Y2="100" />
<Line Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="6"
StrokeThickness="2" Stroke="Black"
X1="0" Y1="100" X2="840" Y2="100" />
<Line Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="6"
StrokeThickness="2" Stroke="Black"
X1="0" Y1="100" X2="840" Y2="100" />
<Line Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="6"
StrokeThickness="2" Stroke="Black"
X1="0" Y1="100" X2="840" Y2="100" />
<Line Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="6"
StrokeThickness="2" Stroke="Black"
X1="420" Y1="0" X2="420" Y2="100" />
</Grid>
</InkCanvas>
</StackPanel>
</Window>

```

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Ink;
using System.Windows.Threading;

public partial class FormAnalyzer : Window
{
    private InkAnalyzer analyzer;

    private AnalysisHintNode hintNodeTitle;
    private AnalysisHintNode hintNodeDirector;
    private AnalysisHintNode hintNodeStarring;
    private AnalysisHintNode hintNodeRating;
    private AnalysisHintNode hintNodeYear;
    private AnalysisHintNode hintNodeGenre;

    // Timer that raises an event to
    // clear the InkCanvas.
    private DispatcherTimer strokeRemovalTimer;

    private const int CLEAR_STROKES_DELAY = 5;

    public FormAnalyzer()
    {

```

```

        InitializeComponent();
    }

protected override void OnContentRendered(EventArgs e)
{
    base.OnContentRendered(e);

    // Initialize the Analyzer.
    analyzer = new InkAnalyzer();
    analyzer.ResultsUpdated += 
        new ResultsUpdatedEventHandler(analyzer_ResultsUpdated);

    // Add analysis hints for each form area.
    // Use the absolute Width and Height of the Grid's
    // RowDefinition and ColumnDefinition properties defined in XAML,
    // to calculate the bounds of the AnalysisHintNode objects.
    hintNodeTitle = analyzer.CreateAnalysisHint(
        new Rect(100, 0, 740, 100));
    hintNodeDirector = analyzer.CreateAnalysisHint(
        new Rect(100, 100, 740, 100));
    hintNodeStarring = analyzer.CreateAnalysisHint(
        new Rect(100, 200, 740, 100));
    hintNodeRating = analyzer.CreateAnalysisHint(
        new Rect(100, 300, 320, 100));
    hintNodeYear = analyzer.CreateAnalysisHint(
        new Rect(520, 300, 320, 100));
    hintNodeGenre = analyzer.CreateAnalysisHint(
        new Rect(100, 400, 740, 100));

    //Set the factoids on the hints.
    hintNodeTitle.Factoid = "(!IS_DEFAULT)";
    hintNodeDirector.Factoid = "(!IS_PERSONALNAME_FULLNAME)";
    hintNodeStarring.Factoid = "(!IS_PERSONALNAME_FULLNAME)";
    hintNodeRating.Factoid = "(!IS_DEFAULT)";
    hintNodeYear.Factoid = "(!IS_DATE_YEAR)";
    hintNodeGenre.Factoid = "(!IS_DEFAULT)";
}

/// <summary>
/// InkCanvas.StrokeCollected event handler. Begins
/// ink analysis and starts the timer to clear the strokes.
/// If five seconds pass without a Stroke being added,
/// the strokes on the InkCanvas will be cleared.
/// </summary>
/// <param name="sender">InkCanvas that raises the
/// StrokeCollected event.</param>
/// <param name="args">Contains the event data.</param>
private void RestartAnalysis(object sender,
    InkCanvasStrokeCollectedEventArgs args)
{

    // If strokeRemovalTimer is enabled, stop it.
    if (strokeRemovalTimer != null && strokeRemovalTimer.IsEnabled)
    {
        strokeRemovalTimer.Stop();
    }

    // Restart the timer to clear the strokes in five seconds
    strokeRemovalTimer = new DispatcherTimer(
        TimeSpan.FromSeconds(CLEAR_STROKES_DELAY),
        DispatcherPriority.Normal,
        ClearCanvas,
        Dispatcher.CurrentDispatcher);

    // Add the new stroke to the InkAnalyzer and
    // begin background analysis.
    analyzer.AddStroke(args.Stroke);
    analyzer.BackgroundAnalyze();
}

```

```

/// <summary>
/// Analyzer.ResultsUpdated event handler.
/// </summary>
/// <param name="sender">InkAnalyzer that raises the
/// event.</param>
/// <param name="e">Event data</param>
/// <remarks>This method checks each AnalysisHint for
/// analyzed ink and then populated the TextBlock that
/// corresponds to the area on the form.</remarks>
void analyzer_ResultsUpdated(object sender, ResultsUpdatedEventArgs e)
{
    string recoText;

    recoText = hintNodeTitle.GetRecognizedString();
    if (recoText != "") xaml_blockTitle.Text = recoText;

    recoText = hintNodeDirector.GetRecognizedString();
    if (recoText != "") xaml_blockDirector.Text = recoText;

    recoText = hintNodeStarring.GetRecognizedString();
    if (recoText != "") xaml_blockStarring.Text = recoText;

    recoText = hintNodeRating.GetRecognizedString();
    if (recoText != "") xaml_blockRating.Text = recoText;

    recoText = hintNodeYear.GetRecognizedString();
    if (recoText != "") xaml_blockYear.Text = recoText;

    recoText = hintNodeGenre.GetRecognizedString();
    if (recoText != "") xaml_blockGenre.Text = recoText;
}

//Clear the canvas, but leave the current strokes in the analyzer.
private void ClearCanvas(object sender, EventArgs args)
{
    strokeRemovalTimer.Stop();

    xaml_writingCanvas.Strokes.Clear();
}
}

```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Ink
Imports System.Windows.Threading

Class FormAnalyzer
    Inherits Window

    Private analyzer As InkAnalyzer

    Private hintNodeTitle As AnalysisHintNode
    Private hintNodeDirector As AnalysisHintNode
    Private hintNodeStarring As AnalysisHintNode
    Private hintNodeRating As AnalysisHintNode
    Private hintNodeYear As AnalysisHintNode
    Private hintNodeGenre As AnalysisHintNode

    ' Timer that raises an event to
    ' clear the InkCanvas.
    Private strokeRemovalTimer As DispatcherTimer

    Private Const CLEAR_STROKES_DELAY As Integer = 5

```

```

Public Sub New()

    InitializeComponent()

End Sub

Protected Overrides Sub OnContentRendered(ByVal e As EventArgs)
    MyBase.OnContentRendered(e)

    ' Initialize the Analyzer.
    analyzer = New InkAnalyzer()
    AddHandler analyzer.ResultsUpdated, AddressOf analyzer_ResultsUpdated

    ' Add analysis hints for each form area.
    ' Use the absolute Width and Height of the Grid's
    ' RowDefinition and ColumnDefinition properties defined in XAML,
    ' to calculate the bounds of the AnalysisHintNode objects.
    hintNodeTitle = analyzer.CreateAnalysisHint(New Rect(100, 0, 740, 100))
    hintNodeDirector = analyzer.CreateAnalysisHint(New Rect(100, 100, 740, 100))
    hintNodeStarring = analyzer.CreateAnalysisHint(New Rect(100, 200, 740, 100))
    hintNodeRating = analyzer.CreateAnalysisHint(New Rect(100, 300, 320, 100))
    hintNodeYear = analyzer.CreateAnalysisHint(New Rect(520, 300, 320, 100))
    hintNodeGenre = analyzer.CreateAnalysisHint(New Rect(100, 400, 740, 100))

    'Set the factoids on the hints.
    hintNodeTitle.Factoid = "(!IS_DEFAULT)"
    hintNodeDirector.Factoid = "(!IS_PERSONALNAME_FULLNAME)"
    hintNodeStarring.Factoid = "(!IS_PERSONALNAME_FULLNAME)"
    hintNodeRating.Factoid = "(!IS_DEFAULT)"
    hintNodeYear.Factoid = "(!IS_DATE_YEAR)"
    hintNodeGenre.Factoid = "(!IS_DEFAULT)"

End Sub

' InkCanvas.StrokeCollected event handler. Begins
' ink analysis and starts the timer to clear the strokes.
' If five seconds pass without a Stroke being added,
' the strokes on the InkCanvas will be cleared.
' <param name="sender">InkCanvas that raises the
' StrokeCollected event.</param>
' <param name="args">Contains the event data.</param>
Private Sub RestartAnalysis(ByVal sender As Object, ByVal args As InkCanvasStrokeCollectedEventArgs)

    ' If strokeRemovalTimer is enabled, stop it.
    If Not (strokeRemovalTimer Is Nothing) AndAlso strokeRemovalTimer.IsEnabled Then
        strokeRemovalTimer.Stop()
    End If

    ' Restart the timer to clear the strokes in five seconds
    strokeRemovalTimer = New DispatcherTimer(
        TimeSpan.FromSeconds(CLEAR_STROKES_DELAY), _
        DispatcherPriority.Normal, _
        AddressOf ClearCanvas, _
        System.Windows.Threading.Dispatcher.CurrentDispatcher)

    ' Add the new stroke to the InkAnalyzer and
    ' begin background analysis.
    analyzer.AddStroke(args.Stroke)
    analyzer.BackgroundAnalyze()

End Sub

' Analyzer.ResultsUpdated event handler.
' <param name="sender">InkAnalyzer that raises the
' event.</param>
' <param name="e">Event data</param>
' <remarks>This method checks each AnalysisHint for
' analyzed ink and then populated the TextBlock that
' corresponds to the area on the form.</remarks>

```

```

Private Sub analyzer_ResultsUpdated(ByVal sender As Object, ByVal e As ResultsUpdatedEventArgs)

    Dim recoText As String

    recoText = hintNodeTitle.GetRecognizedString()
    If recoText <> "" Then
        xaml_blockTitle.Text = recoText
    End If

    recoText = hintNodeDirector.GetRecognizedString()
    If recoText <> "" Then
        xaml_blockDirector.Text = recoText
    End If

    recoText = hintNodeStarring.GetRecognizedString()
    If recoText <> "" Then
        xaml_blockStarring.Text = recoText
    End If

    recoText = hintNodeRating.GetRecognizedString()
    If recoText <> "" Then
        xaml_blockRating.Text = recoText
    End If

    recoText = hintNodeYear.GetRecognizedString()
    If recoText <> "" Then
        xaml_blockYear.Text = recoText
    End If

    recoText = hintNodeGenre.GetRecognizedString()
    If recoText <> "" Then
        xaml_blockGenre.Text = recoText
    End If

End Sub

'Clear the canvas, but leave the current strokes in the analyzer.
Private Sub ClearCanvas(ByVal sender As Object, ByVal args As EventArgs)

    strokeRemovalTimer.Stop()

    xaml_writingCanvas.Strokes.Clear()

End Sub

End Class

```

# How to: Rotate Ink

3/5/2019 • 6 minutes to read • [Edit Online](#)

## Example

The following example copies the ink from an [InkCanvas](#) to a [Canvas](#) that contains an [InkPresenter](#). When the application copies the ink, it also rotates the ink 90 degrees clockwise.

```
<Canvas>
    <InkCanvas Name="inkCanvas1" Background="LightBlue"
        Height="200" Width="200"
        Canvas.Top="20" Canvas.Left="20" />

    <Border Name="canvas1" Background="LightGreen"
        Height="200" Width="200" ClipToBounds="True"
        Canvas.Top="20" Canvas.Left="240" >
        <InkPresenter Name="inkPresenter1"/>
    </Border>
    <Button Click="button_Click"
        Canvas.Top="240" Canvas.Left="170">
        Copy and Rotate Strokes
    </Button>
</Canvas>
```

```
// Button.Click event handler that rotates the strokes
// and copies them to a Canvas.
private void button_Click(object sender, RoutedEventArgs e)
{
    StrokeCollection copiedStrokes = inkCanvas1.Strokes.Clone();
    Matrix rotatingMatrix = new Matrix();
    double canvasLeft = Canvas.GetLeft(inkCanvas1);
    double canvasTop = Canvas.GetTop(inkCanvas1);
    Point rotatePoint = new Point(canvas1.Width / 2, canvas1.Height / 2);

    rotatingMatrix.RotateAt(90, rotatePoint.X, rotatePoint.Y);
    copiedStrokes.Transform(rotatingMatrix, false);
    inkPresenter1.Strokes = copiedStrokes;
}
```

## Example

The following example is a custom [Adorner](#) that rotates the strokes on an [InkPresenter](#).

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Ink;

public class RotatingStrokesAdorner : Adorner
{
    // The Thumb to drag to rotate the strokes.
```

```

Thumb rotateHandle;

// The surrounding boarder.
Path outline;

VisualCollection visualChildren;

// The center of the strokes.
Point center;
double lastAngle;

RotateTransform rotation;

const int HANDLEMARGIN = 10;

// The bounds of the Strokes;
Rect strokeBounds = Rect.Empty;

public RotatingStrokesAdorner(UIElement adornedElement)
    : base(adornedElement)
{

    visualChildren = new VisualCollection(this);
    rotateHandle = new Thumb();
    rotateHandle.Cursor = Cursors.SizeNWSE;
    rotateHandle.Width = 20;
    rotateHandle.Height = 20;
    rotateHandle.Background = Brushes.Blue;

    rotateHandle.DragDelta += new DragDeltaEventHandler(rotateHandle_DragDelta);
    rotateHandle.DragCompleted += new DragCompletedEventHandler(rotateHandle_DragCompleted);

    outline = new Path();
    outline.Stroke = Brushes.Blue;
    outline.StrokeThickness = 1;

    visualChildren.Add(outline);
    visualChildren.Add(rotateHandle);

    strokeBounds = AdornedStrokes.GetBounds();
}

/// <summary>
/// Draw the rotation handle and the outline of
/// the element.
/// </summary>
/// <param name="finalSize">The final area within the
/// parent that this element should use to arrange
/// itself and its children.</param>
/// <returns>The actual size used. </returns>
protected override Size ArrangeOverride(Size finalSize)
{
    if (strokeBounds.IsEmpty)
    {
        return finalSize;
    }

    center = new Point(strokeBounds.X + strokeBounds.Width / 2,
                      strokeBounds.Y + strokeBounds.Height / 2);

    // The rectangle that determines the position of the Thumb.
    Rect handleRect = new Rect(strokeBounds.X,
                               strokeBounds.Y - (strokeBounds.Height / 2 +
                                                 HANDLEMARGIN),
                               strokeBounds.Width, strokeBounds.Height);

    if (rotation != null)
    {
        handleRect.Transform(rotation.Value);
    }
}

```

```

        }

        // Draws the thumb and the rectangle around the strokes.
        rotateHandle.Arrange(handleRect);
        outline.Data = new RectangleGeometry(strokeBounds);
        outline.Arrange(new Rect(finalSize));
        return finalSize;
    }

    /// <summary>
    /// Rotates the rectangle representing the
    /// strokes' bounds as the user drags the
    /// Thumb.
    /// </summary>
    void rotateHandle_DragDelta(object sender, DragDeltaEventArgs e)
    {
        // Find the angle of which to rotate the shape. Use the right
        // triangle that uses the center and the mouse's position
        // as vertices for the hypotenuse.

        Point pos = Mouse.GetPosition(this);

        double deltaX = pos.X - center.X;
        double deltaY = pos.Y - center.Y;

        if (deltaY.Equals(0))
        {

            return;
        }

        double tan = deltaX / deltaY;
        double angle = Math.Atan(tan);

        // Convert to degrees.
        angle = angle * 180 / Math.PI;

        // If the mouse crosses the vertical center,
        // find the complementary angle.
        if (deltaY > 0)
        {
            angle = 180 - Math.Abs(angle);
        }

        // Rotate left if the mouse moves left and right
        // if the mouse moves right.
        if (deltaX < 0)
        {
            angle = -Math.Abs(angle);
        }
        else
        {
            angle = Math.Abs(angle);
        }

        if (Double.IsNaN(angle))
        {
            return;
        }

        // Apply the rotation to the strokes' outline.
        rotation = new RotateTransform(angle, center.X, center.Y);
        outline.RenderTransform = rotation;
    }

    /// <summary>
    /// Rotates the strokes to the same angle as outline.
    /// </summary>
    void rotateHandle_DragCompleted(object sender

```

```

void OnRotateHandle_DragCompleted(object sender,
                                  DragCompletedEventArgs e)
{
    if (rotation == null)
    {
        return;
    }

    // Rotate the strokes to match the new angle.
    Matrix mat = new Matrix();
    mat.RotateAt(rotation.Angle - lastAngle, center.X, center.Y);
    AdornedStrokes.Transform(mat, true);

    // Save the angle of the last rotation.
    lastAngle = rotation.Angle;

    // Redraw rotateHandle.
    this.InvalidateArrange();
}

/// <summary>
/// Gets the strokes of the adorned element
/// (in this case, an InkPresenter).
/// </summary>
private StrokeCollection AdornedStrokes
{
    get
    {
        return ((InkPresenter)AdornedElement).Strokes;
    }
}

// Override the VisualChildrenCount and
// GetVisualChild properties to interface with
// the adorner's visual collection.
protected override int VisualChildrenCount
{
    get { return visualChildren.Count; }
}

protected override Visual GetVisualChild(int index)
{
    return visualChildren[index];
}
}

```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Controls.Primitives
Imports System.Windows.Documents
Imports System.Windows.Input
Imports System.Windows.Media
Imports System.Windows.Shapes
Imports System.Windows.Ink

Public Class RotatingStrokesAdorner
    Inherits Adorner

    ' The Thumb to drag to rotate the strokes.
    Private rotateHandle As Thumb

    ' The surrounding boarder.
    Private outline As Path

    Private visualChildren As VisualCollection

```

```

' The center of the strokes.
Private center As Point
Private lastAngle As Double

Private rotation As RotateTransform

Private Const HANDLEMARGIN As Integer = 10

' The bounds of the Strokes;
Private strokeBounds As Rect = Rect.Empty


Public Sub New(ByVal adornedElement As UIElement)
    MyBase.New(adornedElement)

    visualChildren = New VisualCollection(Me)
    rotateHandle = New Thumb()
    rotateHandle.Cursor = Cursors.SizeNWSE
    rotateHandle.Width = 20
    rotateHandle.Height = 20
    rotateHandle.Background = Brushes.Blue

    AddHandler rotateHandle.DragDelta, _
        AddressOf rotateHandle_DragDelta
    AddHandler rotateHandle.DragCompleted, _
        AddressOf rotateHandle_DragCompleted

    outline = New Path()
    outline.Stroke = Brushes.Blue
    outline.StrokeThickness = 1

    visualChildren.Add(outline)
    visualChildren.Add(rotateHandle)

    strokeBounds = AdornedStrokes.GetBounds()

End Sub

''' <summary>
''' Draw the rotation handle and the outline of
''' the element.
''' </summary>
''' <param name="finalSize">The final area within the
''' parent that this element should use to arrange
''' itself and its children.</param>
''' <returns>The actual size used. </returns>
Protected Overrides Function ArrangeOverride(ByVal finalSize As Size) _
    As Size

    If strokeBounds.IsEmpty Then
        Return finalSize
    End If

    center = New Point(strokeBounds.X + strokeBounds.Width / 2, _
        strokeBounds.Y + strokeBounds.Height / 2)

    ' The rectangle that determines the position of the Thumb.
    Dim handleRect As New Rect(strokeBounds.X, _
        strokeBounds.Y - (strokeBounds.Height / 2 + _
            HANDLEMARGIN), _
        strokeBounds.Width, strokeBounds.Height)

    If Not (rotation Is Nothing) Then
        handleRect.Transform(rotation.Value)
    End If

    ' Draws the thumb and the rectangle around the strokes.
    rotateHandle.Arrange(handleRect)

```

```

rotateHandle.Arrange(handleRect)
outline.Data = New RectangleGeometry(strokeBounds)
outline.Arrange(New Rect(finalSize))
Return finalSize

End Function 'ArrangeOverride

''' <summary>
''' Rotates the rectangle representing the
''' strokes' bounds as the user drags the
''' Thumb.
''' </summary>
Private Sub rotateHandle_DragDelta(ByVal sender As Object, _
                                   ByVal e As DragDeltaEventArgs)

    'Find the angle of which to rotate the shape. Use the right
    'triangle that uses the center and the mouse's position
    'as vertices for the hypotenuse.
    Dim pos As Point = Mouse.GetPosition(Me)

    Dim deltaX As Double = pos.X - center.X
    Dim deltaY As Double = pos.Y - center.Y

    If deltaY.Equals(0) Then
        Return
    End If

    Dim tan As Double = deltaX / deltaY
    Dim angle As Double = Math.Atan(tan)

    ' Convert to degrees.
    angle = angle * 180 / Math.PI

    ' If the mouse crosses the vertical center,
    ' find the complementary angle.
    If deltaY > 0 Then
        angle = 180 - Math.Abs(angle)
    End If

    ' Rotate left if the mouse moves left and right
    ' if the mouse moves right.
    If deltaX < 0 Then
        angle = -Math.Abs(angle)
    Else
        angle = Math.Abs(angle)
    End If

    If Double.IsNaN(angle) Then
        Return
    End If

    ' Apply the rotation to the strokes' outline.
    rotation = New RotateTransform(angle, center.X, center.Y)
    outline.RenderTransform = rotation

End Sub

''' <summary>
''' Rotates the strokes to the same angle as outline.
''' </summary>
Private Sub rotateHandle_DragCompleted(ByVal sender As Object, _
                                       ByVal e As DragCompletedEventArgs)

    If rotation Is Nothing Then
        Return
    End If

```

```

    ' Rotate the strokes to match the new angle.
    Dim mat As New Matrix()
    mat.RotateAt(rotation.Angle - lastAngle, center.X, center.Y)
    AdornedStrokes.Transform(mat, True)

    ' Save the angle of the last rotation.
    lastAngle = rotation.Angle

    ' Redraw rotateHandle.
    Me.InvalidateArrange()

End Sub

''' <summary>
''' Gets the strokes of the adorned element
''' (in this case, an InkPresenter).
''' </summary>
Private ReadOnly Property AdornedStrokes() As StrokeCollection
    Get
        Return CType(AdornedElement, InkPresenter).Strokes
    End Get
End Property

' Override the VisualChildrenCount and
' GetVisualChild properties to interface with
' the adorner's visual collection.

Protected Overrides ReadOnly Property VisualChildrenCount() As Integer
    Get
        Return visualChildren.Count
    End Get
End Property

Protected Overrides Function GetVisualChild(ByVal index As Integer) As Visual
    Return visualChildren(index)

End Function 'GetVisualChild
End Class

```

The following example is a Extensible Application Markup Language (XAML) file that defines an [InkPresenter](#) and populates it with ink. The `Window_Loaded` event handler adds the custom adorner to the [InkPresenter](#).

```

<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Rotating Strokes Adorner" Height="500" Width="500"
    Loaded="Window_Loaded"
    >
<InkPresenter Name="inkPresenter1" >
    <InkPresenter.Strokes>
        ALMDAwRIEEU1BQE4GSAyCQD0/wIB6SI6RTMJAPifAgFaIDpFOAgA/gMAAACAfxEAAIA/
        HwkRAAAAAAAA8D8K1wE1h/CPd4SB4NA40icCjcGjcClcDj8Lh8DgUSkUmmU6nUmoUuk
        0ukUCQKVyeHz+rzuly+bzORx+BReRQ+RTaRCH8JyXhPbgcPicPh8Pg80h0qk1SoVGrV
        0o0mi0Xi8rm9Xr9Dqc/p87pc/k8XicHicOj1CoVKtVmV1GqUaiUH1Yg8e14akXK7m7T
        cSJgQgghEyym5zx6+PACk4dhPwg/fhCbxY8dp4p2tqnqxyvbP085z1X1aswhvCd94Tq
        55DRUGi4+Tk60Ln4KLko0ej06ig5KTioOPCD9LlHmrzNxMRCCc3ec8+fe4AKQBmE/Cw
        9+FkPNvl0dkrYsWa+acp3Z8er0IT8JaX4S6+FbFilbHNvvPXNJbFqluxghKc5DkwrVF
        GEEIJ1w5eLKYAKShuF+Dnr40a8HVHXNPFFFFho8VFkqsMRYuuuvJxiF+F9r4Xx8HFiqs
        FNcirnweDw9+LvviixdV0+GhONmlj3wjN0cSCEYTnfLy4oA
    </InkPresenter.Strokes>
</InkPresenter>
</Window>

```

```
void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Add the rotating strokes adorner to the InkPresenter.
    adornerLayer = AdornerLayer.GetAdornerLayer(inkPresenter1);
    adorner = new RotatingStrokesAdorner(inkPresenter1);

    adornerLayer.Add(adorner);
}
```

```
Private Sub Window_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)

    ' Add the rotating strokes adorner to the InkPresenter.
    adornerLayer = adornerLayer.GetAdornerLayer(inkPresenter1)
    adorner = New RotatingStrokesAdorner(inkPresenter1)

    adornerLayer.Add(adorner)

End Sub
```

# Disable the RealTimeStylus for WPF Applications

9/14/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) has built in support for processing Windows 7 touch input. The support comes through the tablet platform's real-time stylus input as [OnStylusDown](#), [OnStylusUp](#), and [OnStylusMove](#) events. Windows 7 also provides multi-touch input as Win32 WM\_TOUCH window messages. These two APIs are mutually exclusive on the same HWND. Enabling touch input via the tablet platform (the default for WPF applications) disables WM\_TOUCH messages. As a result, to use WM\_TOUCH to receive touch messages from a WPF window, you must disable the built-in stylus support in WPF. This is applicable in a scenario such as a WPF window hosting a component that uses WM\_TOUCH.

To disable WPF listening to stylus input, remove any tablet support added by the WPF window.

## Example

The following sample code shows how to remove the default tablet platform support by using reflection.

```
public static void DisableWPFTabletSupport()
{
    // Get a collection of the tablet devices for this window.
    TabletDeviceCollection devices = System.Windows.Input.Tablet.TabletDevices;

    if (devices.Count > 0)
    {
        // Get the Type of InputManager.
        Type inputManagerType = typeof(System.Windows.Input.InputManager);

        // Call the StylusLogic method on the InputManager.Current instance.
        object stylusLogic = inputManagerType.InvokeMember("StylusLogic",
            BindingFlags.GetProperty | BindingFlags.Instance | BindingFlags.NonPublic,
            null, InputManager.Current, null);

        if (stylusLogic != null)
        {
            // Get the type of the stylusLogic returned from the call to StylusLogic.
            Type stylusLogicType = stylusLogic.GetType();

            // Loop until there are no more devices to remove.
            while (devices.Count > 0)
            {
                // Remove the first tablet device in the devices collection.
                stylusLogicType.InvokeMember("OnTabletRemoved",
                    BindingFlags.InvokeMethod | BindingFlags.Instance | BindingFlags.NonPublic,
                    null, stylusLogic, new object[] { (uint)0 });
            }
        }
    }
}
```

## See also

- [Intercepting Input from the Stylus](#)

# Drag and Drop

8/22/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a highly flexible drag and drop infrastructure which supports dragging and dropping of data within both WPF applications as well as other Windows applications.

## In This Section

[Drag and Drop Overview](#)

[Data and Data Objects](#)

[Walkthrough: Enabling Drag and Drop on a User Control](#)

[How-to Topics](#)

## Reference

[DataFormat](#)

[DataObject](#)

[DragDrop](#)

[DragDropEffects](#)

[DragEventHandler](#)

[TextDataFormat](#)

## Related Sections

# Drag and Drop Overview

10/12/2019 • 16 minutes to read • [Edit Online](#)

This topic provides an overview of drag-and-drop support in Windows Presentation Foundation (WPF) applications. Drag-and-drop commonly refers to a method of data transfer that involves using a mouse (or some other pointing device) to select one or more objects, dragging these objects over some desired drop target in the user interface (UI), and dropping them.

## Drag-and-Drop Support in WPF

Drag-and-drop operations typically involve two parties: a drag source from which the dragged object originates and a drop target which receives the dropped object. The drag source and drop target may be UI elements in the same application or a different application.

The type and number of objects that can be manipulated with drag-and-drop is completely arbitrary. For example, files, folders, and selections of content are some of the more common objects manipulated through drag-and-drop operations.

The particular actions performed during a drag-and-drop operation are application specific, and often determined by context. For example, dragging a selection of files from one folder to another on the same storage device moves the files by default, whereas dragging files from a Universal Naming Convention (UNC) share to a local folder copies the files by default.

The drag-and-drop facilities provided by WPF are designed to be highly flexible and customizable to support a wide variety of drag-and-drop scenarios. Drag-and-drop supports manipulating objects within a single application, or between different applications. Dragging-and-dropping between WPF applications and other Windows applications is also fully supported.

In WPF, any [UIElement](#) or [ContentElement](#) can participate in drag-and-drop. The events and methods required for drag-and-drop operations are defined in the [DragDrop](#) class. The [UIElement](#) and [ContentElement](#) classes contain aliases for the [DragDrop](#) attached events so that the events appear in the class members list when a [UIElement](#) or [ContentElement](#) is inherited as a base element. Event handlers that are attached to these events are attached to the underlying [DragDrop](#) attached event and receive the same event data instance. For more information, see the [UIElement.Drop](#) event.

### IMPORTANT

OLE drag-and-drop does not work while in the Internet zone.

## Data Transfer

Drag-and-drop is part of the more general area of data transfer. Data transfer includes drag-and-drop and copy-and-paste operations. A drag-and-drop operation is analogous to a copy-and-paste or cut-and-paste operation that is used to transfer data from one object or application to another by using the system clipboard. Both types of operations require:

- A source object that provides the data.
- A way to temporarily store the transferred data.
- A target object that receives the data.

In a copy-and-paste operation, the system clipboard is used to temporarily store the transferred data; in a drag-and-drop operation, a [DataObject](#) is used to store the data. Conceptually, a data object consists of one or more pairs of an [Object](#) that contains the actual data, and a corresponding data format identifier.

The drag source initiates a drag-and-drop operation by calling the static [DragDrop.DoDragDrop](#) method and passing the transferred data to it. The [DoDragDrop](#) method will automatically wrap the data in a [DataObject](#) if necessary. For greater control over the data format, you can wrap the data in a [DataObject](#) before passing it to the [DoDragDrop](#) method. The drop target is responsible for extracting the data from the [DataObject](#). For more information about working with data objects, see [Data and Data Objects](#).

The source and target of a drag-and-drop operation are UI elements; however, the data that is actually being transferred typically does not have a visual representation. You can write code to provide a visual representation of the data that is dragged, such as occurs when dragging files in Windows Explorer. By default, feedback is provided to the user by changing the cursor to represent the effect that the drag-and-drop operation will have on the data, such as whether the data will be moved or copied.

### Drag-and-Drop Effects

Drag-and-drop operations can have different effects on the transferred data. For example, you can copy the data or you can move the data. WPF defines a [DragDropEffects](#) enumeration that you can use to specify the effect of a drag-and-drop operation. In the drag source, you can specify the effects that the source will allow in the [DoDragDrop](#) method. In the drop target, you can specify the effect that the target intends in the [Effects](#) property of the [DragEventArgs](#) class. When the drop target specifies its intended effect in the [DragOver](#) event, that information is passed back to the drag source in the [GiveFeedback](#) event. The drag source uses this information to inform the user what effect the drop target intends to have on the data. When the data is dropped, the drop target specifies its actual effect in the [Drop](#) event. That information is passed back to the drag source as the return value of the [DoDragDrop](#) method. If the drop target returns an effect that is not in the drag sources list of `allowedEffects`, the drag-and-drop operation is cancelled without any data transfer occurring.

It is important to remember that in WPF, the [DragDropEffects](#) values are only used to provide communication between the drag source and the drop target regarding the effects of the drag-and-drop operation. The actual effect of the drag-and-drop operation depends on you to write the appropriate code in your application.

For example, the drop target might specify that the effect of dropping data on it is to move the data. However, to move the data, it must be both added to the target element and removed from the source element. The source element might indicate that it allows moving the data, but if you do not provide the code to remove the data from the source element, the end result will be that the data is copied, and not moved.

## Drag-and-Drop Events

Drag-and-drop operations support an event driven model. Both the drag source and the drop target use a standard set of events to handle drag-and-drop operations. The following tables summarize the standard drag-and-drop events. These are attached events on the [DragDrop](#) class. For more information about attached events, see [Attached Events Overview](#).

### Drag Source Events

EVENT	SUMMARY
<a href="#">GiveFeedback</a>	This event occurs continuously during a drag-and-drop operation, and enables the drop source to give feedback information to the user. This feedback is commonly given by changing the appearance of the mouse pointer to indicate the effects allowed by the drop target. This is a bubbling event.

EVENT	SUMMARY
QueryContinueDrag	This event occurs when there is a change in the keyboard or mouse button states during a drag-and-drop operation, and enables the drop source to cancel the drag-and-drop operation depending on the key/button states. This is a bubbling event.
PreviewGiveFeedback	Tunneling version of <a href="#">GiveFeedback</a> .
PreviewQueryContinueDrag	Tunneling version of <a href="#">QueryContinueDrag</a> .

## Drop Target Events

EVENT	SUMMARY
DragEnter	This event occurs when an object is dragged into the drop target's boundary. This is a bubbling event.
DragLeave	This event occurs when an object is dragged out of the drop target's boundary. This is a bubbling event.
DragOver	This event occurs continuously while an object is dragged (moved) within the drop target's boundary. This is a bubbling event.
Drop	This event occurs when an object is dropped on the drop target. This is a bubbling event.
PreviewDragEnter	Tunneling version of <a href="#">DragEnter</a> .
PreviewDragLeave	Tunneling version of <a href="#">DragLeave</a> .
PreviewDragOver	Tunneling version of <a href="#">DragOver</a> .
PreviewDrop	Tunneling version of <a href="#">Drop</a> .

To handle drag-and-drop events for instances of an object, add handlers for the events listed in the preceding tables. To handle drag-and-drop events at the class level, override the corresponding virtual `On*Event` and `On*PreviewEvent` methods. For more information, see [Class Handling of Routed Events by Control Base Classes](#).

## Implementing Drag-and-Drop

A UI element can be a drag source, a drop target, or both. To implement basic drag-and-drop, you write code to initiate the drag-and-drop operation and to process the dropped data. You can enhance the drag-and-drop experience by handling optional drag-and-drop events.

To implement basic drag-and-drop, you will complete the following tasks:

- Identify the element that will be a drag source. A drag source can be a [UIElement](#) or a [ContentElement](#).
- Create an event handler on the drag source that will initiate the drag-and-drop operation. The event is typically the [MouseMove](#) event.
- In the drag source event handler, call the [DoDragDrop](#) method to initiate the drag-and-drop operation. In the [DoDragDrop](#) call, specify the drag source, the data to be transferred, and the allowed effects.

- Identify the element that will be a drop target. A drop target can be [UIElement](#) or a [ContentElement](#).
- On the drop target, set the [AllowDrop](#) property to `true`.
- In the drop target, create a [Drop](#) event handler to process the dropped data.
- In the [Drop](#) event handler, extract the data from the [DragEventArgs](#) by using the [GetDataPresent](#) and [GetData](#) methods.
- In the [Drop](#) event handler, use the data to perform the desired drag-and-drop operation.

You can enhance your drag-and-drop implementation by creating a custom [DataObject](#) and by handling optional drag source and drop target events, as shown in the following tasks:

- To transfer custom data or multiple data items, create a [DataObject](#) to pass to the [DoDragDrop](#) method.
- To perform additional actions during a drag, handle the [DragEnter](#), [DragOver](#), and [DragLeave](#) events on the drop target.
- To change the appearance of the mouse pointer, handle the [GiveFeedback](#) event on the drag source.
- To change how the drag-and-drop operation is canceled, handle the [QueryContinueDrag](#) event on the drag source.

## Drag-and-Drop Example

This section describes how to implement drag-and-drop for an [Ellipse](#) element. The [Ellipse](#) is both a drag source and a drop target. The transferred data is the string representation of the ellipse's [Fill](#) property. The following XAML shows the [Ellipse](#) element and the drag-and-drop related events that it handles. For complete steps on how to implement drag-and-drop, see [Walkthrough: Enabling Drag and Drop on a User Control](#).

```
<Ellipse Height="50" Width="50" Fill="Green"
    MouseMove="ellipse_MouseMove"
    GiveFeedback="ellipse_GiveFeedback"
    AllowDrop="True"
    DragEnter="ellipse_DragEnter" DragLeave="ellipse_DragLeave"
    DragOver="ellipse_DragOver" Drop="ellipse_Drop" />
```

### Enabling an Element to be a Drag Source

An object that is a drag source is responsible for:

- Identifying when a drag occurs.
- Initiating the drag-and-drop operation.
- Identifying the data to be transferred.
- Specifying the effects that the drag-and-drop operation is allowed to have on the transferred data.

The drag source may also give feedback to the user regarding the allowed actions (move, copy, none), and can cancel the drag-and-drop operation based on additional user input, such as pressing the ESC key during the drag.

It is the responsibility of your application to determine when a drag occurs, and then initiate the drag-and-drop operation by calling the [DoDragDrop](#) method. Typically, this is when a [MouseMove](#) event occurs over the element to be dragged while a mouse button is pressed. The following example shows how to initiate a drag-and-drop operation from the [MouseMove](#) event handler of an [Ellipse](#) element to make it a drag source. The transferred data is the string representation of the ellipse's [Fill](#) property.

```

private void ellipse_MouseMove(object sender, MouseEventArgs e)
{
    Ellipse ellipse = sender as Ellipse;
    if (ellipse != null && e.LeftButton == MouseButtonState.Pressed)
    {
        DragDrop.DoDragDrop(ellipse,
                            ellipse.Fill.ToString(),
                            DragDropEffects.Copy);
    }
}

```

```

Private Sub Ellipse_MouseMove(ByVal sender As System.Object, ByVal e As System.Windows.Input.MouseEventArgs)
    Dim ellipse = TryCast(sender, Ellipse)
    If ellipse IsNot Nothing AndAlso e.LeftButton = MouseButtonState.Pressed Then
        DragDrop.DoDragDrop(ellipse, ellipse.Fill.ToString(), DragDropEffects.Copy)
    End If
End Sub

```

Inside of the [MouseMove](#) event handler, call the [DoDragDrop](#) method to initiate the drag-and-drop operation.

The [DoDragDrop](#) method takes three parameters:

- `dragSource` – A reference to the dependency object that is the source of the transferred data; this is typically the source of the [MouseMove](#) event.
- `data` - An object that contains the transferred data, wrapped in a [DataObject](#).
- `allowedEffects` - One of the [DragDropEffects](#) enumeration values that specifies the permitted effects of the drag-and-drop operation.

Any serializable object can be passed in the `data` parameter. If the data is not already wrapped in a [DataObject](#), it will automatically be wrapped in a new [DataObject](#). To pass multiple data items, you must create the [DataObject](#) yourself, and pass it to the [DoDragDrop](#) method. For more information, see [Data and Data Objects](#).

The `allowedEffects` parameter is used to specify what the drag source will allow the drop target to do with the transferred data. The common values for a drag source are [Copy](#), [Move](#), and [All](#).

#### **NOTE**

The drop target is also able to specify what effects it intends in response to the dropped data. For example, if the drop target does not recognize the data type to be dropped, it can refuse the data by setting its allowed effects to [None](#). It typically does this in its [DragOver](#) event handler.

A drag source can optionally handle the [GiveFeedback](#) and [QueryContinueDrag](#) events. These events have default handlers that are used unless you mark the events as handled. You will typically ignore these events unless you have a specific need to change their default behavior.

The [GiveFeedback](#) event is raised continuously while the drag source is being dragged. The default handler for this event checks whether the drag source is over a valid drop target. If it is, it checks the allowed effects of the drop target. It then gives feedback to the end user regarding the allowed drop effects. This is typically done by changing the mouse cursor to a no-drop, copy, or move cursor. You should only handle this event if you need to use custom cursors to provide feedback to the user. If you handle this event, be sure to mark it as handled so that the default handler does not override your handler.

The [QueryContinueDrag](#) event is raised continuously while the drag source is being dragged. You can handle this event to determine what action ends the drag-and-drop operation based on the state of the ESC, SHIFT, CTRL, and ALT keys, as well as the state of the mouse buttons. The default handler for this event cancels the drag-and-

drop operation if the ESC key is pressed, and drops the data if the mouse button is released.

#### Caution

These events are raised continuously during the drag-and-drop operation. Therefore, you should avoid resource-intensive tasks in the event handlers. For example, use a cached cursor instead of creating a new cursor each time the [GiveFeedback](#) event is raised.

### Enabling an Element to be a Drop Target

An object that is a drop target is responsible for:

- Specifying that it is a valid drop target.
- Responding to the drag source when it drags over the target.
- Checking that the transferred data is in a format that it can receive.
- Processing the dropped data.

To specify that an element is a drop target, you set its [AllowDrop](#) property to `true`. The drop target events will then be raised on the element so that you can handle them. During a drag-and-drop operation, the following sequence of events occurs on the drop target:

1. [DragEnter](#)
2. [DragOver](#)
3. [DragLeave](#) or [Drop](#)

The [DragEnter](#) event occurs when the data is dragged into the drop target's boundary. You typically handle this event to provide a preview of the effects of the drag-and-drop operation, if appropriate for your application. Do not set the [DragEventArgs.Effects](#) property in the [DragEnter](#) event, as it will be overwritten in the [DragOver](#) event.

The following example shows the [DragEnter](#) event handler for an [Ellipse](#) element. This code previews the effects of the drag-and-drop operation by saving the current [Fill](#) brush. It then uses the [GetDataPresent](#) method to check whether the [DataObject](#) being dragged over the ellipse contains string data that can be converted to a [Brush](#). If so, the data is extracted using the [GetData](#) method. It is then converted to a [Brush](#) and applied to the ellipse. The change is reverted in the [DragLeave](#) event handler. If the data cannot be converted to a [Brush](#), no action is performed.

```

private Brush _previousFill = null;
private void ellipse_DragEnter(object sender, DragEventArgs e)
{
    Ellipse ellipse = sender as Ellipse;
    if (ellipse != null)
    {
        // Save the current Fill brush so that you can revert back to this value in DragLeave.
        _previousFill = ellipse.Fill;

        // If the DataObject contains string data, extract it.
        if (e.Data.GetDataPresent(DataFormats.StringFormat))
        {
            string dataString = (string)e.Data.GetData(DataFormats.StringFormat);

            // If the string can be converted into a Brush, convert it.
            BrushConverter converter = new BrushConverter();
            if (converter.IsValid(dataString))
            {
                Brush newFill = (Brush)converter.ConvertFromString(dataString);
                ellipse.Fill = newFill;
            }
        }
    }
}

```

```

Private _previousFill As Brush = Nothing
Private Sub Ellipse_DragEnter(ByVal sender As System.Object, ByVal e As System.Windows.DragEventArgs)
    Dim ellipse = TryCast(sender, Ellipse)
    If ellipse IsNot Nothing Then
        ' Save the current Fill brush so that you can revert back to this value in DragLeave.
        _previousFill = ellipse.Fill

        ' If the DataObject contains string data, extract it.
        If e.Data.GetDataPresent(DataFormats.StringFormat) Then
            Dim dataString = e.Data.GetData(DataFormats.StringFormat)

            ' If the string can be converted into a Brush, convert it.
            Dim converter As New BrushConverter()
            If converter.IsValid(dataString) Then
                Dim newFill As Brush = CType(converter.ConvertFromString(dataString), Brush)
                ellipse.Fill = newFill
            End If
        End If
    End If
End Sub

```

The [DragOver](#) event occurs continuously while the data is dragged over the drop target. This event is paired with the [GiveFeedback](#) event on the drag source. In the [DragOver](#) event handler, you typically use the [GetDataPresent](#) and [GetData](#) methods to check whether the transferred data is in a format that the drop target can process. You can also check whether any modifier keys are pressed, which will typically indicate whether the user intends a move or copy action. After these checks are performed, you set the [DragEventArgs.Effects](#) property to notify the drag source what effect dropping the data will have. The drag source receives this information in the [GiveFeedback](#) event args, and can set an appropriate cursor to give feedback to the user.

The following example shows the [DragOver](#) event handler for an [Ellipse](#) element. This code checks to see if the [DataObject](#) being dragged over the ellipse contains string data that can be converted to a [Brush](#). If so, it sets the [DragEventArgs.Effects](#) property to [Copy](#). This indicates to the drag source that the data can be copied to the ellipse. If the data cannot be converted to a [Brush](#), the [DragEventArgs.Effects](#) property is set to [None](#). This indicates to the drag source that the ellipse is not a valid drop target for the data.

```

private void ellipse_DragOver(object sender, DragEventArgs e)
{
    e.Effects = DragDropEffects.None;

    // If the DataObject contains string data, extract it.
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string dataString = (string)e.Data.GetData(DataFormats.StringFormat);

        // If the string can be converted into a Brush, allow copying.
        BrushConverter converter = new BrushConverter();
        if (converter.IsValid(dataString))
        {
            e.Effects = DragDropEffects.Copy | DragDropEffects.Move;
        }
    }
}

```

```

Private Sub Ellipse_DragOver(ByVal sender As System.Object, ByVal e As System.Windows.DragEventArgs)
    e.Effects = DragDropEffects.None

    ' If the DataObject contains string data, extract it.
    If e.Data.GetDataPresent(DataFormats.StringFormat) Then
        Dim dataString = e.Data.GetData(DataFormats.StringFormat)

        ' If the string can be converted into a Brush, convert it.
        Dim converter As New BrushConverter()
        If converter.IsValid(dataString) Then
            e.Effects = DragDropEffects.Copy Or DragDropEffects.Move
        End If
    End If
End Sub

```

The [DragLeave](#) event occurs when the data is dragged out of the target's boundary without being dropped. You handle this event to undo anything that you did in the [DragEnter](#) event handler.

The following example shows the [DragLeave](#) event handler for an [Ellipse](#) element. This code undoes the preview performed in the [DragEnter](#) event handler by applying the saved [Brush](#) to the ellipse.

```

private void ellipse_DragLeave(object sender, DragEventArgs e)
{
    Ellipse ellipse = sender as Ellipse;
    if (ellipse != null)
    {
        ellipse.Fill = _previousFill;
    }
}

```

```

Private Sub Ellipse_DragLeave(ByVal sender As System.Object, ByVal e As System.Windows.DragEventArgs)
    Dim ellipse = TryCast(sender, Ellipse)
    If ellipse IsNot Nothing Then
        ellipse.Fill = _previousFill
    End If
End Sub

```

The [Drop](#) event occurs when the data is dropped over the drop target; by default, this happens when the mouse button is released. In the [Drop](#) event handler, you use the [GetData](#) method to extract the transferred data from the [DataObject](#) and perform any data processing that your application requires. The [Drop](#) event ends the drag-and-drop operation.

The following example shows the [Drop](#) event handler for an [Ellipse](#) element. This code applies the effects of the drag-and-drop operation, and is similar to the code in the [DragEnter](#) event handler. It checks to see if the [DataObject](#) being dragged over the ellipse contains string data that can be converted to a [Brush](#). If so, the [Brush](#) is applied to the ellipse. If the data cannot be converted to a [Brush](#), no action is performed.

```
private void ellipse_Drop(object sender, DragEventArgs e)
{
    Ellipse ellipse = sender as Ellipse;
    if (ellipse != null)
    {
        // If the DataObject contains string data, extract it.
        if (e.Data.GetDataPresent(DataFormats.StringFormat))
        {
            string dataString = (string)e.Data.GetData(DataFormats.StringFormat);

            // If the string can be converted into a Brush,
            // convert it and apply it to the ellipse.
            BrushConverter converter = new BrushConverter();
            if (converter.IsValid(dataString))
            {
                Brush newFill = (Brush)converter.ConvertFromString(dataString);
                ellipse.Fill = newFill;
            }
        }
    }
}
```

```
Private Sub Ellipse_Drop(ByVal sender As System.Object, ByVal e As System.Windows.DragEventArgs)
    Dim ellipse = TryCast(sender, Ellipse)
    If ellipse IsNot Nothing Then

        ' If the DataObject contains string data, extract it.
        If e.Data.GetDataPresent(DataFormats.StringFormat) Then
            Dim dataString = e.Data.GetData(DataFormats.StringFormat)

            ' If the string can be converted into a Brush, convert it.
            Dim converter As New BrushConverter()
            If converter.IsValid(dataString) Then
                Dim newFill As Brush = CType(converter.ConvertFromString(dataString), Brush)
                ellipse.Fill = newFill
            End If
        End If
    End Sub
```

## See also

- [Clipboard](#)
- [Walkthrough: Enabling Drag and Drop on a User Control](#)
- [How-to Topics](#)
- [Drag and Drop](#)

# Data and Data Objects

8/22/2019 • 8 minutes to read • [Edit Online](#)

Data that is transferred as part of a drag-and-drop operation is stored in a data object. Conceptually, a data object consists of one or more of the following pairs:

- An [Object](#) that contains the actual data.
- A corresponding data format identifier.

The data itself can consist of anything that can be represented as a base [Object](#). The corresponding data format is a string or [Type](#) that provides a hint about what format the data is in. Data objects support hosting multiple data/format pairs; this enables a single data object to provide data in multiple formats.

## Data Objects

All data objects must implement the [IDataObject](#) interface, which provides the following standard set of methods that enable and facilitate data transfer.

METHOD	SUMMARY
<a href="#">GetData</a>	Retrieves a data object in a specified data format.
<a href="#">GetDataPresent</a>	Checks to see whether the data is available in, or can be converted to, a specified format.
<a href="#">GetFormats</a>	Returns a list of formats that the data in this data object is stored in, or can be converted to.
<a href="#">SetData</a>	Stores the specified data in this data object.

WPF provides a basic implementation of [IDataObject](#) in the [DataObject](#) class. The stock [DataObject](#) class is sufficient for many common data transfer scenarios.

There are several pre-defined formats, such as bitmap, CSV, file, HTML, RTF, string, text, and audio. For information about pre-defined data formats provided with WPF, see the [DataFormats](#) class reference topic.

Data objects commonly include a facility for automatically converting data stored in one format to a different format while extracting data; this facility is referred to as auto-convert. When querying for the data formats available in a data object, auto-convertible data formats can be filtered from native data formats by calling the [GetFormats\(Boolean\)](#) or [GetDataPresent\(String, Boolean\)](#) method and specifying the `autoConvert` parameter as `false`. When adding data to a data object with the [SetData\(String, Object, Boolean\)](#) method, auto-conversion of data can be prohibited by setting the `autoConvert` parameter to `false`.

## Working with Data Objects

This section describes common techniques for creating and working with data objects.

### Creating New Data Objects

The [DataObject](#) class provides several overloaded constructors that facilitate populating a new [DataObject](#) instance with a single data/format pair.

The following example code creates a new data object and uses one of the overloaded constructors [DataObject\(DataObject\(String, Object\)\)](#) to initialize the data object with a string and a specified data format. In this case, the data format is specified by a string; the [DataFormats](#) class provides a set of pre-defined type strings. Auto-conversion of the stored data is allowed by default.

```
string stringData = "Some string data to store...";  
string dataFormat = DataFormats.UnicodeText;  
DataObject dataObject = new DataObject(dataFormat, stringData);
```

```
Dim stringData As String = "Some string data to store..."  
Dim dataFormat As String = DataFormats.UnicodeText  
Dim dataObject As New DataObject(dataFormat, stringData)
```

For more examples of code that creates a data object, see [Create a Data Object](#).

## Storing Data in Multiple Formats

A single data object is able to store data in multiple formats. Strategic use of multiple data formats within a single data object potentially makes the data object consumable by a wider variety of drop targets than if only a single data format could be represented. Note that, in general, a drag source must be agnostic about the data formats that are consumable by potential drop targets.

The following example shows how to use the [SetData\(String, Object\)](#) method to add data to a data object in multiple formats.

```
DataObject dataObject = new DataObject();  
string sourceData = "Some string data to store...";  
  
// Encode the source string into Unicode byte arrays.  
byte[] unicodeText = Encoding.Unicode.GetBytes(sourceData); // UTF-16  
byte[] utf8Text = Encoding.UTF8.GetBytes(sourceData);  
byte[] utf32Text = Encoding.UTF32.GetBytes(sourceData);  
  
// The DataFormats class does not provide data format fields for denoting  
// UTF-32 and UTF-8, which are seldom used in practice; the following strings  
// will be used to identify these "custom" data formats.  
string utf32DateFormat = "UTF-32";  
string utf8DateFormat = "UTF-8";  
  
// Store the text in the data object, letting the data object choose  
// the data format (which will be DataFormats.Text in this case).  
dataObject.SetData(sourceData);  
// Store the Unicode text in the data object. Text data can be automatically  
// converted to Unicode (UTF-16 / UCS-2) format on extraction from the data object;  
// Therefore, explicitly converting the source text to Unicode is generally unnecessary, and  
// is done here as an exercise only.  
dataObject.SetData(DataFormats.UnicodeText, unicodeText);  
// Store the UTF-8 text in the data object...  
dataObject.SetData(utf8DateFormat, utf8Text);  
// Store the UTF-32 text in the data object...  
dataObject.SetData(utf32DateFormat, utf32Text);
```

```

Dim dataObject As New DataObject()
Dim sourceData As String = "Some string data to store..."

' Encode the source string into Unicode byte arrays.
Dim unicodeText() As Byte = Encoding.Unicode.GetBytes(sourceData) ' UTF-16
Dim utf8Text() As Byte = Encoding.UTF8.GetBytes(sourceData)
Dim utf32Text() As Byte = Encoding.UTF32.GetBytes(sourceData)

' The DataFormats class does not provide data format fields for denoting
' UTF-32 and UTF-8, which are seldom used in practice; the following strings
' will be used to identify these "custom" data formats.
Dim utf32DateFormat As String = "UTF-32"
Dim utf8DateFormat As String = "UTF-8"

' Store the text in the data object, letting the data object choose
' the data format (which will be DataFormats.Text in this case).
dataObject.SetData(sourceData)

' Store the Unicode text in the data object. Text data can be automatically
' converted to Unicode (UTF-16 / UCS-2) format on extraction from the data object;
' Therefore, explicitly converting the source text to Unicode is generally unnecessary, and
' is done here as an exercise only.
dataObject.SetData(DataFormats.UnicodeText, unicodeText)

' Store the UTF-8 text in the data object...
dataObject.SetData(utf8DateFormat, utf8Text)

' Store the UTF-32 text in the data object...
dataObject.SetData(utf32DateFormat, utf32Text)

```

## Querying a Data Object for Available Formats

Because a single data object can contain an arbitrary number of data formats, data objects include facilities for retrieving a list of available data formats.

The following example code uses the [GetFormats](#) overload to get an array of strings denoting all data formats available in a data object (both native and by auto-convert).

```

DataObject dataObject = new DataObject("Some string data to store...");

// Get an array of strings, each string denoting a data format
// that is available in the data object. This overload of GetDataFormats
// returns all available data formats, native and auto-convertible.
string[] dataFormats = dataObject.GetFormats();

// Get the number of data formats present in the data object, including both
// auto-convertible and native data formats.
int numberOfDataFormats = dataFormats.Length;

// To enumerate the resulting array of data formats, and take some action when
// a particular data format is found, use a code structure similar to the following.
foreach (string dataFormat in dataFormats)
{
    if (dataFormat == DataFormats.Text)
    {
        // Take some action if/when data in the Text data format is found.
        break;
    }
    else if(dataFormat == DataFormats.StringFormat)
    {
        // Take some action if/when data in the string data format is found.
        break;
    }
}

```

```

Dim dataObject As New DataObject("Some string data to store...")

' Get an array of strings, each string denoting a data format
' that is available in the data object. This overload of GetDataFormats
' returns all available data formats, native and auto-convertible.
Dim dataFormats() As String = dataObject.GetFormats()

' Get the number of data formats present in the data object, including both
' auto-convertible and native data formats.
Dim numberOfDataFormats As Integer = dataFormats.Length

' To enumerate the resulting array of data formats, and take some action when
' a particular data format is found, use a code structure similar to the following.
For Each dataFormat As String In dataFormats
    If dataFormat = System.Windows.DataFormats.Text Then
        ' Take some action if/when data in the Text data format is found.
        Exit For
    ElseIf dataFormat = System.Windows.DataFormats.StringFormat Then
        ' Take some action if/when data in the string data format is found.
        Exit For
    End If
Next dataFormat

```

For more examples of code that queries a data object for available data formats, see [List the Data Formats in a Data Object](#). For examples of querying a data object for the presence of a particular data format, see [Determine if a Data Format is Present in a Data Object](#).

### **Retrieving Data from a Data Object**

Retrieving data from a data object in a particular format simply involves calling one of the [GetData](#) methods and specifying the desired data format. One of the [GetDataPresent](#) methods can be used to check for the presence of a particular data format. [GetData](#) returns the data in an [Object](#); depending on the data format, this object can be cast to a type-specific container.

The following example code uses the [GetDataPresent\(String\)](#) overload to check if a specified data format is available (native or by auto-convert). If the specified format is available, the example retrieves the data by using the [GetData\(String\)](#) method.

```

DataObject dataObject = new DataObject("Some string data to store...");

string desiredFormat = DataFormats.UnicodeText;
byte[] data = null;

// Use the GetDataPresent method to check for the presence of a desired data format.
// This particular overload of GetDataPresent looks for both native and auto-convertible
// data formats.
if (dataObject.GetDataPresent(desiredFormat))
{
    // If the desired data format is present, use one of the GetData methods to retrieve the
    // data from the data object.
    data = dataObject.GetData(desiredFormat) as byte[];
}

```

```
Dim dataObject As New DataObject("Some string data to store...")

Dim desiredFormat As String = DataFormats.UnicodeText
Dim data() As Byte = Nothing

' Use the GetDataPresent method to check for the presence of a desired data format.
' This particular overload of GetDataPresent looks for both native and auto-convertible
' data formats.
If dataObject.GetDataPresent(desiredFormat) Then
    ' If the desired data format is present, use one of the GetData methods to retrieve the
    ' data from the data object.
    data = TryCast(dataObject.GetData(desiredFormat), Byte())
End If
```

For more examples of code that retrieves data from a data object, see [Retrieve Data in a Particular Data Format](#).

## Removing Data From a Data Object

Data cannot be directly removed from a data object. To effectively remove data from a data object, follow these steps:

1. Create a new data object that will contain only the data you want to retain.
2. "Copy" the desired data from the old data object to the new data object. To copy the data, use one of the [GetData](#) methods to retrieve an [Object](#) that contains the raw data, and then use one of the [SetData](#) methods to add the data to the new data object.
3. Replace the old data object with the new one.

### NOTE

The [SetData](#) methods only add data to a data object; they do not replace data, even if the data and data format are exactly the same as a previous call. Calling [SetData](#) twice for the same data and data format will result in the data/data format being present twice in the data object.

# Walkthrough: Enabling Drag and Drop on a User Control

9/14/2019 • 17 minutes to read • [Edit Online](#)

This walkthrough demonstrates how to create a custom user control that can participate in drag-and-drop data transfer in Windows Presentation Foundation (WPF).

In this walkthrough, you will create a custom WPF [UserControl](#) that represents a circle shape. You will implement functionality on the control to enable data transfer through drag-and-drop. For example, if you drag from one Circle control to another, the Fill color data is copied from the source Circle to the target. If you drag from a Circle control to a [TextBox](#), the string representation of the Fill color is copied to the [TextBox](#). You will also create a small application that contains two panel controls and a [TextBox](#) to test the drag-and-drop functionality. You will write code that enables the panels to process dropped Circle data, which will enable you to move or copy Circles from the Children collection of one panel to the other.

This walkthrough illustrates the following tasks:

- Create a custom user control.
- Enable the user control to be a drag source.
- Enable the user control to be a drop target.
- Enable a panel to receive data dropped from the user control.

## Prerequisites

You need Visual Studio to complete this walkthrough.

## Create the Application Project

In this section, you will create the application infrastructure, which includes a main page with two panels and a [TextBox](#).

1. Create a new WPF Application project in Visual Basic or Visual C# named `DragDropExample`. For more information, see [Walkthrough: My first WPF desktop application](#).
2. Open `MainWindow.xaml`.
3. Add the following markup between the opening and closing [Grid](#) tags.

This markup creates the user interface for the test application.

```
<Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
<StackPanel Grid.Column="0"
            Background="Beige">
    <TextBox Width="Auto" Margin="2"
            Text="green"/>
</StackPanel>
<StackPanel Grid.Column="1"
            Background="Bisque">
</StackPanel>
```

## Add a New User Control to the Project

In this section, you will add a new user control to the project.

1. On the Project menu, select **Add User Control**.
2. In the **Add New Item** dialog box, change the name to `Circle.xaml`, and click **Add**.

`Circle.xaml` and its code-behind is added to the project.

3. Open `Circle.xaml`.

This file will contain the user interface elements of the user control.

4. Add the following markup to the root **Grid** to create a simple user control that has a blue circle as its UI.

```
<Ellipse x:Name="circleUI"
        Height="100" Width="100"
        Fill="Blue" />
```

5. Open `Circle.xaml.cs` or `Circle.xaml.vb`.

6. In C#, add the following code after the parameterless constructor to create a copy constructor. In Visual Basic, add the following code to create both a parameterless constructor and a copy constructor.

In order to allow the user control to be copied, you add a copy constructor method in the code-behind file. In the simplified Circle user control, you will only copy the Fill and the size of the of the user control.

```
public Circle(Circle c)
{
    InitializeComponent();
    this.circleUI.Height = c.circleUI.Height;
    this.circleUI.Width = c.circleUI.Width;
    this.circleUI.Fill = c.circleUI.Fill;
}
```

```

Public Sub New()
    ' This call is required by the designer.
    InitializeComponent()
End Sub

Public Sub New(ByVal c As Circle)
    InitializeComponent()
    Me.circleUI.Height = c.circleUI.Height
    Me.circleUI.Width = c.circleUI.Height
    Me.circleUI.Fill = c.circleUI.Fill
End Sub

```

## Add the user control to the main window

1. Open MainWindow.xaml.
2. Add the following XAML to the opening [Window](#) tag to create an XML namespace reference to the current application.

```
xmlns:local="clr-namespace:DragDropExample"
```

3. In the first [StackPanel](#), add the following XAML to create two instances of the Circle user control in the first panel.

```
<local:Circle Margin="2" />
<local:Circle Margin="2" />
```

The full XAML for the panel looks like the following.

```

<StackPanel Grid.Column="0"
            Background="Beige">
    <TextBox Width="Auto" Margin="2"
             Text="green"/>
    <local:Circle Margin="2" />
    <local:Circle Margin="2" />
</StackPanel>
<StackPanel Grid.Column="1"
            Background="Bisque">
</StackPanel>

```

## Implement Drag Source Events in the User Control

In this section, you will override the [OnMouseMove](#) method and initiate the drag-and-drop operation.

If a drag is started (a mouse button is pressed and the mouse is moved), you will package the data to be transferred into a [DataObject](#). In this case, the Circle control will package three data items; a string representation of its Fill color, a double representation of its height, and a copy of itself.

### To initiate a drag-and-drop operation

1. Open Circle.xaml.cs or Circle.xaml.vb.
2. Add the following [OnMouseMove](#) override to provide class handling for the [MouseMove](#) event.

```

protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);
    if (e.LeftButton == MouseButtonState.Pressed)
    {
        // Package the data.
        DataObject data = new DataObject();
        data.SetData(DataFormats.StringFormat, circleUI.Fill.ToString());
        data.SetData("Double", circleUI.Height);
        data.SetData("Object", this);

        // Initiate the drag-and-drop operation.
        DragDrop.DoDragDrop(this, data, DragDropEffects.Copy | DragDropEffects.Move);
    }
}

```

```

Protected Overrides Sub OnMouseMove(ByVal e As System.Windows.Input.MouseEventArgs)
    MyBase.OnMouseMove(e)
    If e.LeftButton = MouseButtonState.Pressed Then
        ' Package the data.
        Dim data As New DataObject
        data.SetData(DataFormats.StringFormat, circleUI.Fill.ToString())
        data.SetData("Double", circleUI.Height)
        data.SetData("Object", Me)

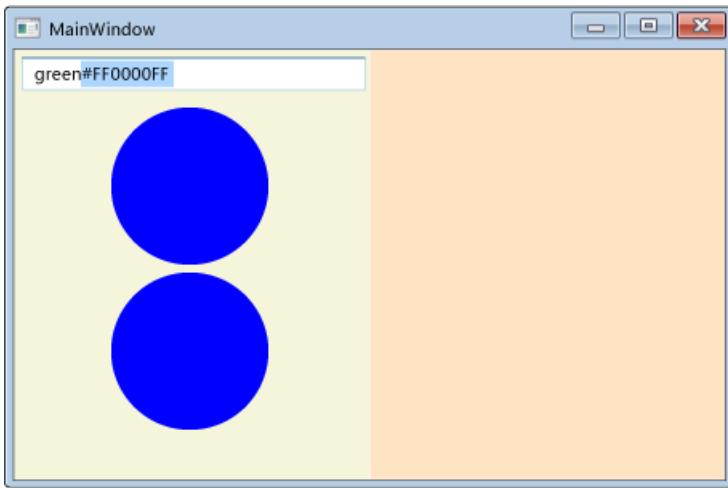
        ' Initiate the drag-and-drop operation.
        DragDrop.DoDragDrop(Me, data, DragDropEffects.Copy Or DragDropEffects.Move)
    End If
End Sub

```

This [OnMouseMove](#) override performs the following tasks:

- Checks whether the left mouse button is pressed while the mouse is moving.
- Packages the Circle data into a [DataObject](#). In this case, the Circle control packages three data items; a string representation of its Fill color, a double representation of its height, and a copy of itself.
- Calls the static [DragDrop.DoDragDrop](#) method to initiate the drag-and-drop operation. You pass the following three parameters to the [DoDragDrop](#) method:
  - `dragSource` – A reference to this control.
  - `data` – The [DataObject](#) created in the previous code.
  - `allowedEffects` – The allowed drag-and-drop operations, which are [Copy](#) or [Move](#).

3. Press **F5** to build and run the application.
4. Click one of the Circle controls and drag it over the panels, the other Circle, and the [TextBox](#). When dragging over the [TextBox](#), the cursor changes to indicate a move.
5. While dragging a Circle over the [TextBox](#), press the **Ctrl** key. Notice how the cursor changes to indicate a copy.
6. Drag and drop a Circle onto the [TextBox](#). The string representation of the Circle's fill color is appended to the [TextBox](#).



By default, the cursor will change during a drag-and-drop operation to indicate what effect dropping the data will have. You can customize the feedback given to the user by handling the [GiveFeedback](#) event and setting a different cursor.

## Give feedback to the user

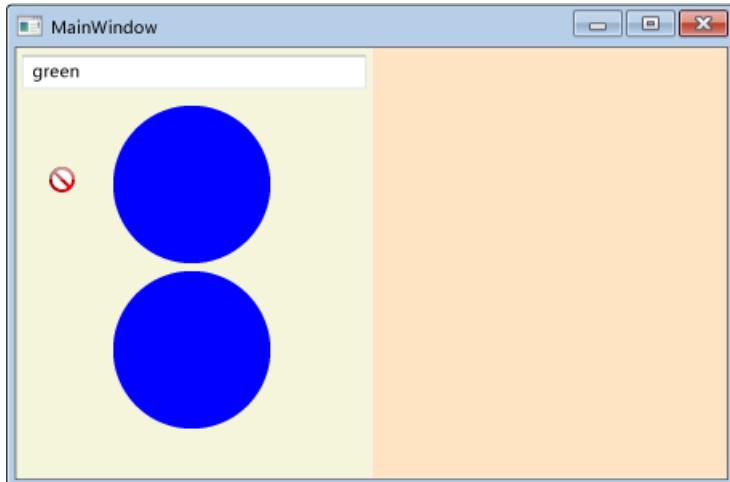
1. Open Circle.xaml.cs or Circle.xaml.vb.
2. Add the following [OnGiveFeedback](#) override to provide class handling for the [GiveFeedback](#) event.

```
protected override void OnGiveFeedback(GiveFeedbackEventArgs e)
{
    base.OnGiveFeedback(e);
    // These Effects values are set in the drop target's
    // DragOver event handler.
    if (e.Effects.HasFlag(DragDropEffects.Copy))
    {
        Mouse.SetCursor(Cursors.Cross);
    }
    else if (e.Effects.HasFlag(DragDropEffects.Move))
    {
        Mouse.SetCursor(Cursors.Pen);
    }
    else
    {
        Mouse.SetCursor(Cursors.No);
    }
    e.Handled = true;
}
```

```
Protected Overrides Sub OnGiveFeedback(ByVal e As System.Windows.GiveFeedbackEventArgs)
    MyBase.OnGiveFeedback(e)
    ' These Effects values are set in the drop target's
    ' DragOver event handler.
    If e.Effects.HasFlag(DragDropEffects.Copy) Then
        Mouse.SetCursor(Cursors.Cross)
    ElseIf e.Effects.HasFlag(DragDropEffects.Move) Then
        Mouse.SetCursor(Cursors.Pen)
    Else
        Mouse.SetCursor(Cursors.No)
    End If
    e.Handled = True
End Sub
```

This [OnGiveFeedback](#) override performs the following tasks:

- Checks the [Effects](#) values that are set in the drop target's [DragOver](#) event handler.
  - Sets a custom cursor based on the [Effects](#) value. The cursor is intended to give visual feedback to the user about what effect dropping the data will have.
3. Press **F5** to build and run the application.
4. Drag one of the Circle controls over the panels, the other Circle, and the [TextBox](#). Notice that the cursors are now the custom cursors that you specified in the [OnGiveFeedback](#) override.



5. Select the text `green` from the [TextBox](#).
6. Drag the `green` text to a Circle control. Notice that the default cursors are shown to indicate the effects of the drag-and-drop operation. The feedback cursor is always set by the drag source.

## Implement Drop Target Events in the User Control

In this section, you will specify that the user control is a drop target, override the methods that enable the user control to be a drop target, and process the data that is dropped on it.

### To enable the user control to be a drop target

1. Open `Circle.xaml`.
2. In the opening [UserControl](#) tag, add the [AllowDrop](#) property and set it to `true`.

```
<UserControl x:Class="DragDropWalkthrough.Circle"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300"
    AllowDrop="True">
```

The [OnDrop](#) method is called when the [AllowDrop](#) property is set to `true` and data from the drag source is dropped on the Circle user control. In this method, you will process the data that was dropped and apply the data to the Circle.

### To process the dropped data

1. Open `Circle.xaml.cs` or `Circle.xaml.vb`.
2. Add the following [OnDrop](#) override to provide class handling for the [Drop](#) event.

```

protected override void OnDrop(DragEventArgs e)
{
    base.OnDrop(e);

    // If the DataObject contains string data, extract it.
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string dataString = (string)e.Data.GetData(DataFormats.StringFormat);

        // If the string can be converted into a Brush,
        // convert it and apply it to the ellipse.
        BrushConverter converter = new BrushConverter();
        if (converter.IsValid(dataString))
        {
            Brush newFill = (Brush)converter.ConvertFromString(dataString);
            circleUI.Fill = newFill;

            // Set Effects to notify the drag source what effect
            // the drag-and-drop operation had.
            // (Copy if CTRL is pressed; otherwise, move.)
            if (e.KeyStates.HasFlag(DragDropKeyStates.ControlKey))
            {
                e.Effects = DragDropEffects.Copy;
            }
            else
            {
                e.Effects = DragDropEffects.Move;
            }
        }
    }
    e.Handled = true;
}

```

```

Protected Overrides Sub OnDrop(ByVal e As System.Windows.DragEventArgs)
    MyBase.OnDrop(e)
    ' If the DataObject contains string data, extract it.
    If e.Data.GetDataPresent(DataFormats.StringFormat) Then
        Dim dataString As String = e.Data.GetData(DataFormats.StringFormat)

        ' If the string can be converted into a Brush,
        ' convert it and apply it to the ellipse.
        Dim converter As New BrushConverter
        If converter.IsValid(dataString) Then
            Dim newFill As Brush = converter.ConvertFromString(dataString)
            circleUI.Fill = newFill

            ' Set Effects to notify the drag source what effect
            ' the drag-and-drop operation had.
            ' (Copy if CTRL is pressed; otherwise, move.)
            If e.KeyStates.HasFlag(DragDropKeyStates.ControlKey) Then
                e.Effects = DragDropEffects.Copy
            Else
                e.Effects = DragDropEffects.Move
            End If
        End If
        e.Handled = True
    End Sub

```

This [OnDrop](#) override performs the following tasks:

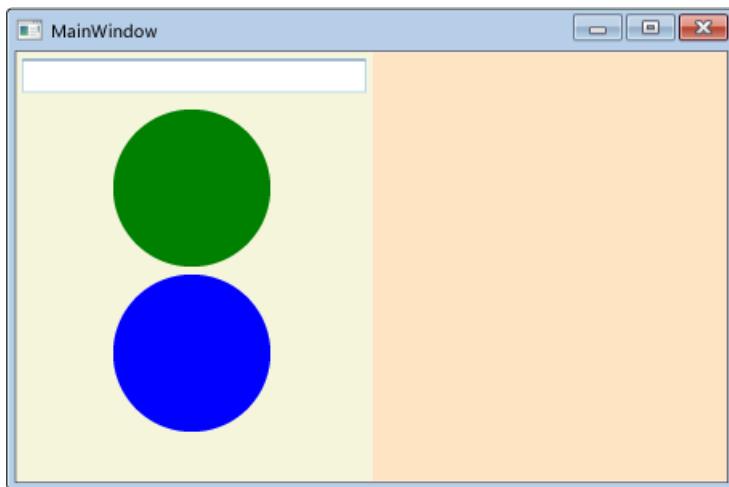
- Uses the [GetDataPresent](#) method to check if the dragged data contains a string object.
- Uses the [GetData](#) method to extract the string data if it is present.

- Uses a [BrushConverter](#) to try to convert the string to a [Brush](#).
- If the conversion is successful, applies the brush to the [Fill](#) of the [Ellipse](#) that provides the UI of the Circle control.
- Marks the [Drop](#) event as handled. You should mark the drop event as handled so that other elements that receive this event know that the Circle user control handled it.

3. Press **F5** to build and run the application.

4. Select the text `green` in the [TextBox](#).

5. Drag the text to a Circle control and drop it. The Circle changes from blue to green.



6. Type the text `green` in the [TextBox](#).

7. Select the text `gre` in the [TextBox](#).

8. Drag it to a Circle control and drop it. Notice that the cursor changes to indicate that the drop is allowed, but the color of the Circle does not change because `gre` is not a valid color.

9. Drag from the green Circle control and drop on the blue Circle control. The Circle changes from blue to green. Notice that which cursor is shown depends on whether the [TextBox](#) or the Circle is the drag source.

Setting the [AllowDrop](#) property to `true` and processing the dropped data is all that is required to enable an element to be a drop target. However, to provide a better user experience, you should also handle the [DragEnter](#), [DragLeave](#), and [DragOver](#) events. In these events, you can perform checks and provide additional feedback to the user before the data is dropped.

When data is dragged over the Circle user control, the control should notify the drag source whether it can process the data that is being dragged. If the control does not know how to process the data, it should refuse the drop. To do this, you will handle the [DragOver](#) event and set the [Effects](#) property.

#### To verify that the data drop is allowed

1. Open `Circle.xaml.cs` or `Circle.xaml.vb`.

2. Add the following [OnDragOver](#) override to provide class handling for the [DragOver](#) event.

```

protected override void OnDragOver(DragEventArgs e)
{
    base.OnDragOver(e);
    e.Effects = DragDropEffects.None;

    // If the DataObject contains string data, extract it.
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string dataString = (string)e.Data.GetData(DataFormats.StringFormat);

        // If the string can be converted into a Brush, allow copying or moving.
        BrushConverter converter = new BrushConverter();
        if (converter.IsValid(dataString))
        {
            // Set Effects to notify the drag source what effect
            // the drag-and-drop operation will have. These values are
            // used by the drag source's GiveFeedback event handler.
            // (Copy if CTRL is pressed; otherwise, move.)
            if (e.KeyStates.HasFlag(DragDropKeyStates.ControlKey))
            {
                e.Effects = DragDropEffects.Copy;
            }
            else
            {
                e.Effects = DragDropEffects.Move;
            }
        }
    }
    e.Handled = true;
}

```

```

Protected Overrides Sub OnDragOver(ByVal e As System.Windows.DragEventArgs)
    MyBase.OnDragOver(e)
    e.Effects = DragDropEffects.None

    ' If the DataObject contains string data, extract it.
    If e.Data.GetDataPresent(DataFormats.StringFormat) Then
        Dim dataString As String = e.Data.GetData(DataFormats.StringFormat)

        ' If the string can be converted into a Brush, allow copying or moving.
        Dim converter As New BrushConverter
        If converter.IsValid(dataString) Then
            ' Set Effects to notify the drag source what effect
            ' the drag-and-drop operation will have. These values are
            ' used by the drag source's GiveFeedback event handler.
            ' (Copy if CTRL is pressed; otherwise, move.)
            If e.KeyStates.HasFlag(DragDropKeyStates.ControlKey) Then
                e.Effects = DragDropEffects.Copy
            Else
                e.Effects = DragDropEffects.Move
            End If
        End If
    End If
    e.Handled = True
End Sub

```

This [OnDragOver](#) override performs the following tasks:

- Sets the [Effects](#) property to [None](#).
- Performs the same checks that are performed in the [OnDrop](#) method to determine whether the Circle user control can process the dragged data.
- If the user control can process the data, sets the [Effects](#) property to [Copy](#) or [Move](#).

3. Press **F5** to build and run the application.
4. Select the text `gre` in the [TextBox](#).
5. Drag the text to a Circle control. Notice that the cursor now changes to indicate that the drop is not allowed because `gre` is not a valid color.

You can further enhance the user experience by applying a preview of the drop operation. For the Circle user control, you will override the [OnDragEnter](#) and [OnDragLeave](#) methods. When the data is dragged over the control, the current background [Fill](#) is saved in a placeholder variable. The string is then converted to a brush and applied to the [Ellipse](#) that provides the Circle's UI. If the data is dragged out of the Circle without being dropped, the original [Fill](#) value is re-applied to the Circle.

### To preview the effects of the drag-and-drop operation

1. Open Circle.xaml.cs or Circle.xaml.vb.

2. In the Circle class, declare a private [Brush](#) variable named `_previousFill` and initialize it to `null`.

```
public partial class Circle : UserControl
{
    private Brush _previousFill = null;
```

```
Public Class Circle
    Private _previousFill As Brush = Nothing
```

3. Add the following [OnDragEnter](#) override to provide class handling for the [DragEnter](#) event.

```
protected override void OnDragEnter(DragEventArgs e)
{
    base.OnDragEnter(e);
    // Save the current Fill brush so that you can revert back to this value in DragLeave.
    _previousFill = circleUI.Fill;

    // If the DataObject contains string data, extract it.
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string dataString = (string)e.Data.GetData(DataFormats.StringFormat);

        // If the string can be converted into a Brush, convert it.
        BrushConverter converter = new BrushConverter();
        if (converter.IsValid(dataString))
        {
            Brush newFill = (Brush)converter.ConvertFromString(dataString.ToString());
            circleUI.Fill = newFill;
        }
    }
}
```

```

Protected Overrides Sub OnDragEnter(ByVal e As System.Windows.DragEventArgs)
    MyBase.OnDragEnter(e)
    _previousFill = circleUI.Fill

    ' If the DataObject contains string data, extract it.
    If e.Data.GetDataPresent(DataFormats.StringFormat) Then
        Dim dataString As String = e.Data.GetData(DataFormats.StringFormat)

        ' If the string can be converted into a Brush, convert it.
        Dim converter As New BrushConverter
        If converter.IsValid(dataString) Then
            Dim newFill As Brush = converter.ConvertFromString(dataString)
            circleUI.Fill = newFill
        End If
    End If
    e.Handled = True
End Sub

```

This [OnDragEnter](#) override performs the following tasks:

- Saves the [Fill](#) property of the [Ellipse](#) in the [\\_previousFill](#) variable.
- Performs the same checks that are performed in the [OnDrop](#) method to determine whether the data can be converted to a [Brush](#).
- If the data is converted to a valid [Brush](#), applies it to the [Fill](#) of the [Ellipse](#).

#### 4. Add the following [OnDragLeave](#) override to provide class handling for the [DragLeave](#) event.

```

protected override void OnDragLeave(DragEventArgs e)
{
    base.OnDragLeave(e);
    // Undo the preview that was applied in OnDragEnter.
    circleUI.Fill = _previousFill;
}

```

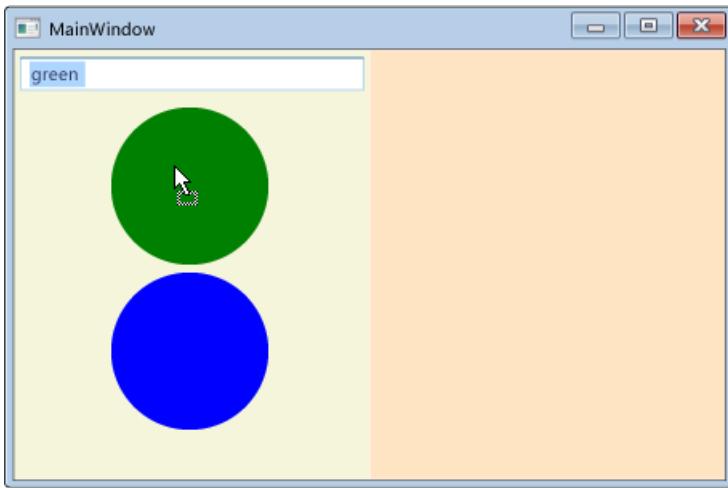
```

Protected Overrides Sub OnDragLeave(ByVal e As System.Windows.DragEventArgs)
    MyBase.OnDragLeave(e)
    ' Undo the preview that was applied in OnDragEnter.
    circleUI.Fill = _previousFill
End Sub

```

This [OnDragLeave](#) override performs the following tasks:

- Applies the [Brush](#) saved in the [\\_previousFill](#) variable to the [Fill](#) of the [Ellipse](#) that provides the UI of the Circle user control.
5. Press **F5** to build and run the application.
  6. Select the text [green](#) in the [TextBox](#).
  7. Drag the text over a Circle control without dropping it. The Circle changes from blue to green.



8. Drag the text away from the Circle control. The Circle changes from green back to blue.

## Enable a Panel to Receive Dropped Data

In this section, you enable the panels that host the Circle user controls to act as drop targets for dragged Circle data. You will implement code that enables you to move a Circle from one panel to another, or to make a copy of a Circle control by holding down the **Ctrl** key while dragging and dropping a Circle.

1. Open MainWindow.xaml.
2. As shown in the following XAML, in each of the **StackPanel** controls, add handlers for the **DragOver** and **Drop** events. Name the **DragOver** event handler, `panel_DragOver`, and name the **Drop** event handler, `panel_Drop`.

```
<StackPanel Grid.Column="0"
    Background="Beige"
    AllowDrop="True"
    DragOver="panel_DragOver"
    Drop="panel_Drop">
    <TextBox Width="Auto" Margin="2"
        Text="green"/>
    <local:Circle Margin="2" />
    <local:Circle Margin="2" />
</StackPanel>
<StackPanel Grid.Column="1"
    Background="Bisque"
    AllowDrop="True"
    DragOver="panel_DragOver"
    Drop="panel_Drop">
</StackPanel>
```

3. Open MainWindows.xaml.cs or MainWindow.xaml.vb.
4. Add the following code for the **DragOver** event handler.

```

private void panel_DragOver(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent("Object"))
    {
        // These Effects values are used in the drag source's
        // GiveFeedback event handler to determine which cursor to display.
        if (e.KeyStates == DragDropKeyStates.ControlKey)
        {
            e.Effects = DragDropEffects.Copy;
        }
        else
        {
            e.Effects = DragDropEffects.Move;
        }
    }
}

```

```

Private Sub panel_DragOver(ByVal sender As System.Object, ByVal e As System.Windows.DragEventArgs)
If e.Data.GetDataPresent("Object") Then
    ' These Effects values are used in the drag source's
    ' GiveFeedback event handler to determine which cursor to display.
    If e.KeyStates = DragDropKeyStates.ControlKey Then
        e.Effects = DragDropEffects.Copy
    Else
        e.Effects = DragDropEffects.Move
    End If
End If
End Sub

```

This [DragOver](#) event handler performs the following tasks:

- Checks that the dragged data contains the "Object" data that was packaged in the [DataObject](#) by the Circle user control and passed in the call to [DoDragDrop](#).
- If the "Object" data is present, checks whether the **Ctrl** key is pressed.
- If the **Ctrl** key is pressed, sets the [Effects](#) property to [Copy](#). Otherwise, set the [Effects](#) property to [Move](#).

5. Add the following code for the [Drop](#) event handler.

```
private void panel_Drop(object sender, DragEventArgs e)
{
    // If an element in the panel has already handled the drop,
    // the panel should not also handle it.
    if (e.Handled == false)
    {
        Panel _panel = (Panel)sender;
        UIElement _element = (UIElement)e.Data.GetData("Object");

        if (_panel != null && _element != null)
        {
            // Get the panel that the element currently belongs to,
            // then remove it from that panel and add it the Children of
            // the panel that its been dropped on.
            Panel _parent = (Panel)VisualTreeHelper.GetParent(_element);

            if (_parent != null)
            {
                if (e.KeyStates == DragDropKeyStates.ControlKey &&
                    e.AllowedEffects.HasFlag(DragDropEffects.Copy))
                {
                    Circle _circle = new Circle((Circle)_element);
                    _panel.Children.Add(_circle);
                    // set the value to return to the DoDragDrop call
                    e.Effects = DragDropEffects.Copy;
                }
                else if (e.AllowedEffects.HasFlag(DragDropEffects.Move))
                {
                    _parent.Children.Remove(_element);
                    _panel.Children.Add(_element);
                    // set the value to return to the DoDragDrop call
                    e.Effects = DragDropEffects.Move;
                }
            }
        }
    }
}
```

```

Private Sub panel_Drop(ByVal sender As System.Object, ByVal e As System.Windows.DragEventArgs)
    ' If an element in the panel has already handled the drop,
    ' the panel should not also handle it.
    If e.Handled = False Then
        Dim _panel As Panel = sender
        Dim _element As UIElement = e.Data.GetData("Object")

        If _panel IsNot Nothing And _element IsNot Nothing Then
            ' Get the panel that the element currently belongs to,
            ' then remove it from that panel and add it the Children of
            ' the panel that its been dropped on.

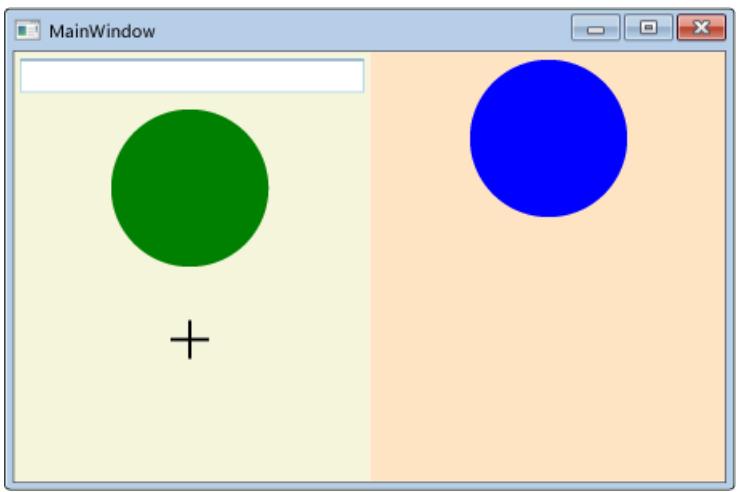
            Dim _parent As Panel = VisualTreeHelper.GetParent(_element)
            If _parent IsNot Nothing Then
                If e.KeyStates = DragDropKeyStates.ControlKey And _
                    e.AllowedEffects.HasFlag(DragDropEffects.Copy) Then
                    Dim _circle As New Circle(_element)
                    _panel.Children.Add(_circle)
                    ' set the value to return to the DoDragDrop call
                    e.Effects = DragDropEffects.Copy
                ElseIf e.AllowedEffects.HasFlag(DragDropEffects.Move) Then
                    _parent.Children.Remove(_element)
                    _panel.Children.Add(_element)
                    ' set the value to return to the DoDragDrop call
                    e.Effects = DragDropEffects.Move
                End If
            End If
        End If
    End If
End Sub

```

This [Drop](#) event handler performs the following tasks:

- Checks whether the [Drop](#) event has already been handled. For instance, if a Circle is dropped on another Circle which handles the [Drop](#) event, you do not want the panel that contains the Circle to also handle it.
- If the [Drop](#) event is not handled, checks whether the **Ctrl** key is pressed.
- If the **Ctrl** key is pressed when the [Drop](#) happens, makes a copy of the Circle control and add it to the [Children](#) collection of the [StackPanel](#).
- If the **Ctrl** key is not pressed, moves the Circle from the [Children](#) collection of its parent panel to the [Children](#) collection of the panel that it was dropped on.
- Sets the [Effects](#) property to notify the [DoDragDrop](#) method whether a move or copy operation was performed.

6. Press **F5** to build and run the application.
7. Select the text `green` from the [TextBox](#).
8. Drag the text over a Circle control and drop it.
9. Drag a Circle control from the left panel to the right panel and drop it. The Circle is removed from the [Children](#) collection of the left panel and added to the [Children](#) collection of the right panel.
10. Drag a Circle control from the panel it is in to the other panel and drop it while pressing the **Ctrl** key. The Circle is copied and the copy is added to the [Children](#) collection of the receiving panel.



## See also

- [Drag and Drop Overview](#)

# Drag and Drop How-to Topics

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following examples demonstrate how to accomplish common tasks using the Windows Presentation Foundation (WPF) drag-and-drop framework.

## In This Section

[Open a File That is Dropped on a RichTextBox Control](#)

[Create a Data Object](#)

[Determine if a Data Format is Present in a Data Object](#)

[List the Data Formats in a Data Object](#)

[Retrieve Data in a Particular Data Format](#)

[Store Multiple Data Formats in a Data Object](#)

## See also

- [Drag and Drop Overview](#)

# How to: Open a File That is Dropped on a RichTextBox Control

8/27/2019 • 2 minutes to read • [Edit Online](#)

In Windows Presentation Foundation (WPF), the [TextBox](#), [RichTextBox](#), and [FlowDocument](#) controls all have built-in drag-and-drop functionality. The built-in functionality enables drag-and-drop of text within and between the controls. However, it does not enable opening a file by dropping the file on the control. These controls also mark the drag-and-drop events as handled. As a result, by default, you cannot add your own event handlers to provide functionality to open dropped files.

To add additional handling for drag-and-drop events in these controls, use the [AddHandler\(RoutedEvent, Delegate, Boolean\)](#) method to add your event handlers for the drag-and-drop events. Set the `handledEventsToo` parameter to `true` to have the specified handler be invoked for a routed event that has already been marked as handled by another element along the event route.

## TIP

You can replace the built-in drag-and-drop functionality of [TextBox](#), [RichTextBox](#), and [FlowDocument](#) by handling the preview versions of the drag-and-drop events and marking the preview events as handled. However, this will disable the built-in drag-and-drop functionality, and is not recommended.

## Example

The following example demonstrates how to add handlers for the [DragOver](#) and [Drop](#) events on a [RichTextBox](#). This example uses the [AddHandler\(RoutedEvent, Delegate, Boolean\)](#) method and sets the `handledEventsToo` parameter to `true` so that the events handlers will be invoked even though the [RichTextBox](#) marks these events as handled. The code in the event handlers adds functionality to open a text file that is dropped on the [RichTextBox](#).

To test this example, drag a text file or a rich text format (RTF) file from Windows Explorer to the [RichTextBox](#). The file will be opened in the [RichTextBox](#). If you press the SHIFT key before the dropping the file, the file will be opened as plain text.

```
<RichTextBox x:Name="richTextBox1"
    AllowDrop="True" />
```

```

public MainWindow()
{
    InitializeComponent();

    // Add using System.Windows.Controls;
    richTextBox1.AddHandler(RichTextBox.DragOverEvent, new DragEventHandler(RichTextBox_DragOver), true);
    richTextBox1.AddHandler(RichTextBox.DropEvent, new DragEventHandler(RichTextBox_Drop), true);
}

private void RichTextBox_DragOver(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        e.Effects = DragDropEffects.All;
    }
    else
    {
        e.Effects = DragDropEffects.None;
    }
    e.Handled = false;
}

private void RichTextBox_Drop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        string[] docPath = (string[])e.Data.GetData(DataFormats.FileDrop);

        // By default, open as Rich Text (RTF).
        var dataFormat = DataFormats.Rtf;

        // If the Shift key is pressed, open as plain text.
        if (e.KeyStates == DragDropKeyStates.ShiftKey)
        {
            dataFormat = DataFormats.Text;
        }

        System.Windows.Documents.TextRange range;
        System.IO.FileStream fStream;
        if (System.IO.File.Exists(docPath[0]))
        {
            try
            {
                // Open the document in the RichTextBox.
                range = new System.Windows.Documents.TextRange(richTextBox1.Document.ContentStart,
richTextBox1.Document.ContentEnd);
                fStream = new System.IO.FileStream(docPath[0], System.IO.FileMode.OpenOrCreate);
                range.Load(fStream, dataFormat);
                fStream.Close();
            }
            catch (System.Exception)
            {
                MessageBox.Show("File could not be opened. Make sure the file is a text file.");
            }
        }
    }
}
}

```

```

Public Sub New()
    InitializeComponent()

    richTextBox1.AddHandler(RichTextBox.DragOverEvent, New DragEventHandler(AddressOf RichTextBox_DragOver),
    True)
    richTextBox1.AddHandler(RichTextBox.DropEvent, New DragEventHandler(AddressOf RichTextBox_Drop), True)

End Sub

Private Sub RichTextBox_DragOver(sender As Object, e As DragEventArgs)
    If e.Data.GetDataPresent(DataFormats.FileDrop) Then
        e.Effects = DragDropEffects.All
    Else
        e.Effects = DragDropEffects.None
    End If
    e.Handled = False
End Sub

Private Sub RichTextBox_Drop(sender As Object, e As DragEventArgs)
    If e.Data.GetDataPresent(DataFormats.FileDrop) Then
        Dim docPath As String() = TryCast(e.Data.GetData(DataFormats.FileDrop), String())

        ' By default, open as Rich Text (RTF).
        Dim dataFormat = DataFormats.Rtf

        ' If the Shift key is pressed, open as plain text.
        If e.KeyStates = DragDropKeyStates.ShiftKey Then
            dataFormat = DataFormats.Text
        End If

        Dim range As TextRange
        Dim fStream As IO.FileStream
        If IO.File.Exists(docPath(0)) Then
            Try
                ' Open the document in the RichTextBox.
                range = New TextRange(richTextBox1.Document.ContentStart, richTextBox1.Document.ContentEnd)
                fStream = New IO.FileStream(docPath(0), IO FileMode.OpenOrCreate)
                range.Load(fStream, dataFormat)
                fStream.Close()
            Catch generatedExceptionName As System.Exception
                MessageBox.Show("File could not be opened. Make sure the file is a text file.")
            End Try
        End If
    End Sub

```

# How to: Create a Data Object

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following examples show various ways to create a data object using the constructors provided by the [DataObject](#) class.

## Example

### Description

The following example code creates a new data object and uses one of the overloaded constructors ([DataObject\(Object\)](#)) to initialize the data object with a string. In this case, an appropriate data format is determined automatically according to the stored data's type, and auto-converting of the stored data is allowed by default.

### Code

```
string stringData = "Some string data to store...";  
DataObject dataObject = new DataObject(stringData);
```

```
Dim stringData As String = "Some string data to store..."  
Dim dataObject As New DataObject(stringData)
```

### Description

The following example code is a condensed version of the code shown above.

### Code

```
DataObject dataObject = new DataObject("Some string data to store...");
```

```
Dim dataObject As New DataObject("Some string data to store...")
```

## Example

### Description

The following example code creates a new data object and uses one of the overloaded constructors ([DataObject\(String, Object\)](#)) to initialize the data object with a string and a specified data format. In this case the data format is specified by a string; the [DataFormats](#) class provides a set of pre-defined type strings. Auto-converting of the stored data is allowed by default.

### Code

```
string stringData = "Some string data to store...";  
string dataFormat = DataFormats.UnicodeText;  
DataObject dataObject = new DataObject(dataFormat, stringData);
```

```
Dim stringData As String = "Some string data to store..."  
Dim dataFormat As String = DataFormats.UnicodeText  
Dim dataObject As New DataObject(dataFormat, stringData)
```

## Description

The following example code is a condensed version of the code shown above.

## Code

```
DataObject dataObject = new DataObject(DataFormats.UnicodeText, "Some string data to store...");
```

```
Dim dataObject As New DataObject(DataFormats.UnicodeText, "Some string data to store...")
```

# Example

## Description

The following example code creates a new data object and uses one of the overloaded constructors ([DataObject](#)) to initialize the data object with a string and a specified data format. In this case the data format is specified by a [Type](#) parameter. Auto-converting of the stored data is allowed by default.

## Code

```
string stringData = "Some string data to store...";  
Type dataFormat = stringData.GetType();  
DataObject dataObject = new DataObject(dataFormat, stringData);
```

```
Dim stringData As String = "Some string data to store..."  
Dim dataFormat As Type = stringData.GetType()  
Dim dataObject As New DataObject(dataFormat, stringData)
```

## Description

The following example code is a condensed version of the code shown above.

## Code

```
DataObject dataObject = new DataObject("".GetType(), "Some string data to store...");
```

```
Dim dataObject As New DataObject("".GetType(), "Some string data to store...")
```

# Example

## Description

The following example code creates a new data object and uses one of the overloaded constructors ([DataObject\(String, Object, Boolean\)](#)) to initialize the data object with a string and a specified data format. In this case the data format is specified by a string; the [DataFormats](#) class provides a set of pre-defined type strings. This particular constructor overload enables the caller to specify whether auto-converting is allowed.

## Code

```
string stringData = "Some string data to store...";  
string dataFormat = DataFormats.Text;  
bool autoConvert = false;  
DataObject dataObject = new DataObject(dataFormat, stringData, autoConvert);
```

```
Dim stringData As String = "Some string data to store..."  
Dim dataFormat As String = DataFormats.Text  
Dim autoConvert As Boolean = False  
Dim dataObject As New DataObject(dataFormat, stringData, autoConvert)
```

## Description

The following example code is a condensed version of the code shown above.

## Code

```
DataObject dataObject = new DataObject(DataFormats.Text, "Some string data to store...", false);
```

```
Dim dataObject As New DataObject(DataFormats.Text, "Some string data to store...", False)
```

## See also

- [IDataObject](#)

# How to: Determine if a Data Format is Present in a Data Object

4/8/2019 • 4 minutes to read • [Edit Online](#)

The following examples show how to use the various [GetDataPresent](#) method overloads to query whether a particular data format is present in a data object.

## Example

### Description

The following example code uses the [GetDataPresent\(String\)](#) overload to query for the presence of a particular data format by descriptor string.

### Code

```
DataObject dataObject = new DataObject("Some string data to store...");

// Query for the presence of Text data in the data object, by a data format descriptor string.
// In this overload of GetDataPresent, the method will return true both for native data formats
// and when the data can automatically be converted to the specified format.

// In this case, string data is present natively, so GetDataPresent returns "true".
string textData = null;
if (dataObject.GetDataPresent(DataFormats.StringFormat))
{
    textData = dataObject.GetData(DataFormats.StringFormat) as string;
}

// In this case, the Text data in the data object can be autoconverted to
// Unicode text, so GetDataPresent returns "true".
byte[] unicodeData = null;
if (dataObject.GetDataPresent(DataFormats.UnicodeText))
{
    unicodeData = dataObject.GetData(DataFormats.UnicodeText) as byte[];
}
```

```
Dim dataObject As New DataObject("Some string data to store...")

' Query for the presence of Text data in the data object, by a data format descriptor string.
' In this overload of GetDataPresent, the method will return true both for native data formats
' and when the data can automatically be converted to the specified format.

' In this case, string data is present natively, so GetDataPresent returns "true".
Dim textData As String = Nothing
If dataObject.GetDataPresent(DataFormats.StringFormat) Then
    textData = TryCast(dataObject.GetData(DataFormats.StringFormat), String)
End If

' In this case, the Text data in the data object can be autoconverted to
' Unicode text, so GetDataPresent returns "true".
Dim unicodeData() As Byte = Nothing
If dataObject.GetDataPresent(DataFormats.UnicodeText) Then
    unicodeData = TryCast(dataObject.GetData(DataFormats.UnicodeText), Byte())
End If
```

## Example

### Description

The following example code uses the [GetDataPresent\(Type\)](#) overload to query for the presence of a particular data format by type.

### Code

```
DataObject dataObject = new DataObject("Some string data to store...");  
  
// Query for the presence of String data in the data object, by type. In this overload  
// of GetDataPresent, the method will return true both for native data formats  
// and when the data can automatically be converted to the specified format.  
  
// In this case, the Text data present in the data object can be autoconverted  
// to type string (also represented by DataFormats.String), so GetDataPresent returns "true".  
string stringData = null;  
if (dataObject.GetDataPresent(typeof(string)))  
{  
    stringData = dataObject.GetData(DataFormats.Text) as string;  
}
```

```
Dim dataObject As New DataObject("Some string data to store...")  
  
' Query for the presence of String data in the data object, by type. In this overload  
' of GetDataPresent, the method will return true both for native data formats  
' and when the data can automatically be converted to the specified format.  
  
' In this case, the Text data present in the data object can be autoconverted  
' to type string (also represented by DataFormats.String), so GetDataPresent returns "true".  
Dim stringData As String = Nothing  
If dataObject.GetDataPresent(GetType(String)) Then  
    stringData = TryCast(dataObject.GetData(DataFormats.Text), String)  
End If
```

## Example

### Description

The following example code uses the [GetDataPresent\(String, Boolean\)](#) overload to query for data by descriptor string, and specifying how to treat auto-convertible data formats.

### Code

```

DataObject dataObject = new DataObject("Some string data to store...");

// Query for the presence of Text data in the data object, by data format descriptor string,
// and specifying whether auto-convertible data formats are acceptable.

// In this case, Text data is present natively, so GetDataPresent returns "true".
string textData = null;
if (dataObject.GetDataPresent(DataFormats.Text, false /* Auto-convert? */))
{
    textData = dataObject.GetData(DataFormats.Text) as string;
}

// In this case, the Text data in the data object can be autoconverted to
// Unicode text, but it is not available natively, so GetDataPresent returns "false".
byte[] unicodeData = null;
if (dataObject.GetDataPresent(DataFormats.UnicodeText, false /* Auto-convert? */))
{
    unicodeData = dataObject.GetData(DataFormats.UnicodeText) as byte[];
}

// In this case, the Text data in the data object can be autoconverted to
// Unicode text, so GetDataPresent returns "true".
if (dataObject.GetDataPresent(DataFormats.UnicodeText, true /* Auto-convert? */))
{
    unicodeData = dataObject.GetData(DataFormats.UnicodeText) as byte[];
}

```

```

Dim dataObject As New DataObject("Some string data to store...")

' Query for the presence of Text data in the data object, by data format descriptor string,
' and specifying whether auto-convertible data formats are acceptable.

' In this case, Text data is present natively, so GetDataPresent returns "true".
Dim textData As String = Nothing
If dataObject.GetDataPresent(DataFormats.Text, False) Then ' Auto-convert?
    textData = TryCast(dataObject.GetData(DataFormats.Text), String)
End If

' In this case, the Text data in the data object can be autoconverted to
' Unicode text, but it is not available natively, so GetDataPresent returns "false".
Dim unicodeData() As Byte = Nothing
If dataObject.GetDataPresent(DataFormats.UnicodeText, False) Then ' Auto-convert?
    unicodeData = TryCast(dataObject.GetData(DataFormats.UnicodeText), Byte())
End If

' In this case, the Text data in the data object can be autoconverted to
' Unicode text, so GetDataPresent returns "true".
If dataObject.GetDataPresent(DataFormats.UnicodeText, True) Then ' Auto-convert?
    unicodeData = TryCast(dataObject.GetData(DataFormats.UnicodeText), Byte())
End If

```

## See also

- [IDataObject](#)

# How to: List the Data Formats in a Data Object

4/8/2019 • 3 minutes to read • [Edit Online](#)

The following examples show how to use the [GetFormats](#) method overloads get an array of strings denoting each data format that is available in a data object.

## Example

### Description

The following example code uses the [GetFormats](#) overload to get an array of strings denoting all data formats available in a data object (both native and auto-convertible).

### Code

```
DataObject dataObject = new DataObject("Some string data to store...");

// Get an array of strings, each string denoting a data format
// that is available in the data object. This overload of GetDataFormats
// returns all available data formats, native and auto-convertible.
string[] dataFormats = dataObject.GetFormats();

// Get the number of data formats present in the data object, including both
// auto-convertible and native data formats.
int numberOfDataFormats = dataFormats.Length;

// To enumerate the resulting array of data formats, and take some action when
// a particular data format is found, use a code structure similar to the following.
foreach (string dataFormat in dataFormats)
{
    if (dataFormat == DataFormats.Text)
    {
        // Take some action if/when data in the Text data format is found.
        break;
    }
    else if(dataFormat == DataFormats.StringFormat)
    {
        // Take some action if/when data in the string data format is found.
        break;
    }
}
```

```

Dim dataObject As New DataObject("Some string data to store...")

' Get an array of strings, each string denoting a data format
' that is available in the data object. This overload of GetDataFormats
' returns all available data formats, native and auto-convertible.
Dim dataFormats() As String = dataObject.GetFormats()

' Get the number of data formats present in the data object, including both
' auto-convertible and native data formats.
Dim numberOfDataFormats As Integer = dataFormats.Length

' To enumerate the resulting array of data formats, and take some action when
' a particular data format is found, use a code structure similar to the following.
For Each dataFormat As String In dataFormats
    If dataFormat = System.Windows.DataFormats.Text Then
        ' Take some action if/when data in the Text data format is found.
        Exit For
    ElseIf dataFormat = System.Windows.DataFormats.StringFormat Then
        ' Take some action if/when data in the string data format is found.
        Exit For
    End If
Next dataFormat

```

## Example

### Description

The following example code uses the [GetFormats](#) overload to get an array of strings denoting only data formats available in a data object (auto-convertible data formats are filtered).

### Code

```

DataObject dataObject = new DataObject("Some string data to store...");

// Get an array of strings, each string denoting a data format
// that is available in the data object. This overload of GetDataFormats
// accepts a Boolean parameter indicating whether to include auto-convertible
// data formats, or only return native data formats.
string[] dataFormats = dataObject.GetFormats(false /* Include auto-convertible? */);

// Get the number of native data formats present in the data object.
int numberOfDataFormats = dataFormats.Length;

// To enumerate the resulting array of data formats, and take some action when
// a particular data format is found, use a code structure similar to the following.
foreach (string dataFormat in dataFormats)
{
    if (dataFormat == DataFormats.Text)
    {
        // Take some action if/when data in the Text data format is found.
        break;
    }
}

```

```
Dim dataObject As New DataObject("Some string data to store...")

' Get an array of strings, each string denoting a data format
' that is available in the data object. This overload of GetDataFormats
' accepts a Boolean parameter indicating whether to include auto-convertible
' data formats, or only return native data formats.
Dim dataFormats() As String = dataObject.GetFormats(False) ' Include auto-convertible?

' Get the number of native data formats present in the data object.
Dim numberOfDataFormats As Integer = dataFormats.Length

' To enumerate the resulting array of data formats, and take some action when
' a particular data format is found, use a code structure similar to the following.
For Each dataFormat As String In dataFormats
    If dataFormat = System.Windows.DataFormats.Text Then
        ' Take some action if/when data in the Text data format is found.
        Exit For
    End If
Next dataFormat
```

## See also

- [IDataObject](#)
- [Drag and Drop Overview](#)

# How to: Retrieve Data in a Particular Data Format

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following examples show how to retrieve data from a data object in a specified format.

## Example

### Description

The following example code uses the [GetDataPresent\(String\)](#) overload to first check if a specified data format is available (natively or by auto-convert); if the specified format is available, the example retrieves the data by using the [GetData\(String\)](#) method.

### Code

```
DataObject dataObject = new DataObject("Some string data to store...");

string desiredFormat = DataFormats.UnicodeText;
byte[] data = null;

// Use the GetDataPresent method to check for the presence of a desired data format.
// This particular overload of GetDataPresent looks for both native and auto-convertible
// data formats.
if (dataObject.GetDataPresent(desiredFormat))
{
    // If the desired data format is present, use one of the GetData methods to retrieve the
    // data from the data object.
    data = dataObject.GetData(desiredFormat) as byte[];
}
```

```
Dim dataObject As New DataObject("Some string data to store...")

Dim desiredFormat As String = DataFormats.UnicodeText
Dim data() As Byte = Nothing

' Use the GetDataPresent method to check for the presence of a desired data format.
' This particular overload of GetDataPresent looks for both native and auto-convertible
' data formats.
If dataObject.GetDataPresent(desiredFormat) Then
    ' If the desired data format is present, use one of the GetData methods to retrieve the
    ' data from the data object.
    data = TryCast(dataObject.GetData(desiredFormat), Byte())
End If
```

## Example

### Description

The following example code uses the [GetDataPresent\(String, Boolean\)](#) overload to first check if a specified data format is available natively (auto-convertible data formats are filtered); if the specified format is available, the example retrieves the data by using the [GetData\(String\)](#) method.

### Code

```
DataObject dataObject = new DataObject("Some string data to store...");

string desiredFormat = DataFormats.UnicodeText;
bool noAutoConvert = false;
byte[] data = null;

// Use the GetDataPresent method to check for the presence of a desired data format.
// The autoconvert parameter is set to false to filter out auto-convertible data formats,
// returning true only if the specified data format is available natively.
if (dataObject.GetDataPresent(desiredFormat, noAutoConvert))
{
    // If the desired data format is present, use one of the GetData methods to retrieve the
    // data from the data object.
    data = dataObject.GetData(desiredFormat) as byte[];
}
```

```
Dim dataObject As New DataObject("Some string data to store...")

Dim desiredFormat As String = DataFormats.UnicodeText
Dim noAutoConvert As Boolean = False
Dim data() As Byte = Nothing

' Use the GetDataPresent method to check for the presence of a desired data format.
' The autoconvert parameter is set to false to filter out auto-convertible data formats,
' returning true only if the specified data format is available natively.
If dataObject.GetDataPresent(desiredFormat, noAutoConvert) Then
    ' If the desired data format is present, use one of the GetData methods to retrieve the
    ' data from the data object.
    data = TryCast(dataObject.GetData(desiredFormat), Byte())
End If
```

## See also

- [IDataObject](#)
- [Drag and Drop Overview](#)

# How to: Store Multiple Data Formats in a Data Object

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to use the [SetData\(String, Object\)](#) method to add data to a data object in multiple formats.

## Example

### Description

### Code

```
DataObject dataObject = new DataObject();
string sourceData = "Some string data to store...";

// Encode the source string into Unicode byte arrays.
byte[] unicodeText = Encoding.Unicode.GetBytes(sourceData); // UTF-16
byte[] utf8Text = Encoding.UTF8.GetBytes(sourceData);
byte[] utf32Text = Encoding.UTF32.GetBytes(sourceData);

// The DataFormats class does not provide data format fields for denoting
// UTF-32 and UTF-8, which are seldom used in practice; the following strings
// will be used to identify these "custom" data formats.
string utf32DateFormat = "UTF-32";
string utf8DateFormat = "UTF-8";

// Store the text in the data object, letting the data object choose
// the data format (which will be DataFormats.Text in this case).
dataObject.SetData(sourceData);
// Store the Unicode text in the data object. Text data can be automatically
// converted to Unicode (UTF-16 / UCS-2) format on extraction from the data object;
// Therefore, explicitly converting the source text to Unicode is generally unnecessary, and
// is done here as an exercise only.
dataObject.SetData(DataFormats.UnicodeText, unicodeText);
// Store the UTF-8 text in the data object...
dataObject.SetData(utf8DateFormat, utf8Text);
// Store the UTF-32 text in the data object...
dataObject.SetData(utf32DateFormat, utf32Text);
```

```
Dim dataObject As New DataObject()
Dim sourceData As String = "Some string data to store..."

' Encode the source string into Unicode byte arrays.
Dim unicodeText() As Byte = Encoding.Unicode.GetBytes(sourceData) ' UTF-16
Dim utf8Text() As Byte = Encoding.UTF8.GetBytes(sourceData)
Dim utf32Text() As Byte = Encoding.UTF32.GetBytes(sourceData)

' The DataFormats class does not provide data format fields for denoting
' UTF-32 and UTF-8, which are seldom used in practice; the following strings
' will be used to identify these "custom" data formats.
Dim utf32DataFormat As String = "UTF-32"
Dim utf8DataFormat As String = "UTF-8"

' Store the text in the data object, letting the data object choose
' the data format (which will be DataFormats.Text in this case).
dataObject.SetData(sourceData)

' Store the Unicode text in the data object. Text data can be automatically
' converted to Unicode (UTF-16 / UCS-2) format on extraction from the data object;
' Therefore, explicitly converting the source text to Unicode is generally unnecessary, and
' is done here as an exercise only.
dataObject.SetData(DataFormats.UnicodeText, unicodeText)
' Store the UTF-8 text in the data object...
dataObject.SetData(utf8DataFormat, utf8Text)
' Store the UTF-32 text in the data object...
dataObject.SetData(utf32DataFormat, utf32Text)
```

## See also

- [IDataObject](#)
- [Drag and Drop Overview](#)

# Resources (WPF)

11/3/2019 • 2 minutes to read • [Edit Online](#)

A resource is an object that can be reused in different places in your application. WPF supports different types of resources. These resources are primarily two types of resources: XAML resources and resource data files. Examples of XAML resources include brushes and styles. Resource data files are non-executable data files that an application needs.

## In This Section

[XAML Resources](#)

[WPF Application Resource, Content, and Data Files](#)

[Pack URIs in WPF](#)

## Reference

[ResourceDictionary](#)

[StaticResource Markup Extension](#)

[DynamicResource Markup Extension](#)

[x:Key Directive](#)

## Related Sections

[XAML in WPF](#)

2 minutes to read

# Resources and Code

11/3/2019 • 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 Extensible Application Markup Language (XAML) syntax. For more information on general resource usage and resources from a XAML syntax perspective, see [XAML Resources](#).

## Accessing Resources from Code

The keys that identify resources if they are defined through XAML 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 is not found. [FindResource](#) raises an exception; [TryFindResource](#) will not 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; see the [Using Objects as Keys](#) section for details. Typically you would cast the returned object to the type required by the property that you are setting when requesting the resource. 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 to successive 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 runtime changes in resource dictionaries that might have been made subsequent to that resource dictionary being loaded from XAML, and also for realtime system resource changes.

The following is a brief code example that finds a resource by key and uses the returned value to set a property, implemented as a [Click](#) event handler.

```
void SetBGByResource(object sender, RoutedEventArgs e)
{
    Button b = sender as Button;
    b.Background = (Brush)this.FindResource("RainbowBrush");
}
```

```
Private Sub SetBGByResource(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim b As Button = TryCast(sender, Button)
    b.Background = CType(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 is 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. Accessing the dictionary contained by this property is also how you can add new resources to existing collections, check to see if a given key name is already taken in the collection, and other dictionary/collection operations. If you are writing a WPF application entirely in code, you can also create the entire collection in code, assign keys to it, and then assign the finished collection to the [Resources](#) property of an established element. This will be described in the next section.

You can index within any given [Resources](#) collection, using a specific key as the index, but you should be aware

that accessing the resource in this way does not follow the normal runtime rules of resource resolution. You are only accessing that particular collection. Resource lookup will not be traversing the 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. See the [ResourceDictionary](#) class for more details on how to work with the resource dictionary directly.

## 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, use the [ResourceDictionary](#) thus created to set the [Resources](#) property on an element that is 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 is not attached to an element `Resources` property would not 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 do not use a string type to specify keys, instead this parameter is an object. The capability of having the resource be keyed by an object is used by the WPF style and theming support. The styles in themes which become the default style 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, and 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](#).

## See also

- [XAML Resources](#)
- [Styling and Templating](#)

# Merged Resource Dictionaries

11/3/2019 • 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.

## Introducing a Merged Resource 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>
```

Note that the [ResourceDictionary](#) element does not 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. The [ResourceDictionary](#) that introduces a merged resource dictionary cannot 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 is legal to define resources within a [ResourceDictionary](#) that is 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 is not 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, you should typically 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 is 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 is 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 is 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 is contained in [MergedDictionaries](#). The object you add must be a new [ResourceDictionary](#). In code, you do not 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](#) to call [XamlReader.Load](#) on an existing XAML file stream that has a [ResourceDictionary](#) root, then casting the [XamlReader.Load](#) return value to [ResourceDictionary](#).

## Merged Resource 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 will use. Broadly speaking, these techniques can be divided into two categories: resources that are compiled as part of the project, and resources that are not 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 do not 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

Do not use the Embedded Resource build action. The build action itself is supported for WPF applications, but the resolution of [Source](#) does not 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 are not 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 noncompiled 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 will depend on your application deployment strategy and which application model you follow. The aforementioned 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 is 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 / 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](#)
- [XAML Resources](#)
- [Resources and Code](#)
- [WPF Application Resource, Content, and Data Files](#)

# Resources How-to Topics

11/3/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to use Windows Presentation Foundation (WPF) resources.

## In This Section

[Define and Reference a Resource](#)

[Use Application Resources](#)

[Use SystemFonts](#)

[Use System Fonts Keys](#)

[Use SystemParameters](#)

[Use System Parameters Keys](#)

## Reference

[Resources](#)

[SystemColors](#)

[SystemParameters](#)

[SystemFonts](#)

## Related Sections

[XAML Resources](#)

# How to: Define and Reference a Resource

11/3/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to define a resource and reference it by using an attribute in Extensible Application Markup Language (XAML).

## Example

The following example defines two types of resources: a `SolidColorBrush` resource, and several `Style` resources.

The `SolidColorBrush` resource `MyBrush` is used to provide the value of several properties that each take a `Brush` type value. The `Style` resources `PageBackground`, `TitleText` and `Label` each target a particular control type. The styles set a variety of different properties on the targeted controls, when that style resource is referenced by resource key and is used to set the `Style` property of several specific control elements defined in XAML.

Note that one of the properties within the setters of the `Label` style also references the `MyBrush` resource defined earlier. This is a common technique, but it is important to remember that resources are parsed and entered into a resource dictionary in the order that they are given. 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 earlier within the resources collection than where that resource is then requested. 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 details, see [XAML Resources](#).

```

<Page Name="root"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Page.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
        <Style TargetType="Border" x:Key="PageBackground">
            <Setter Property="Background" Value="Blue"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="TitleText">
            <Setter Property="Background" Value="Blue"/>
            <Setter Property="DockPanel.Dock" Value="Top"/>
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="Foreground" Value="#4E87D4"/>
            <Setter Property="FontFamily" Value="Trebuchet MS"/>
            <Setter Property="Margin" Value="0,40,10,10"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="Label">
            <Setter Property="DockPanel.Dock" Value="Right"/>
            <Setter Property="FontSize" Value="8"/>
            <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>
    </Page.Resources>
    <StackPanel>
        <Border Style="{StaticResource PageBackground}">
            <DockPanel>
                <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
                <TextBlock Style="{StaticResource Label}">Label</TextBlock>
                <TextBlock DockPanel.Dock="Top" HorizontalAlignment="Left" FontSize="36" Foreground="{StaticResource MyBrush}" Text="Text" Margin="20" />
                <Button DockPanel.Dock="Top" HorizontalAlignment="Left" Height="30" Background="{StaticResource MyBrush}" Margin="40">Button</Button>
                <Ellipse DockPanel.Dock="Top" HorizontalAlignment="Left" Width="100" Height="100" Fill="{StaticResource MyBrush}" Margin="40" />
            </DockPanel>
        </Border>
    </StackPanel>
</Page>

```

## See also

- [XAML Resources](#)
- [Styling and Templating](#)

# How to: Use Application Resources

11/3/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use application resources.

## 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="Button" x:Key="GelButton" >
        <Setter Property="Margin" Value="1,2,1,2"/>
        <Setter Property="HorizontalAlignment" Value="Left"/>
        <Setter Property="Template">
            <Setter.Value>
                ...
            </Setter.Value>
        </Setter>
    </Style>
</Application.Resources>
```

The following example shows a XAML page that references the application-level resource that the previous example defined. The resource is referenced by using a [StaticResource Markup Extension](#) that specifies the unique resource key for the requested resource. No resource with key of "GelButton" is found in the current page, so the resource lookup scope for the requested resource continues beyond the current page and into the defined application-level resources.

```
<StackPanel
    Name="root"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <Button Height="50" Width="250" Style="{StaticResource GelButton}" Content="Button 1" />
    <Button Height="50" Width="250" Style="{StaticResource GelButton}" Content="Button 2" />
</StackPanel>
```

## See also

- [XAML Resources](#)
- [Application Management Overview](#)
- [How-to Topics](#)

# How to: Use SystemFonts

11/3/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use the static resources of the [SystemFonts](#) class in order to style or customize a button.

## Example

System resources expose several system-determined values as both resources and properties in order to help you create visuals that are consistent with system settings. [SystemFonts](#) is a class that contains both 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.

In XAML, you can use the members of [SystemFonts](#) as either static properties or dynamic resource references (with the static property value as the key). Use a dynamic resource reference if you want the font metric to automatically update while the application runs; otherwise, use a static value reference.

### NOTE

The resource keys have the suffix "Key" appended to the property name.

The following example shows how to access and use the properties of [SystemFonts](#) as static values in order to style or customize a button. This markup example assigns [SystemFonts](#) values to a button.

```
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="3"
    FontSize="{x:Static SystemFonts.IconFontSize}"
    FontWeight="{x:Static SystemFonts.MessageFontWeight}"
    FontFamily="{x:Static SystemFonts.CaptionFontFamily}">
    SystemFonts
</Button>
```

To use the values of [SystemFonts](#) in code, you do not 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.

```
Button btncsharp = new Button();
btncsharp.Content = "SystemFonts";
btncsharp.Background = SystemColors.ControlDarkDarkBrush;
btncsharp.FontSize = SystemFonts.IconFontSize;
btncsharp.FontWeight = SystemFonts.MessageFontWeight;
btncsharp.FontFamily = SystemFonts.CaptionFontFamily;
cv1.Children.Add(btncsharp);
```

```
Dim btn As New Button()
btn.Content = "SystemFonts"
btn.Background = SystemColors.ControlDarkDarkBrush
btn.FontSize = SystemFonts.IconFontSize
btn.FontWeight = SystemFonts.MessageFontWeight
btn.FontFamily = SystemFonts.CaptionFontFamily
cv1.Children.Add(btn)
```

## See also

- [SystemFonts](#)
- [Paint an Area with a System Brush](#)
- [Use SystemParameters](#)
- [Use System Fonts Keys](#)
- [How-to Topics](#)
- [x:Static Markup Extension](#)
- [XAML Resources](#)
- [DynamicResource Markup Extension](#)

# How to: Use System Fonts Keys

8/22/2019 • 2 minutes to read • [Edit Online](#)

System resources expose a number of system metrics as resources to help developers create visuals that are consistent with system settings. [SystemFonts](#) is a class that contains both system font values and system font resources that bind to the values—for example, [CaptionFontFamily](#) and [CaptionFontFamilyKey](#).

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 button. This XAML example creates a button style that assigns [SystemFonts](#) values to a button.

## Example

```
<Style x:Key="SimpleFont" TargetType="{x:Type Button}">
    <Setter Property = "FontSize" Value= "{DynamicResource {x:Static SystemFonts.IconFontSizeKey}}"/>
    <Setter Property = "FontWeight" Value= "{DynamicResource {x:Static SystemFonts.MessageFontWeightKey}}"/>
    <Setter Property = "FontFamily" Value= "{DynamicResource {x:Static SystemFonts.CaptionFontFamilyKey}}"/>
</Style>
```

## See also

- [Paint an Area with a System Brush](#)
- [Use SystemParameters](#)
- [Use SystemFonts](#)

# How to: Use SystemParameters

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to access and use the properties of [SystemParameters](#) in order to style or customize a button.

## Example

System resources expose several system based settings as resources in order to help you create visuals that are consistent with system settings. [SystemParameters](#) is a class that 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.

In XAML, you can use the members of [SystemParameters](#) as either a static property usage, or a dynamic resource references (with the static property value as the key). Use a dynamic resource reference if you want the system based value to update automatically while the application runs; otherwise, use a static reference. Resource keys have the suffix `key` appended to the property name.

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" Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="5"
    HorizontalAlignment="Left"
    Height="{x:Static SystemParameters.CaptionHeight}"
    Width="{x:Static SystemParameters.IconGridWidth}>
    SystemParameters
</Button>
```

To use the values of [SystemParameters](#) in code, you do not 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 runtime behavior of WPF as hosted by the system will reevaluate the properties in realtime, 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.

```
Button btncsharp = new Button();
btncsharp.Content = "SystemParameters";
btncsharp.FontSize = 8;
btncsharp.Background = SystemColors.ControlDarkDarkBrush;
btncsharp.Height = SystemParameters.CaptionHeight;
btncsharp.Width = SystemParameters.IconGridWidth;
cv2.Children.Add(btncsharp);
```

```
Dim btn As New Button()
btn.Content = "SystemParameters"
btn.FontSize = 8
btn.Background = SystemColors.ControlDarkDarkBrush
btn.Height = SystemParameters.CaptionHeight
btn.Width = SystemParameters.IconGridWidth
cv2.Children.Add(btn)
```

## See also

- [SystemParameters](#)
- [Paint an Area with a System Brush](#)
- [Use SystemFonts](#)
- [Use System Parameters Keys](#)
- [How-to Topics](#)

# How to: Use System Parameters Keys

8/22/2019 • 2 minutes to read • [Edit Online](#)

System resources expose a number of system metrics as resources to help developers create visuals that are consistent with system settings. [SystemParameters](#) is a class that contains both system parameter values and resource keys that bind to the values—for example, [FullPrimaryScreenHeight](#) and [FullPrimaryScreenHeightKey](#). 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.

## Example

```
<Style x:Key="SimpleParam" TargetType="{x:Type Button}">
    <Setter Property = "Height" Value= "{DynamicResource {x:Static SystemParameters.CaptionHeightKey}}"/>
    <Setter Property = "Width" Value= "{DynamicResource {x:Static SystemParameters.IconGridWidthKey}}"/>
</Style>
```

## See also

- [Paint an Area with a System Brush](#)
- [Use SystemFonts](#)
- [Use SystemParameters](#)

# Documents

4/8/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a versatile set of components that enable developers to build applications with advanced document features and an improved reading experience. In addition to enhanced capabilities and quality, Windows Presentation Foundation (WPF) also provides simplified management services for document packaging, security, and storage.

## In This Section

[Documents in WPF](#)

[Document Serialization and Storage](#)

[Annotations](#)

[Flow Content](#)

[Typography](#)

[Printing and Print System Management](#)

## See also

- [DocumentViewer](#)
- [FlowDocument](#)
- [System.Windows.Xps](#)
- [isXPS.exe \(isXPS Conformance Tool\)](#)

# Documents in WPF

11/12/2019 • 9 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) offers a wide range of document features that enable the creation of high-fidelity content that is designed to be more easily accessed and read than in previous generations of Windows. In addition to enhanced capabilities and quality, WPF also provides integrated services for document display, packaging, and security. This topic provides an introduction to WPF document types and document packaging.

## Types of Documents

WPF divides documents into two broad categories based on their intended use; these document categories are termed "fixed documents" and "flow documents."

Fixed documents are intended for applications that require a precise "what you see is what you get" (WYSIWYG) presentation, independent of the display or printer hardware used. Typical uses for fixed documents include desktop publishing, word processing, and form layout, where adherence to the original page design is critical. As part of its layout, a fixed document maintains the precise positional placement of content elements independent of the display or print device in use. For example, a fixed document page viewed on 96 dpi display will appear exactly the same when it is output to a 600 dpi laser printer as when it is output to a 4800 dpi phototypesetter. The page layout remains the same in all cases, while the document quality maximizes to the capabilities of each device.

By comparison, flow documents are designed to optimize viewing and readability and are best utilized when ease of reading is the primary document consumption scenario. Rather than being set to one predefined layout, flow documents dynamically adjust and reflow their content based on run-time variables such as window size, device resolution, and optional user preferences. A Web page is a simple example of a flow document where the page content is dynamically formatted to fit the current window. Flow documents optimize the viewing and reading experience for the user, based on the runtime environment. For example, the same flow document will dynamically reformat for optimal readability on either high-resolution 19-inch display or a small 2x3-inch PDA screen. In addition, flow documents have a number of built in features including search, viewing modes that optimize readability, and the ability to change the size and appearance of fonts. See [Flow Document Overview](#) for illustrations, examples, and in-depth information on flow documents.

## Document Controls and Text Layout

The .NET Framework provides a set of pre-built controls that simplify using fixed documents, flow documents, and general text within your application. The display of fixed document content is supported using the [DocumentViewer](#) control. Display of flow document content is supported by three different controls: [FlowDocumentReader](#), [FlowDocumentPageViewer](#), and [FlowDocumentScrollView](#) which map to different user scenarios (see sections below). Other WPF controls provide simplified layout to support general text uses (see [Text in the User Interface](#), below).

### Fixed Document Control - DocumentViewer

The [DocumentViewer](#) control is designed to display [FixedDocument](#) content. The [DocumentViewer](#) control provides an intuitive user interface that provides built-in support for common operations including print output, copy to clipboard, zoom, and text search features. The control provides access to pages of content through a familiar scrolling mechanism. Like all WPF controls, [DocumentViewer](#) supports complete or partial restyling, which enables the control to be visually integrated into virtually any application or environment.

[DocumentViewer](#) is designed to display content in a read-only manner; editing or modification of content is not available and is not supported.

## Flow Document Controls

### NOTE

For more detailed information on flow document features and how to create them, see [Flow Document Overview](#).

Display of flow document content is supported by three controls: [FlowDocumentReader](#), [FlowDocumentPageViewer](#), and [FlowDocumentScrollView](#).

### FlowDocumentReader

[FlowDocumentReader](#) includes features that enable the user to dynamically choose between various viewing modes, including a single-page (page-at-a-time) viewing mode, a two-page-at-a-time (book reading format) viewing mode, and a continuous scrolling (bottomless) viewing mode. For more information about these viewing modes, see [FlowDocumentReaderViewingMode](#). If you do not need the ability to dynamically switch between different viewing modes, [FlowDocumentPageViewer](#) and [FlowDocumentScrollView](#) provide lighter-weight flow content viewers that are fixed in a particular viewing mode.

### FlowDocumentPageViewer and FlowDocumentScrollView

[FlowDocumentPageViewer](#) shows content in page-at-a-time viewing mode, while [FlowDocumentScrollView](#) shows content in continuous scrolling mode. Both [FlowDocumentPageViewer](#) and [FlowDocumentScrollView](#) are fixed to a particular viewing mode. Compare to [FlowDocumentReader](#), which includes features that enable the user to dynamically choose between various viewing modes (as provided by the [FlowDocumentReaderViewingMode](#) enumeration), at the cost of being more resource intensive than [FlowDocumentPageViewer](#) or [FlowDocumentScrollView](#).

By default, a vertical scrollbar is always shown, and a horizontal scrollbar becomes visible if needed. The default UI for [FlowDocumentScrollView](#) does not include a toolbar; however, the [IsToolBarVisible](#) property can be used to enable a built-in toolbar.

## Text in the User Interface

Besides adding text to documents, text can obviously be used in application UI such as forms. WPF includes multiple controls for drawing text to the screen. Each control is targeted to a different scenario and has its own list of features and limitations. In general, the [TextBlock](#) element should be used when limited text support is required, such as a brief sentence in a user interface (UI). [Label](#) can be used when minimal text support is required. For more information, see [TextBlock Overview](#).

## Document Packaging

The [System.IO.Packaging](#) APIs provide an efficient means to organize application data, document content, and related resources in a single container that is simple to access, portable, and easy to distribute. A ZIP file is an example of a [Package](#) type capable of holding multiple objects as a single unit. The packaging APIs provide a default [ZipPackage](#) implementation designed using an Open Packaging Conventions standard with XML and ZIP file architecture. The WPF packaging APIs make it simple to create packages, and to store and access objects within them. An object stored in a [Package](#) is referred to as a [PackagePart](#) ("part"). Packages can also include signed digital certificates that can be used to identify the originator of a part and to validate that the contents of a package have not been modified. Packages also include a [PackageRelationship](#) feature that allows additional information to be added to a package or associated with specific parts without actually modifying the content of existing parts. Package services also support Microsoft Windows Rights Management (RM).

The WPF Package architecture serves as the foundation for a number of key technologies:

- XPS documents conforming to the XML Paper Specification (XPS).
- Microsoft Office "12" open XML format documents (.docx).
- Custom storage formats for your own application design.

Based on the packaging APIs, an [XpsDocument](#) is specifically designed for storing WPF fixed content documents. An [XpsDocument](#) is a self-contained document that can be opened in a viewer, displayed in a [DocumentViewer](#) control, routed to a print spool, or output directly to an XPS-compatible printer.

The following sections provide additional information on the [Package](#) and [XpsDocument](#) APIs provided with WPF.

### **Package Components**

The WPF packaging APIs allow application data and documents to be organized into a single portable unit. A ZIP file is one of the most common types of packages and is the default package type provided with WPF. [Package](#) itself is an abstract class from which [ZipPackage](#) is implemented using an open standard XML and ZIP file architecture. The [Open](#) method uses [ZipPackage](#) to create and use ZIP files by default. A package can contain three basic types of items:

<a href="#">PackagePart</a>	Application content, data, documents, and resource files.
<a href="#">PackageDigitalSignature</a>	[X.509 Certificate] for identification, authentication and validation.
<a href="#">PackageRelationship</a>	Added information related to the package or a specific part.

#### **PackageParts**

A [PackagePart](#) ("part") is an abstract class that refers to an object stored in a [Package](#). In a ZIP file, the package parts correspond to the individual files stored within the ZIP file. [ZipPackagePart](#) provides the default implementation for serializable objects stored in a [ZipPackage](#). Like a file system, parts contained in the package are stored in hierarchical directory or "folder-style" organization. Using the WPF packaging APIs, applications can write, store, and read multiple [PackagePart](#) objects using a single ZIP file container.

#### **PackageDigitalSignatures**

For security, a [PackageDigitalSignature](#) ("digital signature") can be associated with parts within a package. A [PackageDigitalSignature](#) incorporates a [509] that provides two features:

1. Identifies and authenticates the originator of the part.
2. Validates that the part has not been modified.

The digital signature does not preclude a part from being modified, but a validation check against the digital signature will fail if the part is altered in any way. The application can then take appropriate action—for example, block opening the part or notify the user that the part has been modified and is not secure.

#### **PackageRelationships**

A [PackageRelationship](#) ("relationship") provides a mechanism for associating additional information with the package or a part within the package. A relationship is a package-level facility that can associate additional information with a part without modifying the actual part content. Inserting new data directly into the part content of is usually not practical in many cases:

- The actual type of the part and its content schema is not known.
- Even if known, the content schema might not provide a means for adding new information.
- The part might be digitally signed or encrypted, precluding any modification.

Package relationships provide a discoverable means for adding and associating additional information with individual parts or with the entire package. Package relationships are used for two primary functions:

1. Defining dependency relationships from one part to another part.
2. Defining information relationships that add notes or other data related to the part.

A [PackageRelationship](#) provides a quick, discoverable means to define dependencies and add other information associated with a part of the package or the package as a whole.

#### **Dependency Relationships**

Dependency relationships are used to describe dependencies that one part makes to other parts. For example, a package might contain an HTML part that includes one or more <img> image tags. The image tags refer to images that are located either as other parts internal to the package or external to the package (such as accessible over the Internet). Creating a [PackageRelationship](#) associated with HTML file makes discovering and accessing the dependent resources quick and easy. A browser or viewer application can directly access the part relationships and immediately begin assembling the dependent resources without knowing the schema or parsing the document.

#### **Information Relationships**

Similar to a note or annotation, a [PackageRelationship](#) can also be used to store other types of information to be associated with a part without having to actually modify the part content itself.

## XPS Documents

XML Paper Specification (XPS) document is a package that contains one or more fixed-documents along with all the resources and information required for rendering. XPS is also the native Windows Vista print spool file format. An [XpsDocument](#) is stored in standard ZIP dataset, and can include a combination of XML and binary components, such as image and font files. [PackageRelationships](#) are used to define the dependencies between the content and the resources required to fully render the document. The [XpsDocument](#) design provides a single, high-fidelity document solution that supports multiple uses:

- Reading, writing, and storing fixed-document content and resources as a single, portable, and easy-to-distribute file.
- Displaying documents with the XPS Viewer application.
- Outputting documents in the native print spool output format of Windows Vista.
- Routing documents directly to an XPS-compatible printer.

## See also

- [FixedDocument](#)
- [FlowDocument](#)
- [XpsDocument](#)
- [ZipPackage](#)
- [ZipPackagePart](#)
- [PackageRelationship](#)
- [DocumentViewer](#)
- [Text](#)
- [Flow Document Overview](#)
- [Printing Overview](#)
- [Document Serialization and Storage](#)

# Document Serialization and Storage

11/12/2019 • 6 minutes to read • [Edit Online](#)

Microsoft .NET Framework provides a powerful environment for creating and displaying high quality documents. Enhanced features that support both fixed-documents and flow-documents, advanced viewing controls, combined with powerful 2D and 3D graphic capabilities take .NET Framework applications to a new level of quality and user experience. Being able to flexibly manage an in-memory representation of a document is a key feature of .NET Framework, and being able to efficiently save and load documents from a data store is a need of almost every application. The process of converting a document from an internal in-memory representation to an external data store is termed serialization. The reverse process of reading a data store and recreating the original in-memory instance is termed deserialization.

## About Document Serialization

Ideally the process of serializing and deserializing a document from and then back into memory is transparent to the application. The application calls a serializer "write" method to save the document, while a deserializer "read" method accesses the data store and recreates the original instance in memory. The specific format that the data is stored in is generally not a concern of the application as long as the serialize and deserialize process recreates the document back to its original form.

Applications often provide multiple serialization options which allow the user to save documents to different medium or to a different format. For example, an application might offer "Save As" options to store a document to a disk file, database, or web service. Similarly, different serializers could store the document in different formats such as in HTML, RTF, XML, XPS, or alternately to a third-party format. To the application, serialization defines an interface that isolates the details of the storage medium within the implementation of each specific serializer. In addition to the benefits of encapsulating storage details, the .NET Framework [System.Windows.Documents.Serialization](#) APIs provide several other important features.

### Features of .NET Framework 3.0 Document Serializers

- Direct access to the high-level document objects (logical tree and visuals) enable efficient storage of paginated content, 2D/3D elements, images, media, hyperlinks, annotations, and other support content.
- Synchronous and asynchronous operation.
- Support for plug-in serializers with enhanced capabilities:
  - System-wide access for use by all .NET Framework applications.
  - Simple application plug-in discoverability.
  - Simple deployment, installation, and update for custom third-party plug-ins.
  - User interface support for custom run-time settings and options.

### XPS Print Path

The Microsoft .NET Framework XPS print path also provides an extensible mechanism for writing documents through print output. XPS serves as both a document file format and is the native print spool format for Windows Vista. XPS documents can be sent directly to XPS-compatible printers without the need for conversion to an intermediate format. See the [Printing Overview](#) for additional information on print path output options and capabilities.

## Plug-in Serializers

The [System.Windows.Documents.Serialization](#) APIs provide support for both plug-in serializers and linked serializers that are installed separately from the application, bind at run time, and are accessed by using the [SerializerProvider](#) discovery mechanism. Plug-in serializers offer enhanced benefits for ease of deployment and system-wide use. Linked serializers can also be implemented for partial trust environments such as XAML browser applications (XBAPs) where plug-in serializers are not accessible. Linked serializers, which are based on a derived implementation of the [SerializerWriter](#) class, are compiled and linked directly into the application. Both plug-in serializers and linked serializers operate through identical public methods and events which make it easy to use either or both types of serializers in the same application.

Plug-in serializers aid application developers by providing extensibility to new storage designs and file formats without having to code directly for every potential format at build time. Plug-in serializers also benefit third-party developers by providing a standardized means to deploy, install, and update system accessible plug-ins for custom or proprietary file formats.

### Using a Plug-in Serializer

Plug-in serializers are simple to use. The [SerializerProvider](#) class enumerates a [SerializerDescriptor](#) object for each plug-in installed on the system. The [IsLoadable](#) property filters the installed plug-ins based on the current configuration and verifies that the serializer can be loaded and used by the application. The [SerializerDescriptor](#) also provides other properties, such as [DisplayName](#) and [DefaultFileExtension](#), which the application can use to prompt the user in selecting a serializer for an available output format. A default plug-in serializer for XPS is provided with .NET Framework and is always enumerated. After the user selects an output format, the [CreateSerializerWriter](#) method is used to create a [SerializerWriter](#) for the specific format. The [SerializerWriter.Write](#) method can then be called to output the document stream to the data store.

The following example illustrates an application that uses the [SerializerProvider](#) method in a "PlugInFileFilter" property. PlugInFileFilter enumerates the installed plug-ins and builds a filter string with the available file options for a [SaveFileDialog](#).

```

// ----- PlugInFileFilter -----
/// <summary>
/// Gets a filter string for installed plug-in serializers.</summary>
/// <remark>
/// PlugInFileFilter is used to set the SaveFileDialog or
/// OpenFileDialog "Filter" property when saving or opening files
/// using plug-in serializers.</remark>
private string PlugInFileFilter
{
    get
    {
        // Create a SerializerProvider for accessing plug-in serializers.
        SerializerProvider serializerProvider = new SerializerProvider();
        string filter = "";

        // For each loadable serializer, add its display
        // name and extension to the filter string.
        foreach (SerializerDescriptor serializerDescriptor in
            serializerProvider.InstalledSerializers)
        {
            if (serializerDescriptor.IsLoadable)
            {
                // After the first, separate entries with a "|".
                if (filter.Length > 0)    filter += "|";

                // Add an entry with the plug-in name and extension.
                filter += serializerDescriptor.DisplayName + "(*" +
                    serializerDescriptor.DefaultFileExtension + ")|*" +
                    serializerDescriptor.DefaultFileExtension;
            }
        }

        // Return the filter string of installed plug-in serializers.
        return filter;
    }
}

```

After an output file name has been selected by the user, the following example illustrates use of the [CreateSerializerWriter](#) method to store a given document in a specified format.

```

// Create a SerializerProvider for accessing plug-in serializers.
SerializerProvider serializerProvider = new SerializerProvider();

// Locate the serializer that matches the fileName extension.
SerializerDescriptor selectedPlugIn = null;
foreach ( SerializerDescriptor serializerDescriptor in
    serializerProvider.InstalledSerializers )
{
    if ( serializerDescriptor.IsLoadable &&
        fileName.EndsWith(serializerDescriptor.DefaultFileExtension) )
    { // The plug-in serializer and fileName extensions match.
        selectedPlugIn = serializerDescriptor;
        break; // foreach
    }
}

// If a match for a plug-in serializer was found,
// use it to output and store the document.
if (selectedPlugIn != null)
{
    Stream package = File.Create(fileName);
    SerializerWriter serializerWriter =
        serializerProvider.CreateSerializerWriter(selectedPlugIn,
                                                package);
    IDocumentPaginatorSource idoc =
        flowDocument as IDocumentPaginatorSource;
    serializerWriter.Write(idoc.DocumentPaginator, null);
    package.Close();
    return true;
}

```

## Installing Plug-in Serializers

The [SerializerProvider](#) class supplies the upper-level application interface for plug-in serializer discovery and access. [SerializerProvider](#) locates and provides the application a list of the serializers that are installed and accessible on the system. The specifics of the installed serializers are defined through registry settings. Plug-in serializers can be added to the registry by using the [RegisterSerializer](#) method; or if .NET Framework is not yet installed, the plug-in installation script can directly set the registry values itself. The [UnregisterSerializer](#) method can be used to remove a previously installed plug-in, or the registry settings can be reset similarly by an uninstall script.

## Creating a Plug-in Serializer

Both plug-in serializers and linked serializers use the same exposed public methods and events, and similarly can be designed to operate either synchronously or asynchronously. There are three basic steps normally followed to create a plug-in serializer:

1. Implement and debug the serializer first as a linked serializer. Initially creating the serializer compiled and linked directly in a test application provides full access to breakpoints and other debug services helpful for testing.
2. After the serializer is fully tested, an [ISerializerFactory](#) interface is added to create a plug-in. The [ISerializerFactory](#) interface permits full access to all .NET Framework objects which includes the logical tree, [UIElement](#) objects, [IDocumentPaginatorSource](#), and [Visual](#) elements. Additionally [ISerializerFactory](#) provides the same synchronous and asynchronous methods and events used by linked serializers. Since large documents can take time to output, asynchronous operations are recommended to maintain responsive user interaction and offer a "Cancel" option if some problem occurs with the data store.
3. After the plug-in serializer is created, an installation script is implemented for distributing and installing (and uninstalling) the plug-in (see above, "[Installing Plug-in Serializers](#)").

## See also

- [System.Windows.Documents.Serialization](#)
- [XpsDocumentWriter](#)
- [XpsDocument](#)
- [Documents in WPF](#)
- [Printing Overview](#)
- [XML Paper Specification: Overview](#)

# Annotations

3/5/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides document viewing controls that support annotating document content.

## In This Section

[Annotations Overview](#)

[Annotations Schema](#)

## Reference

[Annotation](#)

[AnnotationService](#)

[DocumentViewer](#)

## Related Sections

[Documents in WPF](#)

[Flow Document Overview](#)

# Annotations Overview

10/7/2019 • 5 minutes to read • [Edit Online](#)

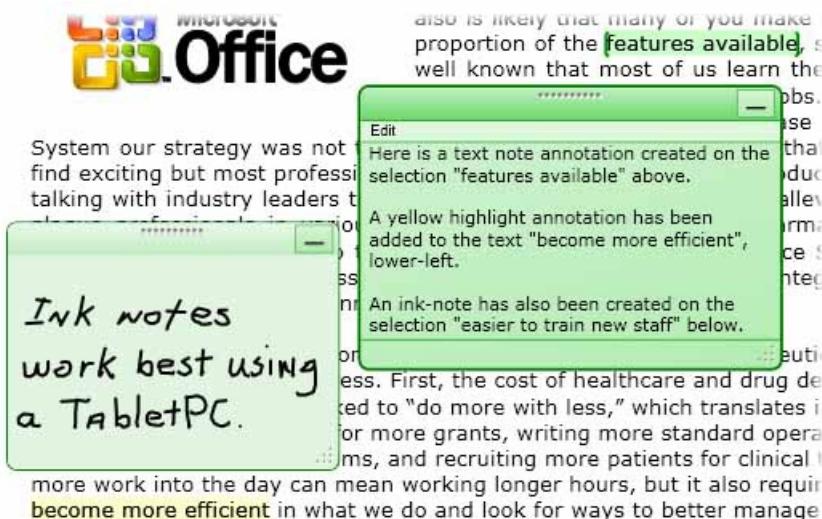
Writing notes or comments on paper documents is such a commonplace activity that we almost take it for granted. These notes or comments are "annotations" that we add to a document to flag information or to highlight items of interest for later reference. Although writing notes on printed documents is easy and commonplace, the ability to add personal comments to electronic documents is typically very limited, if available at all.

This topic reviews several common types of annotations, specifically sticky notes and highlights, and illustrates how the Microsoft Annotations Framework facilitates these types of annotations in applications through the Windows Presentation Foundation (WPF) document viewing controls. WPF document viewing controls that support annotations include [FlowDocumentReader](#) and [FlowDocumentScrollView](#), as well as controls derived from [DocumentViewerBase](#) such as [DocumentViewer](#) and [FlowDocumentPageViewer](#).

## Sticky Notes

A typical sticky note contains information written on a small piece of colored paper that is then "stuck" to a document. Digital sticky notes provide similar functionality for electronic documents, but with the added flexibility to include many other types of content such as typed text, handwritten notes (for example, Tablet PC "ink" strokes), or Web links.

The following illustration shows some examples of highlight, text sticky note, and ink sticky note annotations.



The following example shows the method that you can use to enable annotation support in your application.

```

// ----- StartAnnotations -----
/// <summary>
///   Enables annotations and displays all that are viewable.</summary>
private void StartAnnotations()
{
    // If there is no AnnotationService yet, create one.
    if (_annotService == null)
        // docViewer is a document viewing control named in Window1.xaml.
        _annotService = new AnnotationService(docViewer);

    // If the AnnotationService is currently enabled, disable it.
    if (_annotService.IsEnabled == true)
        _annotService.Disable();

    // Open a stream to the file for storing annotations.
    _annotStream = new FileStream(
        _annotStorePath, FileMode.OpenOrCreate, FileAccess.ReadWrite);

    // Create an AnnotationStore using the file stream.
    _annotStore = new XmlStreamStore(_annotStream);

    // Enable the AnnotationService using the new store.
    _annotService.Enable(_annotStore);
}// end:StartAnnotations()

```

```

' ----- StartAnnotations -----
''' <summary>
'''   Enables annotations and displays all that are viewable.</summary>
Private Sub StartAnnotations()
    ' If there is no AnnotationService yet, create one.
    If _annotService Is Nothing Then
        ' docViewer is a document viewing control named in Window1.xaml.
        _annotService = New AnnotationService(docViewer)
    End If

    ' If the AnnotationService is currently enabled, disable it.
    If _annotService.IsEnabled = True Then
        _annotService.Disable()
    End If

    ' Open a stream to the file for storing annotations.
    _annotStream = New FileStream(_annotStorePath, FileMode.OpenOrCreate, FileAccess.ReadWrite)

    ' Create an AnnotationStore using the file stream.
    _annotStore = New XmlStreamStore(_annotStream)

    ' Enable the AnnotationService using the new store.
    _annotService.Enable(_annotStore)
End Sub

```

## Highlights

People use creative methods to draw attention to items of interest when they mark up a paper document, such as underlining, highlighting, circling words in a sentence, or drawing marks or notations in the margin. Highlight annotations in Microsoft Annotations Framework provide a similar feature for marking up information displayed in WPF document viewing controls.

The following illustration shows an example of a highlight annotation.

**Highlight** The Microsoft Annotation Framework makes it **easy to add annotation capabilities** to virtually any application.

Users typically create annotations by first selecting some text or an item of interest, and then right-clicking to display a [ContextMenu](#) of annotation options. The following example shows the Extensible Application Markup Language (XAML) you can use to declare a [ContextMenu](#) with routed commands that users can access to create and manage annotations.

```
<DocumentViewer.ContextMenu>
  <ContextMenu>
    <MenuItem Command="ApplicationCommands.Copy" />
    <Separator />
    <!-- Add a Highlight annotation to a user selection. -->
    <MenuItem Command="ann:AnnotationService.CreateHighlightCommand"
      Header="Add Highlight" />
    <!-- Add a Text Note annotation to a user selection. -->
    <MenuItem Command="ann:AnnotationService.CreateTextStickyNoteCommand"
      Header="Add Text Note" />
    <!-- Add an Ink Note annotation to a user selection. -->
    <MenuItem Command="ann:AnnotationService.CreateInkStickyNoteCommand"
      Header="Add Ink Note" />
    <Separator />
    <!-- Remove Highlights from a user selection. -->
    <MenuItem Command="ann:AnnotationService.ClearHighlightsCommand"
      Header="Remove Highlights" />
    <!-- Remove Text Notes and Ink Notes from a user selection. -->
    <MenuItem Command="ann:AnnotationService.DeleteStickyNotesCommand"
      Header="Remove Notes" />
    <!-- Remove Highlights, Text Notes, Ink Notes from a selection. -->
    <MenuItem Command="ann:AnnotationService.DeleteAnnotationsCommand"
      Header="Remove Highlights &amp; Notes" />
  </ContextMenu>
</DocumentViewer.ContextMenu>
```

## Data Anchoring

The Annotations Framework binds annotations to the data that the user selects, not just to a position on the display view. Therefore, if the document view changes, such as when the user scrolls or resizes the display window, the annotation stays with the data selection to which it is bound. For example, the following graphic illustrates an annotation that the user has made on a text selection. When the document view changes (scrolls, resizes, scales, or otherwise moves), the highlight annotation moves with the original data selection.

**Original** The Microsoft Annotation Framework makes it **easy to add annotation capabilities** to virtually any application.

**Reflowed** The Microsoft Annotation Framework makes it **easy to add annotation capabilities** to virtually any application.

## Matching Annotations with Annotated Objects

You can match annotations with the corresponding annotated objects. For example, consider a simple document reader application that has a comments pane. The comments pane might be a list box that displays the text from a list of annotations that are anchored to a document. If the user selects an item in the list box, then the application brings into view the paragraph in the document that the corresponding annotation object is anchored to.

The following example demonstrates how to implement the event handler of such a list box that serves as the comments pane.

```

void annotationsListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    Annotation comment = (sender as ListBox).SelectedItem as Annotation;
    if (comment != null)
    {
        // IAnchorInfo info;
        // service is an AnnotationService object
        // comment is an Annotation object
        info = AnnotationHelper.GetAnchorInfo(this.service, comment);
        TextAnchor resolvedAnchor = info.ResolvedAnchor as TextAnchor;
        TextPointer textPointer = (TextPointer)resolvedAnchor.BoundingStart;
        textPointer.Paragraph.BringIntoView();
    }
}

```

```

Private Sub annotationsListBox_SelectionChanged(ByVal sender As Object, ByVal e As SelectionChangedEventArgs)

    Dim comment As Annotation = TryCast((TryCast(sender, ListBox)).SelectedItem, Annotation)
    If comment IsNot Nothing Then
        ' service is an AnnotationService object
        ' comment is an Annotation object
        info = AnnotationHelper.GetAnchorInfo(Me.service, comment)
        Dim resolvedAnchor As TextAnchor = TryCast(info.ResolvedAnchor, TextAnchor)
        Dim textPointer As TextPointer = CType(resolvedAnchor.BoundingStart, TextPointer)
        textPointer.Paragraph.BringIntoView()
    End If
End Sub

```

Another example scenario involves applications that enable the exchange of annotations and sticky notes between document readers through email. This feature enables these applications to navigate the reader to the page that contains the annotation that is being exchanged.

## See also

- [DocumentViewerBase](#)
- [DocumentViewer](#)
- [FlowDocumentPageViewer](#)
- [FlowDocumentScrollView](#)
- [FlowDocumentReader](#)
- [IAnchorInfo](#)
- [Annotations Schema](#)
- [ContextMenu Overview](#)
- [Commanding Overview](#)
- [Flow Document Overview](#)
- [How to: Add a Command to a MenuItem](#)

# Annotations Schema

7/20/2019 • 9 minutes to read • [Edit Online](#)

This topic describes the XML schema definition (XSD) used by the Microsoft Annotations Framework to save and retrieve user annotation data.

The Annotations Framework serializes annotation data from an internal representation to an XML format. The XML format used for this conversion is described by the Annotations Framework XSD Schema. The schema defines the implementation-independent XML format that can be used to exchange annotation data between applications.

The Annotations Framework XML schema definition consists of two subschemas

- The Annotations XML Core Schema (Core Schema).
- The Annotations XML Base Schema (Base Schema).

The Core Schema defines the primary XML structure of an [Annotation](#). The majority of XML elements defined in the Core Schema correspond to types in the [System.Windows.Annotations](#) namespace. The Core Schema exposes three extension points where applications can add their own XML data. These extension points include the [Authors](#), [ContentLocatorPart](#), and "Content". (Content elements are provided in the form of an [XmlElement](#) list.)

The Base Schema described in this topic defines the extensions for the [Authors](#), [ContentLocatorPart](#), and Content types included with the initial Windows Presentation Foundation (WPF) release.

## Annotations XML Core Schema

The Annotations XML Core Schema defines the XML structure that is used to store [Annotation](#) objects.

```
<xsd:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
    blockDefault="#all"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://schemas.microsoft.com/windows/annotations/2003/11/core"
    xmlns:anc="http://schemas.microsoft.com/windows/annotations/2003/11/core">

    <!-- The Annotations element groups a number of annotations. -->
    <xsd:element name="Annotations" type="anc:AnnotationsType" />

    <xsd:complexType name="AnnotationsType">
        <xsd:sequence>
            <xsd:element name="Annotation" minOccurs="0" maxOccurs="unbounded"
                type="anc:AnnotationType" />
        </xsd:sequence>
    </xsd:complexType>

    <!-- AnnotationType defines the structure of the Annotation element. -->
    <xsd:complexType name="AnnotationType">
        <xsd:sequence>

            <!-- List of 0 or more authors. -->
            <xsd:element name="Authors" minOccurs="0" maxOccurs="1"
                type="anc:AuthorListType" />

            <!-- List of 0 or more anchors. -->
            <xsd:element name="Anchors" minOccurs="0" maxOccurs="1"
                type="anc:ResourceListType" />

            <!-- List of 0 or more cargos. -->
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

```

<xsd:element name="Cargos" minOccurs="0" maxOccurs="1"
             type="anc:ResourceListType" />

</xsd:sequence>

<!-- Unique annotation ID. -->
<xsd:attribute name="Id" type="xsd:string" use="required" />

<!-- Annotation "Type" is used to map the annotation to an annotation
     component that takes care of the visual representation of the
     annotation. WPF V1 recognizes three annotation types:
http://schemas.microsoft.com/windows/annotations/2003/11/base:Highlight
http://schemas.microsoft.com/windows/annotations/2003/11/base:TextStickyNote
http://schemas.microsoft.com/windows/annotations/2003/11/base:InkStickyNote
-->
<xsd:attribute name="Type" type="xsd:QName" use="required" />

<!-- Time when the annotation was last modified. -->
<xsd:attribute name="LastModificationTime" use="optional"
               type="xsd:dateTime" />

<!-- Time when the annotation was created. -->
<xsd:attribute name="CreationTime" use="optional"
               type="xsd:dateTime" />
</xsd:complexType>

<!-- "Authors" consists of 0 or more elements that represent an author. -->
<xsd:complexType name="AuthorListType">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:element ref="anc:Author" />
  </xsd:sequence>
</xsd:complexType>

<!-- The core schema allows any author type. Supported author types
     in version 1 (V1) are described in the base schema. -->
<xsd:element name="Author" abstract="true" block="extension restriction"/>

<!-- Both annotation anchor and annotation cargo are represented by the
     ResourceListType which contains 0 or more "Resource" elements. -->
<xsd:complexType name="ResourceListType">
  <xsd:sequence>
    <xsd:element name="Resource" minOccurs="0" maxOccurs="unbounded"
                type="anc:ResourceType" />
  </xsd:sequence>
</xsd:complexType>

<!-- Resource groups identification, location
     and/or content of some information. -->
<xsd:complexType name="ResourceType">
  <xsd:choice minOccurs="0" maxOccurs="unbounded" >
    <xsd:choice>
      <xsd:element name="ContentLocator" type="anc:ContentLocatorType" />
      <xsd:element name="ContentLocatorGroup" type="anc:ContentLocatorGroupType" />
    </xsd:choice>
    <xsd:element ref="anc:Content"/>
  </xsd:choice>

  <!-- Unique resource identifier. -->
  <xsd:attribute name="Id" type="xsd:string" use="required" />

  <!-- Optional resource name. -->
  <xsd:attribute name="Name" type="xsd:string" use="optional" />
</xsd:complexType>

<!-- ContentLocatorGroup contains a set of ContentLocators -->
<xsd:complexType name="ContentLocatorGroupType">
  <xsd:sequence>
    <xsd:element name="ContentLocator" minOccurs="1" maxOccurs="unbounded"
                type="anc:ContentLocatorType" />

```

```

</xsd:sequence>
</xsd:complexType>

<!-- A ContentLocator describes the location or the identification
     of particular data within some context. The ContentLocator consists
     of one or more ContentLocatorParts. Each ContentLocatorPart needs to
     be successively applied to the context to arrive at the data. What
     "applying", "context", and "data" mean is application dependent.
-->
<xsd:complexType name="ContentLocatorType">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:element ref="anc:ContentLocatorPart" />
  </xsd:sequence>
</xsd:complexType>

<!-- A ContentLocatorPart is a set of "Item" elements. Each "Item" element
     has "Name" and "Value" attributes that define a name/value pair.
     ContentLocatorPart is an abstract type that must be restricted for each
     concrete ContentLocatorPart definition. This restriction should define
     allowed names and values for the concrete ContentLocatorPart type. That
     way the application can define its own way of locating information. The
     ContentLocatorPartTypes that are allowed in version 1 (V1) of WPF are
     defined in the Annotations Base Schema.
-->
<xsd:element name="ContentLocatorPart" type="anc:ContentLocatorPartType"
             abstract="true" />

<xsd:complexType name="ContentLocatorPartType" abstract="true"
                  block="restriction">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="Item" type="anc:ItemType" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ItemType" abstract="true" >
  <xsd:attribute name="Name" type='xsd:string' use="required" />
  <xsd:attribute name="Value" type='xsd:string' use="optional" />
</xsd:complexType>

<!-- Content describes the underlying content of a resource. This is an
     abstract type that should be redefined for each concrete content type
     through restriction. Allowed content types in WPF version 1 are
     defined in the Annotations Base Schema.
-->
<xsd:element name="Content" abstract="true" block="extension restriction"/>

</xsd:schema>

```

## Annotations XML Base Schema

The Base Schema defines the XML structure for the three abstract elements defined in the Core Schema – [Authors](#), [ContentLocatorPart](#), and [Contents](#).

```

<xsd:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
             blockDefault="#all"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://schemas.microsoft.com/windows/annotations/2003/11/base"
             xmlns:anb="http://schemas.microsoft.com/windows/annotations/2003/11/base"
             xmlns:anc="http://schemas.microsoft.com/windows/annotations/2003/11/core">

  <xsd:import schemaLocation="AnnotationCoreV1.xsd"
              namespace="http://schemas.microsoft.com/windows/annotations/2003/11/core"/>

  <!-- ***** Author ***** -->
  <!-- Simple DisplayName Author -->
  <xsd:complexType name="StringAuthorType">

```

```

<xsd:simpleContent >
  <xsd:extension base='xsd:string' />
</xsd:simpleContent>
</xsd:complexType>
<xsd:element name="StringAuthor" type="anb:StringAuthorType"
  substitutionGroup="anc:Author"/>

<!-- ***** LocatorParts ***** -->

<!-- Helper types -->

<!-- CountItemNameType - helper type to define count item -->
<xsd:simpleType name="CountItemNameType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value="Count" />
  </xsd:restriction>
</xsd:simpleType>

<!-- NumberType - helper type to define segment count item -->
<xsd:simpleType name="NumberType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value="\d*" />
  </xsd:restriction>
</xsd:simpleType>

<!-- SegmentNameType: helper type to define possible segment name types -->
<xsd:simpleType name="SegmentItemNameType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value="Segment\d*" />
  </xsd:restriction>
</xsd:simpleType>

<!-- Flow Locator Part -->

<!-- FlowSegmentValueItemType: helper type to define flow segment values -->
<xsd:simpleType name="FlowSegmentItemValueType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value=" \d*,\d*" />
  </xsd:restriction>
</xsd:simpleType>

<!-- FlowItemType -->
<xsd:complexType name="FlowItemType" abstract = "true">
  <xsd:complexContent>
    <xsd:restriction base="anc:ItemType">
      </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<!-- FlowSegmentItemType -->
<xsd:complexType name="FlowSegmentItemType">
  <xsd:complexContent>
    <xsd:restriction base="anb:FlowItemType">
      <xsd:attribute name="Name" use="required"
        type="anb:SegmentItemNameType"/>
      <xsd:attribute name="Value" use="required"
        type="anb:FlowSegmentItemValueType"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<!-- FlowCountItemType -->
<xsd:complexType name="FlowCountItemType">
  <xsd:complexContent>
    <xsd:restriction base="anb:FlowItemType">
      <xsd:attribute name="Name" type="anb:CountItemNameType" use="required"/>
      <xsd:attribute name="Value" type="anb:NumberType" use="required"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

```

```

</xsd:complexContent>
</xsd:complexType>

<!-- CharacterRangeType is an extension of ContentLocatorPartType that locates
*   part of the content within a FlowDocument. CharacterRangeType contains one
*   "Item" element with name "Count" and value the number(N) of "SegmentXX"
*   elements that this ContentLocatorPart has. It also contains N "Item"
*   elements with name "SegmentXX" where XX is a number from 0 to N-1. The
*   value of each "SegmentXX" element is a string in the form "offset, length"
*   which locates one sequence of symbols in the FlowDocument. Example:

*     <anb:CharacterRange>
*       <anc:Item Name="Count" Value="2" />
*       <anc:Item Name="Segment0" Value="5,10" />
*       <anc:Item Name="Segment1" Value="25,2" />
*     </anb:CharacterRange>
-->
<xsd:complexType name="CharacterRangeType">
  <xsd:complexContent>
    <xsd:extension base="anc:ContentLocatorPartType">
      <xsd:sequence minOccurs="1" maxOccurs="unbounded">
        <xsd:element name="Item" type="anb:FlowItemType" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- CharacterRange element substitutes ContentLocatorPart element -->
<xsd:element name="CharacterRange" type="anb:CharacterRangeType"
  substitutionGroup="anc:ContentLocatorPart"/>

<!-- Fixed LocatorPart -->

<!-- Helper type - FixedItemType -->
<xsd:complexType name="FixedItemType" abstract = "true">
  <xsd:complexContent>
    <xsd:restriction base="anc:ItemType">
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<!-- Helper type - FixedCountItemType: ContentLocatorPart items count -->
<xsd:complexType name="FixedCountItemType">
  <xsd:complexContent>
    <xsd:restriction base="anb:FixedItemType">
      <xsd:attribute name="Name" type="anb:CountItemNameType" use="required"/>
      <xsd:attribute name="Value" type="anb:NumberType" use="required"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<!-- Helper type -FixedSegmentValue: Defines possible fixed segment values -->
<xsd:simpleType name="FixedSegmentItemValueType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value="\d*,\d*,\d*,\d*" />
  </xsd:restriction>
</xsd:simpleType>

<!-- Helper type - FixedSegmentItemType -->
<xsd:complexType name="FixedSegmentItemType">
  <xsd:complexContent>
    <xsd:restriction base="anb:FixedItemType">
      <xsd:attribute name="Name" use="required"
                    type="anb:SegmentItemNameType"/>
      <xsd:attribute name="Value" use="required"
                    type="anb:FixedSegmentItemValueType" />
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

```

```

<!-- FixedTextRangeType is an extension of ContentLocatorPartType that locates
*   content within a FixedDocument. It contains one "Item" element with name
*   "Count" and value the number (N) of "Item" elements with name "SegmentXX"
*   that this ContentLocatorPart has. FixedTextRange locator part also
*   contains N "Item" elements with one attribute Name="SegmentXX" where XX is
*   a number from 0 to N-1 and one attribute "Value" in the form "X1, Y1, X2,
*   Y2". Here X1,Y1 are the coordinates of the start symbol in this segment,
*   X2,Y2 are the coordinates of the end symbol in this segment. Example:
*

*      <anb:FixedTextRange>
*          <anc:Item Name="Count" Value="2" />
*          <anc:Item Name="Segment0" Value="10,5,20,5" />
*          <anc:Item Name="Segment1" Value="25,15, 25,20" />
*      </anb:FixedTextRange>
-->
<xsd:complexType name="FixedTextRangeType">
    <xsd:complexContent>
        <xsd:extension base="anc:ContentLocatorPartType">
            <xsd:sequence minOccurs="1" maxOccurs="unbounded">
                <xsd:element name="Item" type="anb:FixedItemType" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<!-- FixedTextRange element substitutes ContentLocatorPart element -->
<xsd:element name="FixedTextRange" type="anb:FixedTextRangeType"
    substitutionGroup="anc:ContentLocatorPart"/>

<!-- DataId -->

<!-- ValueItemNameType: helper type to define value item -->
<xsd:simpleType name="ValueItemNameType">
    <xsd:restriction base='xsd:string'>
        <xsd:pattern value="Value" />
    </xsd:restriction>
</xsd:simpleType>

<!-- StringValueItemType -->
<xsd:complexType name="StringValueItemType">
    <xsd:complexContent>
        <xsd:restriction base="anc:ItemType">
            <xsd:attribute name="Name" type="anb:ValueItemNameType" use="required"/>
            <xsd:attribute name="Value" type="xsd:string" use="required"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="StringValueLocatorPartType">
    <xsd:complexContent>
        <xsd:extension base="anc:ContentLocatorPartType">
            <xsd:sequence minOccurs="1" maxOccurs="1">
                <xsd:element name="Item" type="anb:ValueItemType" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<!-- DataId element substitutes ContentLocatorPart and is used to locate a
*   subtree in the logical tree. Including DataId locator part in a
*   ContentLocator helps to narrow down the search for a particular content.
*   Example of DataId ContentLocatorPart:
*

*      <anb:DataId>
*          <anc:Item Name="Value" Value="FlowDocument" />
*      </anb:DataId>
-->
```

```

<xsd:element name="DataId" type="anb: StringValueLocatorPartType "
substitutionGroup="anc:ContentLocatorPart"/>

<!-- PageNumber -->

<!-- NumberValueItemType -->
<xsd:complexType name="NumberValueItemType">
<xsd:complexContent>
<xsd:restriction base="anc:ItemType">
<xsd:attribute name="Name" type="anb:ValueItemNameType" use="required"/>
<xsd:attribute name="Value" type="anb:NumberType" use="required"/>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="NumberValueLocatorPartType">
<xsd:complexContent>
<xsd:extension base="anc:ContentLocatorPartType">
<xsd:sequence minOccurs="1" maxOccurs="1">
<xsd:element name="Item" type="anb:ValueItemType" />
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<!-- PageNumber element substitutes ContentLocatorPart and is used to locate a
*   page in a FixedDocument. PageNumber ContentLocatorPart is used in
*   conjunction with the FixedTextRange ContentLocatorPart and it shows on with
*   page are the coordinates defined in the FixedTextRange.
*   Example of a PageNumber ContentLocatorPart:
*
*     <anb:PageNumber>
*       <anc:Item Name="Value" Value="1" />
*     </anb:PageNumber>
-->
<xsd:element name="PageNumber" type="anb:NumberValueLocatorPartType"
substitutionGroup="anc:ContentLocatorPart"/>

<!-- ***** Content ***** -->
<!-- Highlight colors - defines highlight color for annotations of type
*   Highlight or normal and active anchor colors for annotations of type
*   TextStickyNote and InkStickyNote.
-->
<xsd:complexType name="ColorsContentType">
<xsd:attribute name="Background" type='xsd:string' use="required" />
<xsd:attribute name="ActiveBackground" type='xsd:string' use="optional" />
</xsd:complexType>

<xsd:element name="Colors" type="anb:ColorsContentType"
substitutionGroup="anc:Content"/>

<!-- RTB Text -contains XAML representing StickyNote Reach Text Box text.
*   Used in annotations of type TextStickyNote. -->
<xsd:complexType name="TextContentType">
<!-- See XAML schema for RTB content -->
</xsd:complexType>

<xsd:element name="Text" type="anb:TextContentType"
substitutionGroup="anc:Content"/>

<!-- Ink - contains XAML representing Sticky Note ink.
*   Used in annotations of type InkStickyNote. -->
<xsd:complexType name="InkContentType">
<!-- See XAML schema for Ink content -->
</xsd:complexType>

<xsd:element name="Ink" type="anb:InkContentType"
substitutionGroup="anc:Content"/>

```

```

<!-- SN Metadata - defines StickyNote attributes as position width, height,
*   etc. Used in annotations of type TextStickyNote and InkStickyNote. -->
<xsd:complexType name="MetadataContentType">
    <xsd:attribute name="Left" type='xsd:decimal' use="optional" />
    <xsd:attribute name="Top" type='xsd:decimal' use="optional" />
    <xsd:attribute name="Width" type='xsd:decimal' use="optional" />
    <xsd:attribute name="Height" type='xsd:decimal' use="optional" />
    <xsd:attribute name="XOffset" type='xsd:decimal' use="optional" />
    <xsd:attribute name="YOffset" type='xsd:decimal' use="optional" />
    <xsd:attribute name="ZOrder" type='xsd:decimal' use="optional" />
</xsd:complexType>

<xsd:element name="Metadata" type="anb:MetadataContentType"
    substitutionGroup="anc:Content"/>

</xsd:schema>

```

## Sample XML Produced by Annotations XmlStreamStore

The XML that follows shows the output of an Annotations [XmlStreamStore](#) and the organization of a sample file that contains three annotations - a highlight, a text sticky-note, and an ink stick-note.

```

<?xml version="1.0" encoding="utf-8"?>
<anc:Annotations
    xmlns:anc="http://schemas.microsoft.com/windows/annotations/2003/11/core"
    xmlns:anb="http://schemas.microsoft.com/windows/annotations/2003/11/base">

    <anc:Annotation Id="d308ea9b-36eb-4cc4-94d0-97634f10f7a2"
        CreationTime="2006-09-13T18:28:51.4465702-07:00"
        LastModificationTime="2006-09-13T18:28:51.4465702-07:00"
        Type="anb:Highlight">
        <anc:Anchors>
            <anc:Resource Id="4f53661b-7328-4673-8e3f-c53f08b9cd94">
                <anc:ContentLocator>
                    <anb:DataId>
                        <anc:Item Name="Value" Value="FlowDocument" />
                    </anb:DataId>
                    <anb:CharacterRange>
                        <anc:Item Name="Segment0" Value="600,609" />
                        <anc:Item Name="Count" Value="1" />
                    </anb:CharacterRange>
                </anc:ContentLocator>
            </anc:Resource>
        </anc:Anchors>
    </anc:Annotation>

    <anc:Annotation Id="d7a8d271-387e-4144-9f8b-bc3c97816e5f"
        CreationTime="2006-09-13T18:28:56.7903202-07:00"
        LastModificationTime="2006-09-13T18:28:56.8996952-07:00"
        Type="anb:TextStickyNote">
        <anc:Authors>
            <anb:StringAuthor>Denise Smith</anb:StringAuthor>
        </anc:Authors>

        <anc:Anchors>
            <anc:Resource Id="dab2560e-6ebd-4ad0-80f9-483356a3be0b">
                <anc:ContentLocator>
                    <anb:DataId>
                        <anc:Item Name="Value" Value="FlowDocument" />
                    </anb:DataId>
                    <anb:CharacterRange>
                        <anc:Item Name="Segment0" Value="787,801" />
                        <anc:Item Name="Count" Value="1" />
                    </anb:CharacterRange>
                </anc:ContentLocator>
            </anc:Resource>
        </anc:Anchors>
    </anc:Annotation>

```

```

</anc:Annotations>

```

<anc:Cargos>

```

<anc:Resource Id="ea4dbabd-b400-4cf9-8908-5716b410f9e4" Name="Meta Data">
    <anb:MetaData anb:ZOrder="0" />
</anc:Resource>
</anc:Cargos>
</anc:Annotation>
```

<anc:Annotation Id="66803c69-b0d7-4cc3-bdff-cacc1955e806">

```

    CreationTime="2006-09-13T18:29:03.6653202-07:00"
    LastModificationTime="2006-09-13T18:29:03.7121952-07:00"
    Type="anb:InkStickyNote">
```

<anc:Authors>

```

    <anb:StringAuthor>Mike Nash</anb:StringAuthor>
</anc:Authors>
```

<anc:Anchors>

```

    <anc:Resource Id="52251c53-8eeb-4fd7-b8f3-94e78dfc25fa">
        <anc:ContentLocator>
            <anb:DataId>
                <anc:Item Name="Value" Value="FlowDocument" />
            </anb:DataId>
            <anb:CharacterRange>
                <anc:Item Name="Segment0" Value="880,884" />
                <anc:Item Name="Count" Value="1" />
            </anb:CharacterRange>
        </anc:ContentLocator>
    </anc:Resource>
</anc:Anchors>
```

<anc:Cargos>

```

    <anc:Resource Id="11e50b97-8d91-4ff9-82c3-16607b2b552b" Name="Meta Data">
        <anb:MetaData anb:ZOrder="1" />
    </anc:Resource>
</anc:Cargos>
</anc:Annotation>
```

</anc:Annotations>

## See also

- [System.Windows.Annotations](#)
- [System.Windows.Annotations.Storage](#)
- [Annotation](#)
- [AnnotationStore](#)
- [XmlStreamStore](#)
- [Annotations Overview](#)

# Flow Content

3/5/2019 • 2 minutes to read • [Edit Online](#)

Flow content elements provide the building blocks for creating flow content suitable for hosting in a [FlowDocument](#).

## In This Section

[Flow Document Overview](#)

[TextElement Content Model Overview](#)

[Table Overview](#)

[How-to Topics](#)

## Reference

[FlowDocument](#)

[Block](#)

[List](#)

[Paragraph](#)

[Section](#)

[Table](#)

[Figure](#)

[Floater](#)

[Hyperlink](#)

[Inline](#)

[Run](#)

[Span](#)

[ListItem](#)

## Related Sections

[Documents in WPF](#)

# Flow Document Overview

9/10/2019 • 26 minutes to read • [Edit Online](#)

Flow documents are designed to optimize viewing and readability. Rather than being set to one predefined layout, flow documents dynamically adjust and reflow their content based on run-time variables such as window size, device resolution, and optional user preferences. In addition, flow documents offer advanced document features, such as pagination and columns. This topic provides an overview of flow documents and how to create them.

## What is a Flow Document

A flow document is designed to "reflow content" depending on window size, device resolution, and other environment variables. In addition, flow documents have a number of built in features including search, viewing modes that optimize readability, and the ability to change the size and appearance of fonts. Flow Documents are best utilized when ease of reading is the primary document consumption scenario. In contrast, Fixed Documents are designed to have a static presentation. Fixed Documents are useful when fidelity of the source content is essential. See [Documents in WPF](#) for more information on different types of documents.

The following illustration shows a sample flow document viewed in several windows of different sizes. As the display area changes, the content reflows to make the best use of the available space.

**Flow Content**

Flow content is the primary content of a document, such as text, images, and lists. It is represented by the `FlowContent` class.

**Text Example:**

```
1. Text:  
    Text ipsum dolor sit amet, consectetur adipiscing elit. Nulla ligula tortor, dapibus et, facilisis a, aliquet et, diam. Cras convallis. Fusce at urna. Quisque interdum, turpis sed dictum fringilla, magna arcu gravida libero, elementum tincidunt dolor mauris sed erat. Curabitur egestas aliquet nibh. Etiam quis sem in lorem auctor consequat. Suspendisse vitae lorem. Proin eleifend elementum ligula. Donec mattis consectetur erat. Donec fermentum consectetur neque. Suspendisse tempus faucibus magna. Pellentesque sodales velit eget nibh. Phasellus velit magna, malesuada vitae, rhoncus eget, dignissim ut, metus. Praesent pulvinar suscipit diam.
```



**Image Example:**

Images are included in flow content using the `Image` class. They can be inline or block-level elements.

**Image Example:**

A photograph of a winding dirt path through a dense forest. The path is surrounded by tall trees and lush green undergrowth. The image is used as a visual element within the flow content.



**Text Example:**

```
1. Text:  
    Morbi ut tellus at tellus semper consectetur. Nullam fringilla nonummy justo. Proin rutrum, purus id adipiscing malesuada, enim velit accumsan nisi, pellentesque rhoncus nibh nisi ut magna. Nunc massa nulla, volutpat sit amet, pulvinar ac, scelerisque ac, magna. Nulla facilisi. Integer pharetra felis vitae risus. Nunc aliquet lacus pretium urna varius hendrerit. Duis odio nisl, venenatis at, sollicitudin eu, viverra sed, nibh. Nullam consequat. Duis felis felis, rutrum viverra, auctor eget, iaculis ut, mi. Integer sagittis pharetra ipsum. Donec lacinia scelerisque nisl. Nullam aliquet. In et sapien. Fusce blandit odio et mi.
```

« 1 of 2 »

**Text Example:**

```
1. Text:  
    Morbi ut tellus at tellus semper consectetur. Nullam fringilla nonummy justo. Proin rutrum, purus id adipiscing malesuada, enim velit accumsan nisi, pellentesque rhoncus nibh nisi ut magna. Nunc massa nulla, volutpat sit amet, pulvinar ac, scelerisque ac, magna. Nulla facilisi. Integer pharetra felis vitae risus. Nunc aliquet lacus pretium urna varius hendrerit. Duis odio nisl, venenatis at, sollicitudin eu, viverra sed, nibh. Nullam consequat. Duis felis felis, rutrum viverra, auctor eget, iaculis ut, mi. Integer sagittis pharetra ipsum. Donec lacinia scelerisque nisl. Nullam aliquet. In et sapien. Fusce blandit odio et mi.
```



**Text Example:**

```
1. Text:  
    Morbi ut tellus at tellus semper consectetur. Nullam fringilla nonummy justo. Proin rutrum, purus id adipiscing malesuada, enim velit accumsan nisi, pellentesque rhoncus nibh nisi ut magna. Nunc massa nulla, volutpat sit amet, pulvinar ac, scelerisque ac, magna. Nulla facilisi. Integer pharetra felis vitae risus. Nunc aliquet lacus pretium urna varius hendrerit. Duis odio nisl, venenatis at, sollicitudin eu, viverra sed, nibh. Nullam consequat. Duis felis felis, rutrum viverra, auctor eget, iaculis ut, mi. Integer sagittis pharetra ipsum. Donec lacinia scelerisque nisl. Nullam aliquet. In et sapien. Fusce blandit odio et mi.
```

**Text Example:**

```
1. Text:  
    Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse laoreet condimentum purus. Etiam vel enim. Suspendisse at dolor. Pellentesque gravida. Aenean convallis. Vivamus diam. Aenean lorem libero, suscipit vel, tempor eu, fermentum aliquam, metus. Quisque volutpat urna at augue. Nam placerat. In viverra congue tellus. Proin auctor. Fusce nisl lorem, dapibus vitae, porta sed, malesuada a, lacus. Phasellus mollis elementum enim. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam vitae urna vel velit volutpat tempor. Quisque ac dolor. Suspendisse potenti. Nunc nec tortor. Curabitur pharetra pellentesque diam.
```

« 1 of 1 »

As seen in the image above, flow content can include many components including paragraphs, lists, images, and more. These components correspond to elements in markup and objects in procedural code. We will go over these classes in detail later in the [Flow Related Classes](#) section of this overview. For now, here is a simple code example that creates a flow document consisting of a paragraph with some bold text and a list.

```
<!-- This simple flow document includes a paragraph with some
bold text in it and a list. -->
<FlowDocumentReader xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<FlowDocument>
    <Paragraph>
        <Bold>Some bold text in the paragraph.</Bold>
        Some text that is not bold.
    </Paragraph>

    <List>
        <ListItem>
            <Paragraph>ListItem 1</Paragraph>
        </ListItem>
        <ListItem>
            <Paragraph>ListItem 2</Paragraph>
        </ListItem>
        <ListItem>
            <Paragraph>ListItem 3</Paragraph>
        </ListItem>
    </List>

</FlowDocument>
</FlowDocumentReader>
```

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

namespace SDKSample
{
    public partial class SimpleFlowExample : Page
    {
        public SimpleFlowExample()
        {

            Paragraph myParagraph = new Paragraph();

            // Add some Bold text to the paragraph
            myParagraph.Inlines.Add(new Bold(new Run("Some bold text in the paragraph.")));

            // Add some plain text to the paragraph
            myParagraph.Inlines.Add(new Run(" Some text that is not bold."));

            // Create a List and populate with three list items.
            List myList = new List();

            // First create paragraphs to go into the list item.
            Paragraph paragraphListItem1 = new Paragraph(new Run("ListItem 1"));
            Paragraph paragraphListItem2 = new Paragraph(new Run("ListItem 2"));
            Paragraph paragraphListItem3 = new Paragraph(new Run("ListItem 3"));

            // Add ListItems with paragraphs in them.
            myList.ListItems.Add(new ListItem(paragraphListItem1));
            myList.ListItems.Add(new ListItem(paragraphListItem2));
            myList.ListItems.Add(new ListItem(paragraphListItem3));

            // Create a FlowDocument with the paragraph and list.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myParagraph);
            myFlowDocument.Blocks.Add(myList);

            // Add the FlowDocument to a FlowDocumentReader Control
            FlowDocumentReader myFlowDocumentReader = new FlowDocumentReader();
            myFlowDocumentReader.Document = myFlowDocument;

            this.Content = myFlowDocumentReader;
        }
    }
}
```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Documents

Namespace SDKSample
    Partial Public Class SimpleFlowExample
        Inherits Page
        Public Sub New()

            Dim myParagraph As New Paragraph()

            ' Add some Bold text to the paragraph
            myParagraph.Inlines.Add(New Bold(New Run("Some bold text in the paragraph.")))

            ' Add some plain text to the paragraph
            myParagraph.Inlines.Add(New Run(" Some text that is not bold."))

            ' Create a List and populate with three list items.
            Dim myList As New List()

            ' First create paragraphs to go into the list item.
            Dim paragraphListItem1 As New Paragraph(New Run("ListItem 1"))
            Dim paragraphListItem2 As New Paragraph(New Run("ListItem 2"))
            Dim paragraphListItem3 As New Paragraph(New Run("ListItem 3"))

            ' Add ListItems with paragraphs in them.
            myList.ListItems.Add(New ListItem(paragraphListItem1))
            myList.ListItems.Add(New ListItem(paragraphListItem2))
            myList.ListItems.Add(New ListItem(paragraphListItem3))

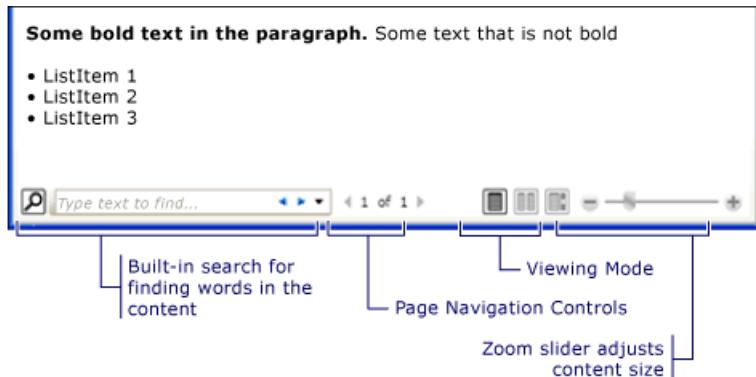
            ' Create a FlowDocument with the paragraph and list.
            Dim myFlowDocument As New FlowDocument()
            myFlowDocument.Blocks.Add(myParagraph)
            myFlowDocument.Blocks.Add(myList)

            ' Add the FlowDocument to a FlowDocumentReader Control
            Dim myFlowDocumentReader As New FlowDocumentReader()
            myFlowDocumentReader.Document = myFlowDocument

            Me.Content = myFlowDocumentReader
        End Sub
    End Class
End Namespace

```

The illustration below shows what this code snippet looks like.



In this example, the [FlowDocumentReader](#) control is used to host the flow content. See [Flow Document Types](#) for more information on flow content hosting controls. [Paragraph](#), [List](#), [ListItem](#), and [Bold](#) elements are used to control content formatting, based on their order in markup. For example, the [Bold](#) element spans across only part of the text in the paragraph; as a result, only that part of the text is bold. If you have used HTML, this will be familiar to you.

As highlighted in the illustration above, there are several features built into Flow Documents:

- Search: Allows the user to perform a full text search of an entire document.
- Viewing Mode: The user can select their preferred viewing mode including a single-page (page-at-a-time) viewing mode, a two-page-at-a-time (book reading format) viewing mode, and a continuous scrolling (bottomless) viewing mode. For more information about these viewing modes, see [FlowDocumentReaderViewingMode](#).
- Page Navigation Controls: If the viewing mode of the document uses pages, the page navigation controls include a button to jump to the next page (the down arrow) or previous page (the up arrow), as well as indicators for the current page number and total number of pages. Flipping through pages can also be accomplished using the keyboard arrow keys.
- Zoom: The zoom controls enable the user to increase or decrease the zoom level by clicking the plus or minus buttons, respectively. The zoom controls also include a slider for adjusting the zoom level. For more information, see [Zoom](#).

These features can be modified based upon the control used to host the flow content. The next section describes the different controls.

## Flow Document Types

Display of flow document content and how it appears is dependent upon what object is used to host the flow content. There are four controls that support viewing of flow content: [FlowDocumentReader](#), [FlowDocumentPageViewer](#), [RichTextBox](#), and [FlowDocumentScrollView](#). These controls are briefly described below.

### NOTE

[FlowDocument](#) is required to directly host flow content, so all of these viewing controls consume a [FlowDocument](#) to enable flow content hosting.

### FlowDocumentReader

[FlowDocumentReader](#) includes features that enable the user to dynamically choose between various viewing modes, including a single-page (page-at-a-time) viewing mode, a two-page-at-a-time (book reading format) viewing mode, and a continuous scrolling (bottomless) viewing mode. For more information about these viewing modes, see [FlowDocumentReaderViewingMode](#). If you do not need the ability to dynamically switch between different viewing modes, [FlowDocumentPageViewer](#) and [FlowDocumentScrollView](#) provide lighter-weight flow content viewers that are fixed in a particular viewing mode.

### FlowDocumentPageViewer and FlowDocumentScrollView

[FlowDocumentPageViewer](#) shows content in page-at-a-time viewing mode, while [FlowDocumentScrollView](#) shows content in continuous scrolling mode. Both [FlowDocumentPageViewer](#) and [FlowDocumentScrollView](#) are fixed to a particular viewing mode. Compare to [FlowDocumentReader](#), which includes features that enable the user to dynamically choose between various viewing modes (as provided by the [FlowDocumentReaderViewingMode](#) enumeration), at the cost of being more resource intensive than [FlowDocumentPageViewer](#) or [FlowDocumentScrollView](#).

By default, a vertical scrollbar is always shown, and a horizontal scrollbar becomes visible if needed. The default UI for [FlowDocumentScrollView](#) does not include a toolbar; however, the [IsToolBarVisible](#) property can be used to enable a built-in toolbar.

### RichTextBox

You use a [RichTextBox](#) when you want to allow the user to edit flow content. For example, if you wanted to

create an editor that allowed a user to manipulate things like tables, italic and bold formatting, etc, you would use a [RichTextBox](#). See [RichTextBox Overview](#) for more information.

#### NOTE

Flow content inside a [RichTextBox](#) does not behave exactly like flow content contained in other controls. For example, there are no columns in a [RichTextBox](#) and hence no automatic resizing behavior. Also, the typically built in features of flow content like search, viewing mode, page navigation, and zoom are not available within a [RichTextBox](#).

## Creating Flow Content

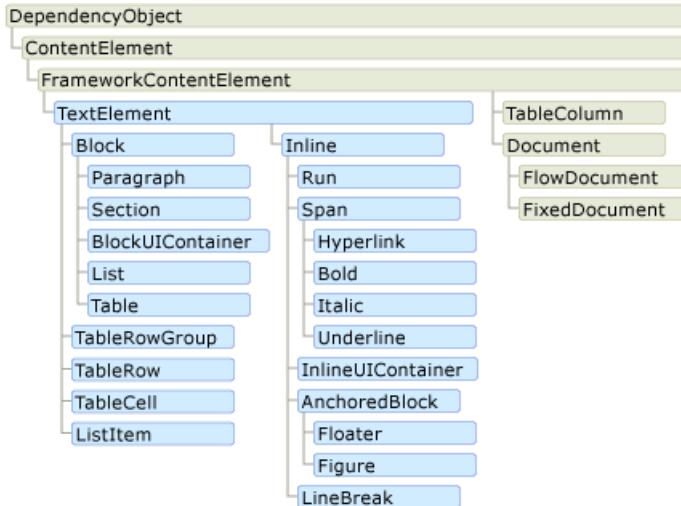
Flow content can be complex, consisting of various elements including text, images, tables, and even [UIElement](#) derived classes like controls. To understand how to create complex flow content, the following points are critical:

- **Flow-related Classes:** Each class used in flow content has a specific purpose. In addition, the hierarchical relation between flow classes helps you understand how they are used. For example, classes derived from the [Block](#) class are used to contain other objects while classes derived from [Inline](#) contain objects that are displayed.
- **Content Schema:** A flow document can require a substantial number of nested elements. The content schema specifies possible parent/child relationships between elements.

The following sections will go over each of these areas in more detail.

## Flow Related Classes

The diagram below shows the objects most typically used with flow content:



For the purposes of flow content, there are two important categories:

1. **Block-derived classes:** Also called "Block content elements" or just "Block Elements". Elements that inherit from [Block](#) can be used to group elements under a common parent or to apply common attributes to a group.
2. **Inline-derived classes:** Also called "Inline content elements" or just "Inline Elements". Elements that inherit from [Inline](#) are either contained within a Block Element or another Inline Element. Inline Elements are often used as the direct container of content that is rendered to the screen. For example, a [Paragraph](#) (Block Element) can contain a [Run](#) (Inline Element) but the [Run](#) actually contains the text that is rendered on the screen.

Each class in these two categories is briefly described below.

## Block-derived Classes

### Paragraph

[Paragraph](#) is typically used to group content into a paragraph. The simplest and most common use of Paragraph is to create a paragraph of text.

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Paragraph>
        Some paragraph text.
    </Paragraph>
</FlowDocument>
```

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

namespace SDKSample
{
    public partial class ParagraphExample : Page
    {
        public ParagraphExample()
        {

            // Create paragraph with some text.
            Paragraph myParagraph = new Paragraph();
            myParagraph.Inlines.Add(new Run("Some paragraph text."));

            // Create a FlowDocument and add the paragraph to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myParagraph);

            this.Content = myFlowDocument;
        }
    }
}
```

```
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Documents

Namespace SDKSample
    Partial Public Class ParagraphExample
        Inherits Page
        Public Sub New()

            ' Create paragraph with some text.
            Dim myParagraph As New Paragraph()
            myParagraph.Inlines.Add(New Run("Some paragraph text.))

            ' Create a FlowDocument and add the paragraph to it.
            Dim myFlowDocument As New FlowDocument()
            myFlowDocument.Blocks.Add(myParagraph)

            Me.Content = myFlowDocument
        End Sub
    End Class
End Namespace
```

However, you can also contain other inline-derived elements as you will see below.

## Section

[Section](#) is used only to contain other [Block](#)-derived elements. It does not apply any default formatting to the elements it contains. However, any property values set on a [Section](#) applies to its child elements. A section also enables you to programmatically iterate through its child collection. [Section](#) is used in a similar manner to the <DIV> tag in HTML.

In the example below, three paragraphs are defined under one [Section](#). The section has a [Background](#) property value of Red, therefore the background color of the paragraphs is also red.

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- By default, Section applies no formatting to elements contained
        within it. However, in this example, the section has a Background
        property value of "Red", therefore, the three paragraphs (the block)
        inside the section also have a red background. -->
    <Section Background="Red">
        <Paragraph>
            Paragraph 1
        </Paragraph>
        <Paragraph>
            Paragraph 2
        </Paragraph>
        <Paragraph>
            Paragraph 3
        </Paragraph>
    </Section>
</FlowDocument>
```

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class SectionExample : Page
    {
        public SectionExample()
        {

            // Create three paragraphs
            Paragraph myParagraph1 = new Paragraph(new Run("Paragraph 1"));
            Paragraph myParagraph2 = new Paragraph(new Run("Paragraph 2"));
            Paragraph myParagraph3 = new Paragraph(new Run("Paragraph 3"));

            // Create a Section and add the three paragraphs to it.
            Section mySection = new Section();
            mySection.Background = Brushes.Red;

            mySection.Blocks.Add(myParagraph1);
            mySection.Blocks.Add(myParagraph2);
            mySection.Blocks.Add(myParagraph3);

            // Create a FlowDocument and add the section to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(mySection);

            this.Content = myFlowDocument;
        }
    }
}
```

```

Imports System.Windows
Imports System.Windows.Media
Imports System.Windows.Controls
Imports System.Windows.Documents

Namespace SDKSample
    Partial Public Class SectionExample
        Inherits Page
        Public Sub New()

            ' Create three paragraphs
            Dim myParagraph1 As New Paragraph(New Run("Paragraph 1"))
            Dim myParagraph2 As New Paragraph(New Run("Paragraph 2"))
            Dim myParagraph3 As New Paragraph(New Run("Paragraph 3"))

            ' Create a Section and add the three paragraphs to it.
            Dim mySection As New Section()
            mySection.Background = Brushes.Red

            mySection.Blocks.Add(myParagraph1)
            mySection.Blocks.Add(myParagraph2)
            mySection.Blocks.Add(myParagraph3)

            ' Create a FlowDocument and add the section to it.
            Dim myFlowDocument As New FlowDocument()
            myFlowDocument.Blocks.Add(mySection)

            Me.Content = myFlowDocument
        End Sub
    End Class
End Namespace

```

## BlockUIContainer

[BlockUIContainer](#) enables [UIElement](#) elements (i.e. a [Button](#)) to be embedded in block-derived flow content. [InlineUIContainer](#) (see below) is used to embed [UIElement](#) elements in inline-derived flow content.

[BlockUIContainer](#) and [InlineUIContainer](#) are important because there is no other way to use a [UIElement](#) in flow content unless it is contained within one of these two elements.

The following example shows how to use the [BlockUIContainer](#) element to host [UIElement](#) objects within flow content.

```

<FlowDocument ColumnWidth="400">
  <Section Background="GhostWhite">
    <Paragraph>
      A UIElement element may be embedded directly in flow content
      by enclosing it in a BlockUIContainer element.
    </Paragraph>
    <BlockUIContainer>
      <Button>Click me!</Button>
    </BlockUIContainer>
    <Paragraph>
      The BlockUIContainer element may host no more than one top-level
      UIElement. However, other UIElements may be nested within the
      UIElement contained by an BlockUIContainer element. For example,
      a StackPanel can be used to host multiple UIElement elements within
      a BlockUIContainer element.
    </Paragraph>
    <BlockUIContainer>
      <StackPanel>
        <Label Foreground="Blue">Choose a value:</Label>
        <ComboBox>
          <ComboBoxItem IsSelected="True">a</ComboBoxItem>
          <ComboBoxItem>b</ComboBoxItem>
          <ComboBoxItem>c</ComboBoxItem>
        </ComboBox>
        <Label Foreground ="Red">Choose a value:</Label>
        <StackPanel>
          <RadioButton>x</RadioButton>
          <RadioButton>y</RadioButton>
          <RadioButton>z</RadioButton>
        </StackPanel>
        <Label>Enter a value:</Label>
        <TextBox>
          A text editor embedded in flow content.
        </TextBox>
      </StackPanel>
    </BlockUIContainer>
  </Section>
</FlowDocument>

```

The following figure shows how this example renders:

A UIElement element may be embedded directly in flow content by enclosing it in a BlockUIContainer element.

**Click me!**

The BlockUIContainer element may host no more than one top-level UIElement. However, other UIElements may be nested within the UIElement contained by an BlockUIContainer element. For example, a StackPanel can be used to host multiple UIElement elements within a BlockUIContainer element.

**Choose a value:**

a

**Choose a value:**

- x
- y
- z

**Enter a value:**

A text editor embedded in flow content.

## List

[List](#) is used to create a bulleted or numeric list. Set the [MarkerStyle](#) property to a [TextMarkerStyle](#) enumeration value to determine the style of the list. The example below shows how to create a simple list.

```

<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <List>
        <ListItem>
            <Paragraph>
                List Item 1
            </Paragraph>
        </ListItem>
        <ListItem>
            <Paragraph>
                List Item 2
            </Paragraph>
        </ListItem>
        <ListItem>
            <Paragraph>
                List Item 3
            </Paragraph>
        </ListItem>
    </List>
</FlowDocument>

```

```

using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class ListExample : Page
    {
        public ListExample()
        {

            // Create three paragraphs
            Paragraph myParagraph1 = new Paragraph(new Run("List Item 1"));
            Paragraph myParagraph2 = new Paragraph(new Run("List Item 2"));
            Paragraph myParagraph3 = new Paragraph(new Run("List Item 3"));

            // Create the ListItem elements for the List and add the
            // paragraphs to them.
            ListItem myListItem1 = new ListItem();
            myListItem1.Blocks.Add(myParagraph1);
            ListItem myListItem2 = new ListItem();
            myListItem2.Blocks.Add(myParagraph2);
            ListItem myListItem3 = new ListItem();
            myListItem3.Blocks.Add(myParagraph3);

            // Create a List and add the three ListItems to it.
            List myList = new List();

            myList.ListItems.Add(myListItem1);
            myList.ListItems.Add(myListItem2);
            myList.ListItems.Add(myListItem3);

            // Create a FlowDocument and add the section to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myList);

            this.Content = myFlowDocument;
        }
    }
}

```

```

Imports System.Windows
Imports System.Windows.Media
Imports System.Windows.Controls
Imports System.Windows.Documents

Namespace SDKSample
    Partial Public Class ListExample
        Inherits Page
        Public Sub New()

            ' Create three paragraphs
            Dim myParagraph1 As New Paragraph(New Run("List Item 1"))
            Dim myParagraph2 As New Paragraph(New Run("List Item 2"))
            Dim myParagraph3 As New Paragraph(New Run("List Item 3"))

            ' Create the ListItem elements for the List and add the
            ' paragraphs to them.
            Dim myListItem1 As New ListItem()
            myListItem1.Blocks.Add(myParagraph1)
            Dim myListItem2 As New ListItem()
            myListItem2.Blocks.Add(myParagraph2)
            Dim myListItem3 As New ListItem()
            myListItem3.Blocks.Add(myParagraph3)

            ' Create a List and add the three ListItems to it.
            Dim myList As New List()

            myList.ListItems.Add(myListItem1)
            myList.ListItems.Add(myListItem2)
            myList.ListItems.Add(myListItem3)

            ' Create a FlowDocument and add the section to it.
            Dim myFlowDocument As New FlowDocument()
            myFlowDocument.Blocks.Add(myList)

            Me.Content = myFlowDocument
        End Sub
    End Class
End Namespace

```

#### NOTE

List is the only flow element that uses the [ListItemCollection](#) to manage child elements.

## Table

**Table** is used to create a table. **Table** is similar to the **Grid** element but it has more capabilities and, therefore, requires greater resource overhead. Because **Grid** is a [UIElement](#), it cannot be used in flow content unless it is contained in a [BlockUIContainer](#) or [InlineUIContainer](#). For more information on **Table**, see [Table Overview](#).

## Inline-derived Classes

### Run

**Run** is used to contain unformatted text. You might expect **Run** objects to be used extensively in flow content. However, in markup, **Run** elements are not required to be used explicitly. **Run** is required to be used when creating or manipulating flow documents by using code. For example, in the markup below, the first [Paragraph](#) specifies the **Run** element explicitly while the second does not. Both paragraphs generate identical output.

```

<Paragraph>
  <Run>Paragraph that explicitly uses the Run element.</Run>
</Paragraph>

<Paragraph>
  This Paragraph omits the Run element in markup. It renders
  the same as a Paragraph with Run used explicitly.
</Paragraph>

```

#### NOTE

Starting in the .NET Framework 4, the [Text](#) property of the [Run](#) object is a dependency property. You can bind the [Text](#) property to a data source, such as a [TextBlock](#). The [Text](#) property fully supports one-way binding. The [Text](#) property also supports two-way binding, except for [RichTextBox](#). For an example, see [Run.Text](#).

## Span

[Span](#) groups other inline content elements together. No inherent rendering is applied to content within a [Span](#) element. However, elements that inherit from [Span](#) including [Hyperlink](#), [Bold](#), [Italic](#) and [Underline](#) do apply formatting to text.

Below is an example of a [Span](#) being used to contain inline content including text, a [Bold](#) element, and a [Button](#).

```

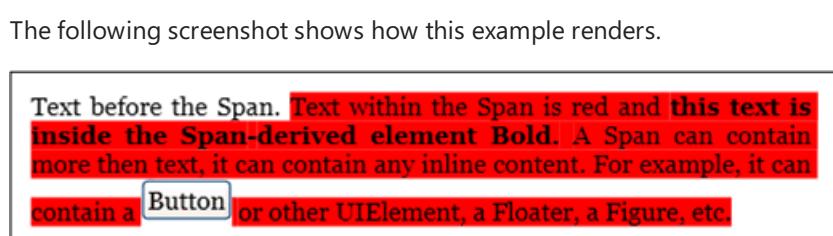
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Paragraph>
    Text before the Span. <Span Background="Red">Text within the Span is
    red and <Bold>this text is inside the Span-derived element Bold.</Bold>
    A Span can contain more than text, it can contain any inline content. For
    example, it can contain a
    <InlineUIContainer>
      <Button>Button</Button>
    </InlineUIContainer>
    or other UIElement, a Floater, a Figure, etc.</Span>
  </Paragraph>

</FlowDocument>

```

The following screenshot shows how this example renders.



Text before the Span. Text within the Span is red and this text is inside the Span-derived element Bold. A Span can contain more than text, it can contain any inline content. For example, it can contain a [Button](#) or other UIElement, a Floater, a Figure, etc.

## InlineUIContainer

[InlineUIContainer](#) enables [UIElement](#) elements (i.e. a control like [Button](#)) to be embedded in an [Inline](#) content element. This element is the inline equivalent to [BlockUIContainer](#) described above. Below is an example that uses [InlineUIContainer](#) to insert a [Button](#) inline in a [Paragraph](#).

```

<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Paragraph>
    Text to precede the button...

    <!-- Set the BaselineAlignment property to "Bottom"
        so that the Button aligns properly with the text. -->
    <InlineUIContainer BaselineAlignment="Bottom">
      <Button>Button</Button>
    </InlineUIContainer>
    Text to follow the button...
  </Paragraph>

</FlowDocument>

```

```

using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class InlineUIContainerExample : Page
    {
        public InlineUIContainerExample()
        {
            Run run1 = new Run(" Text to precede the button... ");
            Run run2 = new Run(" Text to follow the button... ");

            // Create a new button to be hosted in the paragraph.
            Button myButton = new Button();
            myButton.Content = "Click me!";

            // Create a new InlineUIContainer to contain the Button.
            InlineUIContainer myInlineUIContainer = new InlineUIContainer();

            // Set the BaselineAlignment property to "Bottom" so that the
            // Button aligns properly with the text.
            myInlineUIContainer.BaselineAlignment = BaselineAlignment.Bottom;

            // Assign the button as the UI container's child.
            myInlineUIContainer.Child = myButton;

            // Create the paragraph and add content to it.
            Paragraph myParagraph = new Paragraph();
            myParagraph.Inlines.Add(run1);
            myParagraph.Inlines.Add(myInlineUIContainer);
            myParagraph.Inlines.Add(run2);

            // Create a FlowDocument and add the paragraph to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myParagraph);

            this.Content = myFlowDocument;
        }
    }
}

```

```

Imports System.Windows
Imports System.Windows.Media
Imports System.Windows.Controls
Imports System.Windows.Documents

Namespace SDKSample
    Partial Public Class InlineUIContainerExample
        Inherits Page
        Public Sub New()
            Dim run1 As New Run(" Text to precede the button... ")
            Dim run2 As New Run(" Text to follow the button... ")

            ' Create a new button to be hosted in the paragraph.
            Dim myButton As New Button()
            myButton.Content = "Click me!"

            ' Create a new InlineUIContainer to contain the Button.
            Dim myInlineUIContainer As New InlineUIContainer()

            ' Set the BaselineAlignment property to "Bottom" so that the
            ' Button aligns properly with the text.
            myInlineUIContainer.BaselineAlignment = BaselineAlignment.Bottom

            ' Assign the button as the UI container's child.
            myInlineUIContainer.Child = myButton

            ' Create the paragraph and add content to it.
            Dim myParagraph As New Paragraph()
            myParagraph.Inlines.Add(run1)
            myParagraph.Inlines.Add(myInlineUIContainer)
            myParagraph.Inlines.Add(run2)

            ' Create a FlowDocument and add the paragraph to it.
            Dim myFlowDocument As New FlowDocument()
            myFlowDocument.Blocks.Add(myParagraph)

            Me.Content = myFlowDocument
        End Sub
    End Class
End Namespace

```

#### **NOTE**

[InlineUIContainer](#) does not need to be used explicitly in markup. If you omit it, an [InlineUIContainer](#) will be created anyway when the code is compiled.

## **Figure and Floater**

[Figure](#) and [Floater](#) are used to embed content in Flow Documents with placement properties that can be customized independent of the primary content flow. [Figure](#) or [Floater](#) elements are often used to highlight or accentuate portions of content, to host supporting images or other content within the main content flow, or to inject loosely related content such as advertisements.

The following example shows how to embed a [Figure](#) into a paragraph of text.

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Paragraph>
    <Figure
      Width="300" Height="100"
      Background="GhostWhite" HorizontalAnchor="PageLeft" >
      <Paragraph FontStyle="Italic" Background="Beige" Foreground="DarkGreen" >
        A Figure embeds content into flow content with placement properties
        that can be customized independently from the primary content flow
      </Paragraph>
    </Figure>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy
    nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi
    enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis
    nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
  </Paragraph>

</FlowDocument>
```

```

using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class FigureExample : Page
    {
        public FigureExample()
        {

            // Create strings to use as content.
            string strFigure = "A Figure embeds content into flow content with" +
                " placement properties that can be customized" +
                " independently from the primary content flow";
            string strOther = "Lorem ipsum dolor sit amet, consectetur adipiscing" +
                " elit, sed diam nonummy nibh euismod tincidunt ut laoreet" +
                " dolore magna aliquam erat volutpat. Ut wisi enim ad" +
                " minim veniam, quis nostrud exerci tation ullamcorper" +
                " suscipit lobortis nisl ut aliquip ex ea commodo consequat." +
                " Duis autem vel eum iriure.";

            // Create a Figure and assign content and layout properties to it.
            Figure myFigure = new Figure();
            myFigure.Width = new FigureLength(300);
            myFigure.Height = new FigureLength(100);
            myFigure.Background = Brushes.GhostWhite;
            myFigure.HorizontalAnchor = FigureHorizontalAnchor.PageLeft;
            Paragraph myFigureParagraph = new Paragraph(new Run(strFigure));
            myFigureParagraph.FontStyle = FontStyles.Italic;
            myFigureParagraph.Background = Brushes.Beige;
            myFigureParagraph.Foreground = Brushes.DarkGreen;
            myFigure.Blocks.Add(myFigureParagraph);

            // Create the paragraph and add content to it.
            Paragraph myParagraph = new Paragraph();
            myParagraph.Inlines.Add(myFigure);
            myParagraph.Inlines.Add(new Run(strOther));

            // Create a FlowDocument and add the paragraph to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myParagraph);

            this.Content = myFlowDocument;
        }
    }
}

```

```

Imports System.Windows
Imports System.Windows.Media
Imports System.Windows.Controls
Imports System.Windows.Documents

Namespace SDKSample
    Partial Public Class FigureExample
        Inherits Page
        Public Sub New()

            ' Create strings to use as content.
            Dim strFigure As String = "A Figure embeds content into flow content with" & " placement
properties that can be customized" & " independently from the primary content flow"
            Dim strOther As String = "Lorem ipsum dolor sit amet, consectetuer adipiscing" & " elit, sed
diam nonummy nibh euismod tincidunt ut laoreet" & " dolore magna aliquam erat volutpat. Ut wisi enim ad" &
minim veniam, quis nostrud exerci tation ullamcorper" & " suscipit lobortis nisl ut aliquip ex ea commodo
consequat." & " Duis autem vel eum iriure."

            ' Create a Figure and assign content and layout properties to it.
            Dim myFigure As New Figure()
            myFigure.Width = New FigureLength(300)
            myFigure.Height = New FigureLength(100)
            myFigure.Background = Brushes.GhostWhite
            myFigure.HorizontalAnchor = FigureHorizontalAnchor.PageLeft
            Dim myFigureParagraph As New Paragraph(New Run(strFigure))
            myFigureParagraph.FontStyle = FontStyles.Italic
            myFigureParagraph.Background = Brushes.Beige
            myFigureParagraph.Foreground = Brushes.DarkGreen
            myFigure.Blocks.Add(myFigureParagraph)

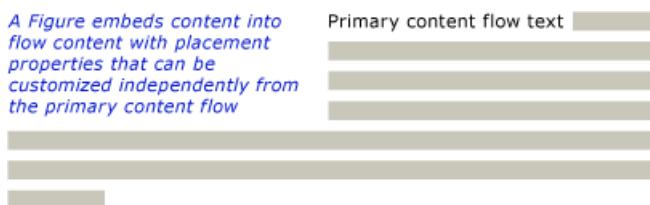
            ' Create the paragraph and add content to it.
            Dim myParagraph As New Paragraph()
            myParagraph.Inlines.Add(myFigure)
            myParagraph.Inlines.Add(New Run(strOther))

            ' Create a FlowDocument and add the paragraph to it.
            Dim myFlowDocument As New FlowDocument()
            myFlowDocument.Blocks.Add(myParagraph)

            Me.Content = myFlowDocument
        End Sub
    End Class
End Namespace

```

The following illustration shows how this example renders.



**Figure** and **Floater** differ in several ways and are used for different scenarios.

### Figure:

- Can be positioned: You can set its horizontal and vertical anchors to dock it relative to the page, content, column or paragraph. You can also use its [HorizontalOffset](#) and [VerticalOffset](#) properties to specify arbitrary offsets.
- Is sizable to more than one column: You can set **Figure** height and width to multiples of page, content or column height or width. Note that in the case of page and content, multiples greater than 1 are not

allowed. For example, you can set the width of a [Figure](#) to be "0.5 page" or "0.25 content" or "2 Column". You can also set height and width to absolute pixel values.

- Does not paginate: If the content inside a [Figure](#) does not fit inside the [Figure](#), it will render whatever content does fit and the remaining content is lost

### Floater:

- Cannot be positioned and will render wherever space can be made available for it. You cannot set the offset or anchor a [Floater](#).
- Cannot be sized to more than one column: By default, [Floater](#) sizes at one column. It has a [Width](#) property that can be set to an absolute pixel value, but if this value is greater than one column width it is ignored and the floater is sized at one column. You can size it to less than one column by setting the correct pixel width, but sizing is not column-relative, so "0.5Column" is not a valid expression for [Floater](#) width. [Floater](#) has no height property and its height cannot be set, its height depends on the content
- [Floater](#) paginates: If its content at its specified width extends to more than 1 column height, floater breaks and paginates to the next column, the next page, etc.

[Figure](#) is a good place to put standalone content where you want to control the size and positioning, and are confident that the content will fit in the specified size. [Floater](#) is a good place to put more free-flowing content that flows similar to the main page content, but is separated from it.

### LineBreak

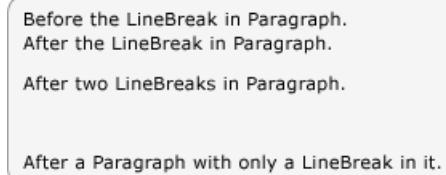
[LineBreak](#) causes a line break to occur in flow content. The following example demonstrates the use of [LineBreak](#).

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Paragraph>
    Before the LineBreak in Paragraph.
    <LineBreak />
    After the LineBreak in Paragraph.
    <LineBreak/><LineBreak/>
    After two LineBreaks in Paragraph.
  </Paragraph>

  <Paragraph>
    <LineBreak/>
  </Paragraph>

  <Paragraph>
    After a Paragraph with only a LineBreak in it.
  </Paragraph>
</FlowDocument>
```

The following screenshot shows how this example renders.



### Flow Collection Elements

In many of the examples above, the [BlockCollection](#) and [InlineCollection](#) are used to construct flow content programmatically. For example, to add elements to a [Paragraph](#), you can use the syntax:

```
myParagraph.Inlines.Add(new Run("Some text"));
```

This adds a [Run](#) to the [InlineCollection](#) of the [Paragraph](#). This is the same as the implicit [Run](#) found inside a [Paragraph](#) in markup:

```
<Paragraph>
  Some Text
</Paragraph>
```

As an example of using the [BlockCollection](#), the following example creates a new [Section](#) and then uses the [Add](#) method to add a new [Paragraph](#) to the [Section](#) contents.

```
Section secx = new Section();
secx.Blocks.Add(new Paragraph(new Run("A bit of text content...")));
```

```
Dim secx As New Section()
secx.Blocks.Add(New Paragraph(New Run("A bit of text content...")))
```

In addition to adding items to a flow collection, you can remove items as well. The following example deletes the last [Inline](#) element in the [Span](#).

```
spanx.Inlines.Remove(spanx.Inlines.LastInline);
```

```
spanx.Inlines.Remove(spanx.Inlines.LastInline)
```

The following example clears all of the contents ([Inline](#) elements) from the [Span](#).

```
spanx.Inlines.Clear();
```

```
spanx.Inlines.Clear()
```

When working with flow content programmatically, you will likely make extensive use of these collections.

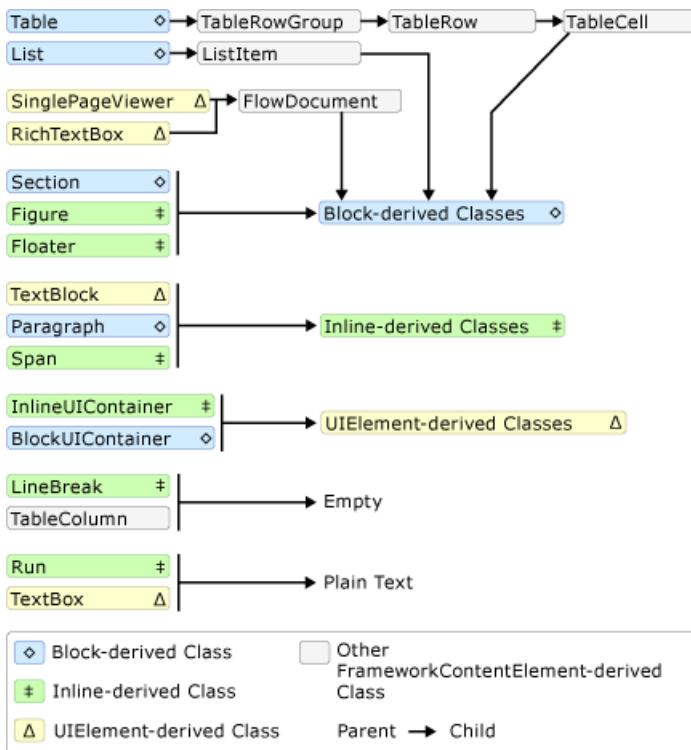
Whether a flow element uses an [InlineCollection](#) ([Inlines](#)) or [BlockCollection](#) ([Blocks](#)) to contain its child elements depends on what type of child elements ([Block](#) or [Inline](#)) can be contained by the parent. Containment rules for flow content elements are summarized in the content schema in the next section.

#### NOTE

There is a third type of collection used with flow content, the [ListItemCollection](#), but this collection is only used with a [List](#). In addition, there are several collections used with [Table](#). See [Table Overview](#) for more information.

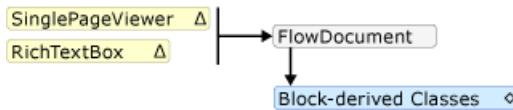
## Content Schema

Given the number of different flow content elements, it can be overwhelming to keep track of what type of child elements an element can contain. The diagram below summarizes the containment rules for flow elements. The arrows represent the possible parent/child relationships.



As can be seen from the diagram above, the children allowed for an element are not necessarily determined by whether it is a **Block** element or an **Inline** element. For example, a **Span** (an **Inline** element) can only have **Inline** child elements while a **Figure** (also an **Inline** element) can only have **Block** child elements. Therefore, a diagram is useful for quickly determining what element can be contained in another. As an example, let's use the diagram to determine how to construct the flow content of a [RichTextBox](#).

**1.** A [RichTextBox](#) must contain a [FlowDocument](#) which in turn must contain a **Block**-derived object. Below is the corresponding segment from the diagram above.

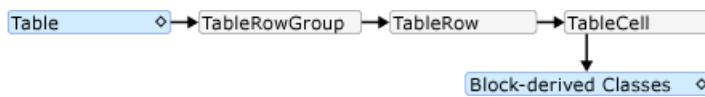


Thus far, this is what the markup might look like.

```

<RichTextBox>
  <FlowDocument>
    <!-- One or more Block-derived object... -->
  </FlowDocument>
</RichTextBox>
  
```

**2.** According to the diagram, there are several **Block** elements to choose from including [Paragraph](#), [Section](#), [Table](#), [List](#), and [BlockUIContainer](#) (see **Block**-derived classes above). Let's say we want a [Table](#). According to the diagram above, a [Table](#) contains a [TableRowGroup](#) containing [TableRow](#) elements, which contain [TableCell](#) elements which contain a **Block**-derived object. Below is the corresponding segment for [Table](#) taken from the diagram above.



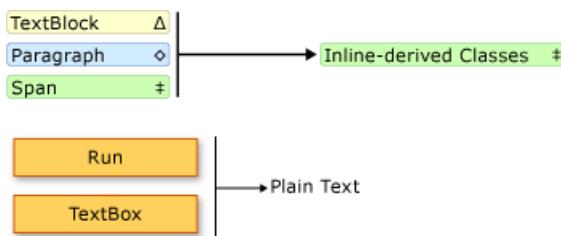
Below is the corresponding markup.

```

<RichTextBox>
  <FlowDocument>
    <Table>
      <TableRowGroup>
        <TableRow>
          <TableCell>
            <!-- One or more Block-derived object... -->
          </TableCell>
        </TableRow>
      </TableRowGroup>
    </Table>
  </FlowDocument>
</RichTextBox>

```

**3.** Again, one or more **Block** elements are required underneath a **TableCell**. To make it simple, let's place some text inside the cell. We can do this using a **Paragraph** with a **Run** element. Below is the corresponding segments from the diagram showing that a **Paragraph** can take an **Inline** element and that a **Run** (an **Inline** element) can only take plain text.



Below is the entire example in markup.

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <RichTextBox>
    <FlowDocument>

      <!-- Normally a table would have multiple rows and multiple
           cells but this code is for demonstration purposes.-->
      <Table>
        <TableRowGroup>
          <TableRow>
            <TableCell>
              <Paragraph>

                <!-- The schema does not actually require
                     explicit use of the Run tag in markup. It
                     is only included here for clarity. -->
                <Run>Paragraph in a Table Cell.</Run>
              </Paragraph>
            </TableCell>
          </TableRow>
        </TableRowGroup>
      </Table>

    </FlowDocument>
  </RichTextBox>
</Page>

```

## Customizing Text

Usually text is the most prevalent type of content in a flow document. Although the objects introduced above can be used to control most aspects of how text is rendered, there are some other methods for customizing text that is covered in this section.

## Text Decorations

Text decorations allow you to apply the underline, overline, baseline, and strikethrough effects to text (see pictures below). These decorations are added using the [TextDecorations](#) property that is exposed by a number of objects including [Inline](#), [Paragraph](#), [TextBlock](#), and [TextBox](#).

The following example shows how to set the [TextDecorations](#) property of a [Paragraph](#).

```
<FlowDocument ColumnWidth="200">
    <Paragraph TextDecorations="Strikethrough">
        This text will render with the strikethrough effect.
    </Paragraph>
</FlowDocument>
```

```
Paragraph parx = new Paragraph(new Run("This text will render with the strikethrough effect."));
parx.TextDecorations = TextDecorations.Strikethrough;
```

```
Dim parx As New Paragraph(New Run("This text will render with the strikethrough effect."))
parx.TextDecorations = TextDecorations.Strikethrough
```

The following figure shows how this example renders.

This text will render with the default strikethrough effect.

The following figures show how the **Overline**, **Baseline**, and **Underline** decorations render, respectively.

This text will render with the default overline effect.

This text will render with the default baseline effect.

This text will render with the default underline effect.

## Typography

The [Typography](#) property is exposed by most flow-related content including [TextElement](#), [FlowDocument](#), [TextBlock](#), and [TextBox](#). This property is used to control typographical characteristics/variations of text (i.e. small or large caps, making superscripts and subscripts, etc).

The following example shows how to set the [Typography](#) attribute, using [Paragraph](#) as the example element.

```

<Paragraph
    TextAlignment="Left"
    FontSize="18"
    FontFamily="Palatino Linotype"
    Typography.NumeralStyle="OldStyle"
    Typography.Fraction="Stacked"
    Typography.Variants="Inferior"
>
<Run>
    This text has some altered typography characteristics. Note
    that use of an open type font is necessary for most typographic
    properties to be effective.
</Run>
<LineBreak/><LineBreak/>
<Run>
    0123456789 10 11 12 13
</Run>
<LineBreak/><LineBreak/>
<Run>
    1/2 2/3 3/4
</Run>
</Paragraph>

```

The following figure shows how this example renders.

This text has some altered typography characteristics. Note that use of an open type font is necessary for most typographic properties to be effective.

0123456789 10 11 12 13

$\frac{1}{2}$   $\frac{2}{3}$   $\frac{3}{4}$

In contrast, the following figure shows how a similar example with default typographic properties renders.

This text has some altered typography characteristics. Note that use of an open type font is necessary for most typographic properties to be effective.

0123456789 10 11 12 13

1/2 2/3 3/4

The following example shows how to set the [Typography](#) property programmatically.

```

Paragraph par = new Paragraph();

Run runText = new Run(
    "This text has some altered typography characteristics. Note" +
    "that use of an open type font is necessary for most typographic" +
    "properties to be effective.");
Run runNumerals = new Run("0123456789 10 11 12 13");
Run runFractions = new Run("1/2 2/3 3/4");

par.Inlines.Add(runText);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runNumerals);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runFractions);

par.TextAlignment = TextAlignment.Left;
par.FontSize = 18;
par.FontFamily = new FontFamily("Palatino Linotype");

par.Typography.NumeralStyle = FontNumeralStyle.OldStyle;
par.Typography.Fraction = FontFraction.Stacked;
par.Typography.Variants = FontVariants.Inferior;

```

```

Dim par As New Paragraph()

Dim runText As New Run("This text has some altered typography characteristics. Note" & "that use of an open
type font is necessary for most typographic" & "properties to be effective.")
Dim runNumerals As New Run("0123456789 10 11 12 13")
Dim runFractions As New Run("1/2 2/3 3/4")

par.Inlines.Add(runText)
par.Inlines.Add(New LineBreak())
par.Inlines.Add(New LineBreak())
par.Inlines.Add(runNumerals)
par.Inlines.Add(New LineBreak())
par.Inlines.Add(New LineBreak())
par.Inlines.Add(runFractions)

par.TextAlignment = TextAlignment.Left
par.FontSize = 18
par.FontFamily = New FontFamily("Palatino Linotype")

par.Typography.NumeralStyle = FontNumeralStyle.OldStyle
par.Typography.Fraction = FontFraction.Stacked
par.Typography.Variants = FontVariants.Inferior

```

See [Typography in WPF](#) for more information on typography.

## See also

- [Text](#)
- [Typography in WPF](#)
- [How-to Topics](#)
- [TextElement Content Model Overview](#)
- [RichTextBox Overview](#)
- [Documents in WPF](#)
- [Table Overview](#)
- [Annotations Overview](#)



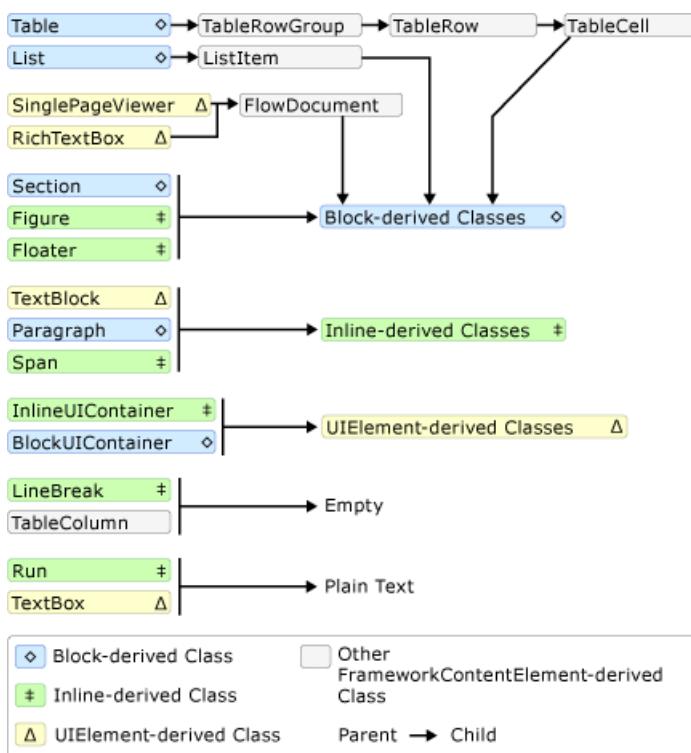
# TextElement Content Model Overview

11/7/2019 • 3 minutes to read • [Edit Online](#)

This content model overview describes the supported content for a [TextElement](#). The [Paragraph](#) class is a type of [TextElement](#). A content model describes what objects/elements can be contained in others. This overview summarizes the content model used for objects derived from [TextElement](#). For more information, see [Flow Document Overview](#).

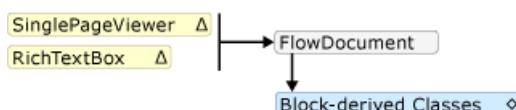
## Content Model Diagram

The following diagram summarizes the content model for classes derived from [TextElement](#) as well as how other non- [TextElement](#) classes fit into this model.



As can be seen from the preceding diagram, the children allowed for an element are not necessarily determined by whether a class is derived from the [Block](#) class or an [Inline](#) class. For example, a [Span](#) (an [Inline](#)-derived class) can only have [Inline](#) child elements, but a [Figure](#) (also an [Inline](#)-derived class) can only have [Block](#) child elements. Therefore, a diagram is useful for quickly determining what element can be contained in another. As an example, let's use the diagram to determine how to construct the flow content of a [RichTextBox](#).

1. A [RichTextBox](#) must contain a [FlowDocument](#) which in turn must contain a [Block](#)-derived object. The following is the corresponding segment from the preceding diagram.



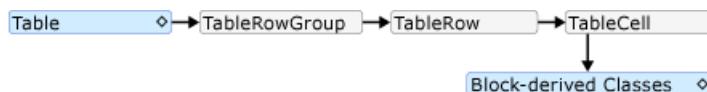
Thus far, this is what the markup might look like.

```

<RichTextBox>
  <FlowDocument>
    <!-- One or more Block-derived object... -->
  </FlowDocument>
</RichTextBox>

```

2. According to the diagram, there are several **Block** elements to choose from including **Paragraph**, **Section**, **Table**, **List**, and **BlockUIContainer** (see Block-derived classes in the preceding diagram). Let's say we want a **Table**. According to the preceding diagram, a **Table** contains a **TableRowGroup** containing **TableRow** elements, which contain **TableCell** elements which contain a **Block**-derived object. The following is the corresponding segment for **Table** taken from the preceding diagram.



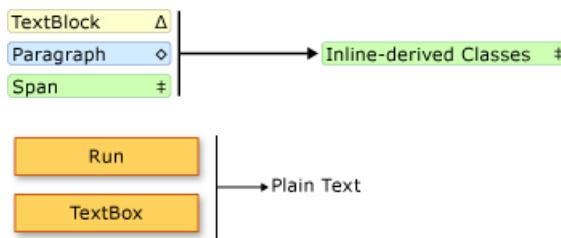
The following is the corresponding markup.

```

<RichTextBox>
  <FlowDocument>
    <Table>
      <TableRowGroup>
        <TableRow>
          <TableCell>
            <!-- One or more Block-derived object... -->
          </TableCell>
        </TableRow>
      </TableRowGroup>
    </Table>
  </FlowDocument>
</RichTextBox>

```

3. Again, one or more **Block** elements are required underneath a **TableCell**. To make it simple, let's place some text inside the cell. We can do this using a **Paragraph** with a **Run** element. The following is the corresponding segments from the diagram showing that a **Paragraph** can take an **Inline** element and that a **Run** (an **Inline** element) can only take plain text.



The following is the entire example in markup.

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <RichTextBox>
        <FlowDocument>

            <!-- Normally a table would have multiple rows and multiple
                 cells but this code is for demonstration purposes.-->
            <Table>
                <TableRowGroup>
                    <TableRow>
                        <TableCell>
                            <Paragraph>

                                <!-- The schema does not actually require
                                     explicit use of the Run tag in markup. It
                                     is only included here for clarity. -->
                                <Run>Paragraph in a Table Cell.</Run>
                            </Paragraph>
                        </TableCell>
                    </TableRow>
                </TableRowGroup>
            </Table>

        </FlowDocument>
    </RichTextBox>
</Page>

```

## Working with TextElement Content Programmatically

The contents of a [TextElement](#) is made up by collections and so programmatically manipulating the contents of [TextElement](#) objects is done by working with these collections. There are three different collections used by [TextElement](#) -derived classes:

- [InlineCollection](#): Represents a collection of [Inline](#) elements. [InlineCollection](#) defines the allowable child content of the [Paragraph](#), [Span](#), and [TextBlock](#) elements.
- [BlockCollection](#): Represents a collection of [Block](#) elements. [BlockCollection](#) defines the allowable child content of the [FlowDocument](#), [Section](#), [ListItem](#), [TableCell](#), [Floater](#), and [Figure](#) elements.
- [ListCollection](#): A flow content element that represents a particular content item in an ordered or unordered [List](#).

You can manipulate (add or remove items) from these collections using the respective properties of [Inlines](#), [Blocks](#), and [ListItems](#). The following examples show how to manipulate the contents of a [Span](#) using the [Inlines](#) property.

### NOTE

Table uses several collections to manipulate its contents, but they are not covered here. For more information, see [Table Overview](#).

The following example creates a new [Span](#) object, and then uses the [Add](#) method to add two text runs as content children of the [Span](#).

```

Span spanx = new Span();
spanx.Inlines.Add(new Run("A bit of text content..."));
spanx.Inlines.Add(new Run("A bit more text content..."));

```

```
Dim spanx As New Span()
spanx.Inlines.Add(New Run("A bit of text content..."))
spanx.Inlines.Add(New Run("A bit more text content..."))
```

The following example creates a new [Run](#) element and inserts it at the beginning of the [Span](#).

```
Run runx = new Run("Text to insert...");
spanx.Inlines.InsertBefore(spanx.Inlines.FirstInline, runx);
```

```
Dim runx As New Run("Text to insert...")
spanx.Inlines.InsertBefore(spanx.Inlines.FirstInline, runx)
```

The following example deletes the last [Inline](#) element in the [Span](#).

```
spanx.Inlines.Remove(spanx.Inlines.LastInline);
```

```
spanx.Inlines.Remove(spanx.Inlines.LastInline)
```

The following example clears all of the contents ([Inline](#) elements) from the [Span](#).

```
spanx.Inlines.Clear();
```

```
spanx.Inlines.Clear()
```

## Types That Share This Content Model

The following types inherit from the [TextElement](#) class and may be used to display the content described in this overview.

[Bold](#), [Figure](#), [Floater](#), [Hyperlink](#), [InlineUIContainer](#), [Italic](#), [LineBreak](#), [List](#), [ListItem](#), [Paragraph](#), [Run](#), [Section](#), [Span](#), [Table](#), [Underline](#).

Note that this list only includes nonabstract types distributed with the Windows SDK. You may use other types that inherit from [TextElement](#).

## Types That Can Contain TextElement Objects

See [WPF Content Model](#).

## See also

- [Manipulate a FlowDocument through the Blocks Property](#)
- [Manipulate Flow Content Elements through the Blocks Property](#)
- [Manipulate a FlowDocument through the Blocks Property](#)
- [Manipulate a Table's Columns through the Columns Property](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)

# Table Overview

10/7/2019 • 9 minutes to read • [Edit Online](#)

**Table** is a block level element that supports grid-based presentation of Flow document content. The flexibility of this element makes it very useful, but also makes it more complicated to understand and use correctly.

This topic contains the following sections.

- [Table Basics](#)
- [How is Table Different then Grid?](#)
- [Basic Table Structure](#)
- [Table Containment](#)
- [Row Groupings](#)
- [Background Rendering Precedence](#)
- [Spanning Rows or Columns](#)
- [Building a Table With Code](#)
- [Related Topics]

## Table Basics

### How is Table Different then Grid?

**Table** and **Grid** share some common functionality, but each is best suited for different scenarios. A **Table** is designed for use within flow content (see [Flow Document Overview](#) for more information on flow content). Grids are best used inside of forms (basically anywhere outside of flow content). Within a [FlowDocument](#), **Table** supports flow content behaviors like pagination, column reflow, and content selection while a **Grid** does not. A **Grid** on the other hand is best used outside of a [FlowDocument](#) for many reasons including **Grid** adds elements based on a row and column index, **Table** does not. The **Grid** element allows layering of child content, allowing more than one element to exist within a single "cell." **Table** does not support layering. Child elements of a **Grid** can be absolutely positioned relative to the area of their "cell" boundaries. **Table** does not support this feature. Finally, a **Grid** requires less resources than a **Table** so consider using a **Grid** to improve performance.

### Basic Table Structure

**Table** provides a grid-based presentation consisting of columns (represented by  **TableColumn** elements) and rows (represented by  **TableRow** elements).  **TableColumn** elements do not host content; they simply define columns and characteristics of columns.  **TableRow** elements must be hosted in a  **TableRowGroup** element, which defines a grouping of rows for the table.  **TableCell** elements, which contain the actual content to be presented by the table, must be hosted in a  **TableRow** element.  **TableCell** may only contain elements that derive from  **Block**. Valid child elements for a  **TableCell** include.

- [BlockUIContainer](#)
- [List](#)
- [Paragraph](#)
- [Section](#)

- [Table](#)

**NOTE**

[TableCell](#) elements may not directly host text content. For more information about the containment rules for flow content elements like [TableCell](#), see [Flow Document Overview](#).

**NOTE**

[Table](#) is similar to the [Grid](#) element but has more capabilities and, therefore, requires greater resource overhead.

The following example defines a simple 2 x 3 table with XAML.

```

<!--
Table is a Block element, and as such must be hosted in a container
for Block elements. FlowDocument provides such a container.
-->
<FlowDocument>
  <Table>
    <!--
      This table has 3 columns, each described by a TableColumn
      element nested in a Table.Columns collection element.
    -->
    <Table.Columns>
      <TableColumn />
      <TableColumn />
      <TableColumn />
    </Table.Columns>
    <!--
      This table includes a single TableRowGroup which hosts 2 rows,
      each described by a TableRow element.
    -->
    <TableRowGroup>
      <!--
        Each of the 2 TableRow elements hosts 3 cells, described by
        TableCell elements.
      -->
      <TableRow>
        <TableCell>
          <!--
            TableCell elements may only host elements derived from Block.
            In this example, Paragraph elements serve as the ultimate content
            containers for the cells in this table.
          -->
          <Paragraph>Cell at Row 1 Column 1</Paragraph>
        </TableCell>
        <TableCell>
          <Paragraph>Cell at Row 1 Column 2</Paragraph>
        </TableCell>
        <TableCell>
          <Paragraph>Cell at Row 1 Column 3</Paragraph>
        </TableCell>
      </TableRow>
      <TableRow>
        <TableCell>
          <Paragraph>Cell at Row 2 Column 1</Paragraph>
        </TableCell>
        <TableCell>
          <Paragraph>Cell at Row 2 Column 2</Paragraph>
        </TableCell>
        <TableCell>
          <Paragraph>Cell at Row 2 Column 3</Paragraph>
        </TableCell>
      </TableRow>
    </TableRowGroup>
  </Table>
</FlowDocument>

```

The following figure shows how this example renders.

Cell at Row 1 Column 1	Cell at Row 1 Column 2	Cell at Row 1 Column 3
Cell at Row 2 Column 1	Cell at Row 2 Column 2	Cell at Row 2 Column 3

## Table Containment

[Table](#) derives from the [Block](#) element, and adheres to the common rules for [Block](#) level elements. A [Table](#) element may be contained by any of the following elements:

- [FlowDocument](#)
- [TableCell](#)
- [ListBoxItem](#)
- [ListViewItem](#)
- [Section](#)
- [Floater](#)
- [Figure](#)

## Row Groupings

The [TableRowGroup](#) element provides a way to arbitrarily group rows within a table; every row in a table must belong to a row grouping. Rows within a row group often share a common intent, and may be styled as a group. A common use for row groupings is to separate special-purpose rows, such as a title, header, and footer rows, from the primary content contained by the table.

The following example uses XAML to define a table with styled header and footer rows.

```

<Table>
  <Table.Resources>
    <!-- Style for header/footer rows. -->
    <Style x:Key="headerFooterRowStyle" TargetType="{x:Type TableRowGroup}">
      <Setter Property="FontWeight" Value="DemiBold"/>
      <Setter Property="FontSize" Value="16"/>
      <Setter Property="Background" Value="LightGray"/>
    </Style>

    <!-- Style for data rows. -->
    <Style x:Key="dataRowStyle" TargetType="{x:Type TableRowGroup}">
      <Setter Property="FontSize" Value="12"/>
      <Setter Property="FontStyle" Value="Italic"/>
    </Style>
  </Table.Resources>

  <Table.Columns>
    <TableColumn/> <TableColumn/> <TableColumn/> <TableColumn/>
  </Table.Columns>

  <!-- This TableRowGroup hosts a header row for the table. -->
  <TableRowGroup Style="{StaticResource headerFooterRowStyle}">
    <TableRow>
      <TableCell/>
      <TableCell><Paragraph>Gizmos</Paragraph></TableCell>
      <TableCell><Paragraph>Thingamajigs</Paragraph></TableCell>
      <TableCell><Paragraph>Doohickies</Paragraph></TableCell>
    </TableRow>
  </TableRowGroup>

  <!-- This TableRowGroup hosts the main data rows for the table. -->
  <TableRowGroup Style="{StaticResource dataRowStyle}">
    <TableRow>
      <TableCell><Paragraph Foreground="Blue">Blue</Paragraph></TableCell>
      <TableCell><Paragraph>1</Paragraph></TableCell>
      <TableCell><Paragraph>2</Paragraph></TableCell>
      <TableCell><Paragraph>3</Paragraph></TableCell>
    </TableRow>
    <TableRow>
      <TableCell><Paragraph Foreground="Red">Red</Paragraph></TableCell>
      <TableCell><Paragraph>1</Paragraph></TableCell>
      <TableCell><Paragraph>2</Paragraph></TableCell>
      <TableCell><Paragraph>3</Paragraph></TableCell>
    </TableRow>
    <TableRow>
      <TableCell><Paragraph Foreground="Green">Green</Paragraph></TableCell>
      <TableCell><Paragraph>1</Paragraph></TableCell>
      <TableCell><Paragraph>2</Paragraph></TableCell>
      <TableCell><Paragraph>3</Paragraph></TableCell>
    </TableRow>
  </TableRowGroup>

  <!-- This TableRowGroup hosts a footer row for the table. -->
  <TableRowGroup Style="{StaticResource headerFooterRowStyle}">
    <TableRow>
      <TableCell><Paragraph>Totals</Paragraph></TableCell>
      <TableCell><Paragraph>3</Paragraph></TableCell>
      <TableCell><Paragraph>6</Paragraph></TableCell>
      <TableCell>
        <Table></Table>
      </TableCell>
    </TableRow>
  </TableRowGroup>
</Table>

```

The following figure shows how this example renders.

	<b>Gizmos</b>	<b>Thingamajigs</b>	<b>Doohickies</b>
<i>Blue</i>	1	2	3
<i>Red</i>	1	2	3
<i>Green</i>	1	2	3
<b>Totals</b>	<b>3</b>	<b>6</b>	<b>9</b>

## Background Rendering Precedence

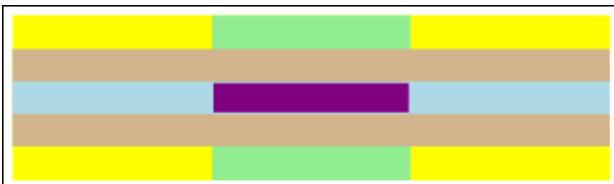
Table elements render in the following order (z-order from lowest to highest). This order cannot be changed. For example, there is no "Z-order" property for these elements that you can use to override this established order.

1. [Table](#)
2. [TableColumn](#)
3. [TableRowGroup](#)
4. [TableRow](#)
5. [TableCell](#)

Consider the following example, which defines background colors for each of these elements within a table.

```
<Table Background="Yellow">
  <Table.Columns>
    < TableColumn />
    < TableColumn Background="LightGreen" />
    < TableColumn />
  </Table.Columns>
  < TableRowGroup >
    < TableRow >
      < TableCell />< TableCell />< TableCell />
    </ TableRow >
  </ TableRowGroup >
  < TableRowGroup Background="Tan" >
    < TableRow >
      < TableCell />< TableCell />< TableCell />
    </ TableRow >
    < TableRow Background="LightBlue" >
      < TableCell />< TableCell Background="Purple" />< TableCell />
    </ TableRow >
    < TableRow >
      < TableCell />< TableCell />< TableCell />
    </ TableRow >
  </ TableRowGroup >
  < TableRowGroup >
    < TableRow >
      < TableCell />< TableCell />< TableCell />
    </ TableRow >
  </ TableRowGroup >
</Table>
```

The following figure shows how this example renders (showing background colors only).



## Spanning Rows or Columns

Table cells may be configured to span multiple rows or columns by using the [RowSpan](#) or [ColumnSpan](#) attributes, respectively.

Consider the following example, in which a cell spans three columns.

```
<Table>
  <Table.Columns>
    <TableColumn/>
    <TableColumn/>
    <TableColumn/>
  </Table.Columns>

  <TableRowGroup>
    <TableRow>
      <TableCell ColumnSpan="3" Background="Cyan">
        <Paragraph>This cell spans all three columns.</Paragraph>
      </TableCell>
    </TableRow>
    <TableRow>
      <TableCell Background="LightGray"><Paragraph>Cell 1</Paragraph></TableCell>
      <TableCell Background="LightGray"><Paragraph>Cell 2</Paragraph></TableCell>
      <TableCell Background="LightGray"><Paragraph>Cell 3</Paragraph></TableCell>
    </TableRow>
  </TableRowGroup>
</Table>
```

The following figure shows how this example renders.

This cell spans all three columns.		
Cell 1	Cell 2	Cell 3

## Building a Table With Code

The following examples show how to programmatically create a [Table](#) and populate it with content. The contents of the table are apportioned into five rows (represented by [TableRow](#) objects contained in a [RowGroups](#) object) and six columns (represented by [TableColumn](#) objects). The rows are used for different presentation purposes, including a title row intended to title the entire table, a header row to describe the columns of data in the table, and a footer row with summary information. Note that the notion of "title", "header", and "footer" rows are not inherent to the table; these are simply rows with different characteristics. Table cells contain the actual content, which can be comprised of text, images, or nearly any other user interface (UI) element.

First, a [FlowDocument](#) is created to host the [Table](#), and a new [Table](#) is created and added to the contents of the [FlowDocument](#).

```
// Create the parent FlowDocument...
flowDoc = new FlowDocument();

// Create the Table...
table1 = new Table();
// ...and add it to the FlowDocument Blocks collection.
flowDoc.Blocks.Add(table1);

// Set some global formatting properties for the table.
table1.CellSpacing = 10;
table1.Background = Brushes.White;
```

```

' Create the parent FlowDocument...
flowDoc = New FlowDocument()

' Create the Table...
table1 = New Table()

' ...and add it to the FlowDocument Blocks collection.
flowDoc.Blocks.Add(table1)

' Set some global formatting properties for the table.
table1.CellSpacing = 10
table1.Background = Brushes.White

```

Next, six [TableColumn](#) objects are created and added to the table's [Columns](#) collection, with some formatting applied.

#### **NOTE**

Note that the table's [Columns](#) collection uses standard zero-based indexing.

```

// Create 6 columns and add them to the table's Columns collection.
int numberofColumns = 6;
for (int x = 0; x < numberofColumns; x++)
{
    table1.Columns.Add(new TableColumn());

    // Set alternating background colors for the middle columns.
    if(x%2 == 0)
        table1.Columns[x].Background = Brushes.Beige;
    else
        table1.Columns[x].Background = Brushes.LightSteelBlue;
}

```

```

' Create 6 columns and add them to the table's Columns collection.
Dim numberofColumns = 6
Dim x
For x = 0 To numberofColumns
    table1.Columns.Add(new TableColumn())

    ' Set alternating background colors for the middle columns.
    If x Mod 2 = 0 Then
        table1.Columns(x).Background = Brushes.Beige
    Else
        table1.Columns(x).Background = Brushes.LightSteelBlue
    End If
Next x

```

Next, a title row is created and added to the table with some formatting applied. The title row happens to contain a single cell that spans all six columns in the table.

```

// Create and add an empty TableRowGroup to hold the table's Rows.
table1.RowGroups.Add(new TableRowGroup());

// Add the first (title) row.
table1.RowGroups[0].Rows.Add(new TableRow());

// Alias the current working row for easy reference.
TableRow currentRow = table1.RowGroups[0].Rows[0];

// Global formatting for the title row.
currentRow.Background = Brushes.Silver;
currentRow.FontSize = 40;
currentRow.FontWeight = System.Windows.FontWeights.Bold;

// Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("2004 Sales Project"))));
// and set the row to span all 6 columns.
currentRow.Cells[0].ColumnSpan = 6;

```

```

' Create and add an empty TableRowGroup to hold the table's Rows.
table1.RowGroups.Add(new TableRowGroup())

' Add the first (title) row.
table1.RowGroups(0).Rows.Add(new TableRow())

' Alias the current working row for easy reference.
Dim currentRow As New TableRow()
currentRow = table1.RowGroups(0).Rows(0)

' Global formatting for the title row.
currentRow.Background = Brushes.Silver
currentRow.FontSize = 40
currentRow.FontWeight = System.Windows.FontWeights.Bold

' Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("2004 Sales Project"))))
' and set the row to span all 6 columns.
currentRow.Cells(0).ColumnSpan = 6

```

Next, a header row is created and added to the table, and the cells in the header row are created and populated with content.

```

// Add the second (header) row.
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[1];

// Global formatting for the header row.
currentRow.FontSize = 18;
currentRow.FontWeight = FontWeights.Bold;

// Add cells with content to the second row.
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Product"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 1"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 2"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 3"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 4"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("TOTAL"))));

```

```

' Add the second (header) row.
table1.RowGroups(0).Rows.Add(new TableRow())
currentRow = table1.RowGroups(0).Rows(1)

' Global formatting for the header row.
currentRow.FontSize = 18
currentRow.FontWeight = FontWeights.Bold

' Add cells with content to the second row.
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Product"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 1"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 2"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 3"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 4"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("TOTAL"))))

```

Next, a row for data is created and added to the table, and the cells in this row are created and populated with content. Building this row is similar to building the header row, with slightly different formatting applied.

```

// Add the third row.
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[2];

// Global formatting for the row.
currentRow.FontSize = 12;
currentRow.FontWeight = FontWeights.Normal;

// Add cells with content to the third row.
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Widgets"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$50,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$55,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$60,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$65,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$230,000"))));

// Bold the first cell.
currentRow.Cells[0].FontWeight = FontWeights.Bold;

```

```

' Add the third row.
table1.RowGroups(0).Rows.Add(new TableRow())
currentRow = table1.RowGroups(0).Rows(2)

' Global formatting for the row.
currentRow.FontSize = 12
currentRow.FontWeight = FontWeights.Normal

' Add cells with content to the third row.
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Widgets"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$50,000"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$55,000"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$60,000"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$65,000"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$230,000"))))

' Bold the first cell.
currentRow.Cells(0).FontWeight = FontWeights.Bold

```

Finally, a footer row is created, added, and formatted. Like the title row, the footer contains a single cell that spans all six columns in the table.

```
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[3];

// Global formatting for the footer row.
currentRow.Background = Brushes.LightGray;
currentRow.FontSize = 18;
currentRow.FontWeight = System.Windows.FontWeights.Normal;

// Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Projected 2004 Revenue: $810,000"))));
// and set the row to span all 6 columns.
currentRow.Cells[0].ColumnSpan = 6;
```

```
table1.RowGroups(0).Rows.Add(new TableRow())
currentRow = table1.RowGroups(0).Rows(3)

' Global formatting for the footer row.
currentRow.Background = Brushes.LightGray
currentRow.FontSize = 18
currentRow.FontWeight = System.Windows.FontWeights.Normal

' Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Projected 2004 Revenue: $810,000"))))
' and set the row to span all 6 columns.
currentRow.Cells(0).ColumnSpan = 6
```

## See also

- [Flow Document Overview](#)
- [Define a Table with XAML](#)
- [Documents in WPF](#)
- [Use Flow Content Elements](#)

# Flow Content Elements How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to accomplish common tasks using various flow content elements and related features.

## In This Section

- [Adjust Spacing Between Paragraphs](#)
- [Build a Table Programmatically](#)
- [Change the FlowDirection of Content Programmatically](#)
- [Change the TextWrapping Property Programmatically](#)
- [Define a Table with XAML](#)
- [Alter the Typography of Text](#)
- [Enable Text Trimming](#)
- [Insert an Element Into Text Programmatically](#)
- [Manipulate Flow Content Elements through the Blocks Property](#)
- [Manipulate Flow Content Elements through the Inlines Property](#)
- [Manipulate a FlowDocument through the Blocks Property](#)
- [Manipulate a Table's Columns through the Columns Property](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)
- [Use Flow Content Elements](#)
- [Use FlowDocument Column-Separating Attributes](#)

## Reference

- [FlowDocument](#)
- [Block](#)
- [Inline](#)

## Related Sections

- [Documents in WPF](#)

# How to: Adjust Spacing Between Paragraphs

3/5/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to adjust or eliminate spacing between paragraphs in flow content.

In flow content, extra space that appears between paragraphs is the result of margins set on these paragraphs; thus, the spacing between paragraphs can be controlled by adjusting the margins on those paragraphs. To eliminate extra spacing between two paragraphs altogether, set the margins for the paragraphs to **0**. To achieve uniform spacing between paragraphs throughout an entire [FlowDocument](#), use styling to set a uniform margin value for all paragraphs in the [FlowDocument](#).

It is important to note that the margins for two adjacent paragraphs will "collapse" to the larger of the two margins, rather than doubling up. So, if two adjacent paragraphs have margins of 20 pixels and 40 pixels respectively, the resulting space between the paragraphs is 40 pixels, the larger of the two margin values.

## Example

The following example uses styling to set the margin for all [Paragraph](#) elements in a [FlowDocument](#) to **0**, which effectively eliminates extra spacing between paragraphs in the [FlowDocument](#).

```
<FlowDocument>
  <FlowDocument.Resources>
    <!-- This style is used to set the margins for all paragraphs in the FlowDocument to 0. -->
    <Style TargetType="{x:Type Paragraph}">
      <Setter Property="Margin" Value="0"/>
    </Style>
  </FlowDocument.Resources>

  <Paragraph>
    Spacing between paragraphs is caused by margins set on the paragraphs. Two adjacent margins
    will "collapse" to the larger of the two margin widths, rather than doubling up.
  </Paragraph>

  <Paragraph>
    To eliminate extra spacing between two paragraphs, just set the paragraph margins to 0.
  </Paragraph>
</FlowDocument>
```

# How to: Build a Table Programmatically

8/22/2019 • 4 minutes to read • [Edit Online](#)

The following examples show how to programmatically create a [Table](#) and populate it with content. The contents of the table are apportioned into five rows (represented by [TableRow](#) objects contained in a [RowGroups](#) object) and six columns (represented by [TableColumn](#) objects). The rows are used for different presentation purposes, including a title row intended to title the entire table, a header row to describe the columns of data in the table, and a footer row with summary information. Note that the notion of "title", "header", and "footer" rows are not inherent to the table; these are simply rows with different characteristics. Table cells contain the actual content, which can be comprised of text, images, or nearly any other user interface (UI) element.

## Example

First, a [FlowDocument](#) is created to host the [Table](#), and a new [Table](#) is created and added to the contents of the [FlowDocument](#).

```
// Create the parent FlowDocument...
flowDoc = new FlowDocument();

// Create the Table...
table1 = new Table();
// ...and add it to the FlowDocument Blocks collection.
flowDoc.Blocks.Add(table1);

// Set some global formatting properties for the table.
table1.CellSpacing = 10;
table1.Background = Brushes.White;
```

```
' Create the parent FlowDocument...
flowDoc = New FlowDocument()

' Create the Table...
table1 = New Table()

' ...and add it to the FlowDocument Blocks collection.
flowDoc.Blocks.Add(table1)

' Set some global formatting properties for the table.
table1.CellSpacing = 10
table1.Background = Brushes.White
```

## Example

Next, six [TableColumn](#) objects are created and added to the table's [Columns](#) collection, with some formatting applied.

### NOTE

Note that the table's [Columns](#) collection uses standard zero-based indexing.

```

// Create 6 columns and add them to the table's Columns collection.
int numberOfColumns = 6;
for (int x = 0; x < numberOfColumns; x++)
{
    table1.Columns.Add(new TableColumn());

    // Set alternating background colors for the middle columns.
    if(x%2 == 0)
        table1.Columns[x].Background = Brushes.Beige;
    else
        table1.Columns[x].Background = Brushes.LightSteelBlue;
}

```

```

' Create 6 columns and add them to the table's Columns collection.
Dim numberOfColumns = 6
Dim x
For x = 0 To numberOfColumns
    table1.Columns.Add(new TableColumn())

    ' Set alternating background colors for the middle columns.
    If x Mod 2 = 0 Then
        table1.Columns(x).Background = Brushes.Beige
    Else
        table1.Columns(x).Background = Brushes.LightSteelBlue
    End If
Next x

```

## Example

Next, a title row is created and added to the table with some formatting applied. The title row happens to contain a single cell that spans all six columns in the table.

```

// Create and add an empty TableRowGroup to hold the table's Rows.
table1.RowGroups.Add(new TableRowGroup());

// Add the first (title) row.
table1.RowGroups[0].Rows.Add(new TableRow());

// Alias the current working row for easy reference.
TableRow currentRow = table1.RowGroups[0].Rows[0];

// Global formatting for the title row.
currentRow.Background = Brushes.Silver;
currentRow.FontSize = 40;
currentRow.FontWeight = System.Windows.FontWeights.Bold;

// Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("2004 Sales Project"))));
// and set the row to span all 6 columns.
currentRow.Cells[0].ColumnSpan = 6;

```

```

' Create and add an empty TableRowGroup to hold the table's Rows.
table1.RowGroups.Add(new TableRowGroup())

' Add the first (title) row.
table1.RowGroups(0).Rows.Add(new TableRow())

' Alias the current working row for easy reference.
Dim currentRow As New TableRow()
currentRow = table1.RowGroups(0).Rows(0)

' Global formatting for the title row.
currentRow.Background = Brushes.Silver
currentRow.FontSize = 40
currentRow.FontWeight = System.Windows.FontWeights.Bold

' Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("2004 Sales Project"))))

' and set the row to span all 6 columns.
currentRow.Cells(0).ColumnSpan = 6

```

## Example

Next, a header row is created and added to the table, and the cells in the header row are created and populated with content.

```

// Add the second (header) row.
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[1];

// Global formatting for the header row.
currentRow.FontSize = 18;
currentRow.FontWeight = FontWeights.Bold;

// Add cells with content to the second row.
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Product"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 1"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 2"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 3"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 4"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("TOTAL"))));

```

```

' Add the second (header) row.
table1.RowGroups(0).Rows.Add(new TableRow())
currentRow = table1.RowGroups(0).Rows(1)

' Global formatting for the header row.
currentRow.FontSize = 18
currentRow.FontWeight = FontWeights.Bold

' Add cells with content to the second row.
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Product"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 1"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 2"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 3"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 4"))))
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("TOTAL"))))

```

## Example

Next, a row for data is created and added to the table, and the cells in this row are created and populated with

content. Building this row is similar to building the header row, with slightly different formatting applied.

```
// Add the third row.  
table1.RowGroups[0].Rows.Add(new TableRow());  
currentRow = table1.RowGroups[0].Rows[2];  
  
// Global formatting for the row.  
currentRow.FontSize = 12;  
currentRow.FontWeight = FontWeights.Normal;  
  
// Add cells with content to the third row.  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Widgets"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$50,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$55,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$60,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$65,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$230,000"))));  
  
// Bold the first cell.  
currentRow.Cells[0].FontWeight = FontWeights.Bold;
```

```
' Add the third row.  
table1.RowGroups(0).Rows.Add(new TableRow())  
currentRow = table1.RowGroups(0).Rows(2)  
  
' Global formatting for the row.  
currentRow.FontSize = 12  
currentRow.FontWeight = FontWeights.Normal  
  
' Add cells with content to the third row.  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Widgets"))))  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$50,000"))))  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$55,000"))))  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$60,000"))))  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$65,000"))))  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$230,000"))))  
  
' Bold the first cell.  
currentRow.Cells(0).FontWeight = FontWeights.Bold
```

## Example

Finally, a footer row is created, added, and formatted. Like the title row, the footer contains a single cell that spans all six columns in the table.

```
table1.RowGroups[0].Rows.Add(new TableRow());  
currentRow = table1.RowGroups[0].Rows[3];  
  
// Global formatting for the footer row.  
currentRow.Background = Brushes.LightGray;  
currentRow.FontSize = 18;  
currentRow.FontWeight = System.Windows.FontWeights.Normal;  
  
// Add the header row with content,  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Projected 2004 Revenue: $810,000"))));  
// and set the row to span all 6 columns.  
currentRow.Cells[0].ColumnSpan = 6;
```

```
table1.RowGroups(0).Rows.Add(new TableRow())
currentRow = table1.RowGroups(0).Rows(3)

' Global formatting for the footer row.
currentRow.Background = Brushes.LightGray
currentRow.FontSize = 18
currentRow.FontWeight = System.Windows.FontWeights.Normal

' Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Projected 2004 Revenue: $810,000"))))
' and set the row to span all 6 columns.
currentRow.Cells(0).ColumnSpan = 6
```

## See also

- [Table Overview](#)

# How to: Change the FlowDirection of Content Programmatically

3/5/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to programmatically change the `FlowDirection` property of a `FlowDocumentReader`.

## Example

Two `Button` elements are created, each representing one of the possible values of `FlowDirection`. When a button is clicked, the associated property value is applied to the contents of a `FlowDocumentReader` named `tf1`. The property value is also written to a `TextBlock` named `txt1`.

```
<StackPanel DockPanel.Dock="Top" Orientation="Horizontal" Margin="0,0,0,10">
    <Button Click="LR">LeftToRight</Button>
    <Button Click="RL">RightToLeft</Button>
</StackPanel>

<TextBlock Name="txt1" DockPanel.Dock="Bottom" Margin="0,50,0,0"/>

<FlowDocumentReader>
    <FlowDocument FontFamily="Arial" Name="tf1">
        <Paragraph>
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit,
            sed diam nonummy nibh euismod tincidunt ut laoreet dolore
            magna aliquam erat volutpat. Ut wisi enim ad minim veniam,
            quis nostrud exerci tation ullamcorper suscipit lobortis nisl
            ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
            nostrud exerci tation ullamcorper suscipit lobortis nisl ut
            aliquip ex ea commodo consequat. Duis autem vel eum iriure.
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
            nostrud exerci tation ullamcorper suscipit lobortis nisl ut
            aliquip ex ea commodo consequat. Duis autem vel eum iriure.
        </Paragraph>
        <Paragraph>
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
            nostrud exerci tation ullamcorper suscipit lobortis nisl ut
            aliquip ex ea commodo consequat. Duis autem vel eum iriure.
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
            nostrud exerci tation ullamcorper suscipit lobortis nisl ut
            aliquip ex ea commodo consequat. Duis autem vel eum iriure.
        </Paragraph>
    </FlowDocument>
</FlowDocumentReader>
```

## Example

The events associated with the button clicks defined above are handled in a C# code-behind file.

```
private void LR(object sender, RoutedEventArgs e)
{
    tf1.FlowDirection = FlowDirection.LeftToRight;
    txt1.Text = "FlowDirection is now " + tf1.FlowDirection;
}
private void RL(object sender, RoutedEventArgs e)
{
    tf1.FlowDirection = FlowDirection.RightToLeft;
    txt1.Text = "FlowDirection is now " + tf1.FlowDirection;
}
```

```
Private Sub LR(ByVal sender As Object, ByVal e As RoutedEventArgs)
    tf1.FlowDirection = FlowDirection.LeftToRight
    txt1.Text = "FlowDirection is now " & tf1.FlowDirection
End Sub
Private Sub RL(ByVal sender As Object, ByVal e As RoutedEventArgs)
    tf1.FlowDirection = FlowDirection.RightToLeft
    txt1.Text = "FlowDirection is now " & tf1.FlowDirection
End Sub
```

# How to: Change the TextWrapping Property Programmatically

4/8/2019 • 2 minutes to read • [Edit Online](#)

## Example

The following code example shows how to change the value of the `TextWrapping` property programmatically.

Three `Button` elements are placed within a `StackPanel` element in XAML. Each `Click` event for a `Button` corresponds with an event handler in the code. The event handlers use the same name as the `TextWrapping` value they will apply to `txt2` when the button is clicked. Also, the text in `txt1` (a `TextBlock` not shown in the XAML) is updated to reflect the change in the property.

```
<StackPanel Orientation="Horizontal" Margin="0,0,0,20">
    <Button Name="btn1" Background="Silver" Width="100" Click="Wrap">Wrap</Button>
    <Button Name="btn2" Background="Silver" Width="100" Click="NoWrap">NoWrap</Button>
    <Button Name="btn4" Background="Silver" Width="100" Click="WrapWithOverflow">WrapWithOverflow</Button>
</StackPanel>

<TextBlock Name="txt2" TextWrapping="Wrap" Margin="0,0,0,20" Foreground="Black">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Lorem ipsum dolor sit amet,
    consectetur adipiscing elit. Lorem ipsum dolor sit aet, consectetur adipiscing elit.
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
</TextBlock>
```

```
private void Wrap(object sender, RoutedEventArgs e)
{
    txt2.TextWrapping = System.Windows.TextWrapping.Wrap;
    txt1.Text = "The TextWrap property is currently set to Wrap.";
}

private void NoWrap(object sender, RoutedEventArgs e)
{
    txt2.TextWrapping = System.Windows.TextWrapping.NoWrap;
    txt1.Text = "The TextWrap property is currently set to NoWrap.";
}

private void WrapWithOverflow(object sender, RoutedEventArgs e)
{
    txt2.TextWrapping = System.Windows.TextWrapping.WrapWithOverflow;
    txt1.Text = "The TextWrap property is currently set to WrapWithOverflow.";
}
```

```
Private Sub Wrap(ByVal sender As Object, ByVal e As System.Windows.RoutedEventArgs)
    txt2.TextWrapping = System.Windows.TextWrapping.Wrap
    txt1.Text = "The TextWrap property is currently set to Wrap."
End Sub

Private Sub NoWrap(ByVal sender As Object, ByVal e As System.Windows.RoutedEventArgs)
    txt2.TextWrapping = System.Windows.TextWrapping.NoWrap
    txt1.Text = "The TextWrap property is currently set to NoWrap."
End Sub

Private Sub WrapWithOverflow(ByVal sender As Object, ByVal e As System.Windows.RoutedEventArgs)
    txt2.TextWrapping = System.Windows.TextWrapping.WrapWithOverflow
    txt1.Text = "The TextWrap property is currently set to WrapWithOverflow."
End Sub
```

## See also

- [TextWrapping](#)
- [TextWrapping](#)

# How to: Define a Table with XAML

3/29/2019 • 2 minutes to read • [Edit Online](#)

The following example demonstrates how to define a **Table** using Extensible Application Markup Language (XAML). The example table has four columns (represented by **TableColumn** elements) and several rows (represented by **TableRow** elements) containing data as well as title, header, and footer information. Rows must be contained in a **TableRowGroup** element. Each row in the table is comprised of one or more cells (represented by **TableCell** elements). Content in a table cell must be contained in a **Block** element; in this case **Paragraph** elements are used. The table also hosts a hyperlink (represented by the **Hyperlink** element) in the footer row.

## Example

```
<FlowDocumentReader>
<FlowDocument>

    <Table CellSpacing="5">

        <Table.Columns>
            < TableColumn />
            < TableColumn />
            < TableColumn />
            < TableColumn />
        </Table.Columns>

        < TableRowGroup >

            <!-- Title row for the table. -->
            < TableRow Background="SkyBlue">
                < TableCell ColumnSpan="4" TextAlignment="Center">
                    < Paragraph FontSize="24pt" FontWeight="Bold" >Planetary Information</ Paragraph >
                </ TableCell >
            </ TableRow >

            <!-- Header row for the table. -->
            < TableRow Background="LightGoldenrodYellow">
                < TableCell >< Paragraph FontSize="14pt" FontWeight="Bold" >Planet</ Paragraph ></ TableCell >
                < TableCell >< Paragraph FontSize="14pt" FontWeight="Bold" >Mean Distance from Sun</ Paragraph >
                < TableCell >< Paragraph FontSize="14pt" FontWeight="Bold" >Mean Diameter</ Paragraph ></ TableCell >
                < TableCell >< Paragraph FontSize="14pt" FontWeight="Bold" >Approximate Mass</ Paragraph ></ TableCell >
            </ TableRow >

            <!-- Sub-title row for the inner planets. -->
            < TableRow >
                < TableCell ColumnSpan="4" >< Paragraph FontSize="14pt" FontWeight="Bold" >The Inner Planets</ Paragraph >
            </ TableCell >
            </ TableRow >

            <!-- Four data rows for the inner planets. -->
            < TableRow >
                < TableCell >< Paragraph >Mercury</ Paragraph ></ TableCell >
                < TableCell >< Paragraph >57,910,000 km</ Paragraph ></ TableCell >
                < TableCell >< Paragraph >4,880 km</ Paragraph ></ TableCell >
                < TableCell >< Paragraph >3.30e23 kg</ Paragraph ></ TableCell >
            </ TableRow >
            < TableRow Background="lightgray">
                < TableCell >< Paragraph >Venus</ Paragraph ></ TableCell >
                < TableCell >< Paragraph >108,200,000 km</ Paragraph ></ TableCell >
                < TableCell >< Paragraph >12,103.6 km</ Paragraph ></ TableCell >
                < TableCell >< Paragraph >4.869e24 kg</ Paragraph ></ TableCell >
            </ TableRow >
        </ TableRowGroup >
    </ Table >
</ FlowDocument >
</ FlowDocumentReader >
```

```

</TableColumn>
</TableColumn>
<TableColumn><Paragraph>Earth</Paragraph></TableColumn>
<TableColumn><Paragraph>149,600,000 km</Paragraph></TableColumn>
<TableColumn><Paragraph>12,756.3 km</Paragraph></TableColumn>
<TableColumn><Paragraph>5.972e24 kg</Paragraph></TableColumn>
</TableRow>
<TableRow Background="lightgray">
<TableCell><Paragraph>Mars</Paragraph></TableCell>
<TableCell><Paragraph>227,940,000 km</Paragraph></TableCell>
<TableCell><Paragraph>6,794 km</Paragraph></TableCell>
<TableCell><Paragraph>6.4219e23 kg</Paragraph></TableCell>
</TableRow>

<!-- Sub-title row for the outer planets. --&gt;
&lt;TableRow&gt;
&lt;TableCell ColumnSpan="4"&gt;&lt;Paragraph FontSize="14pt" FontWeight="Bold"&gt;The Major Outer Planets&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;/TableRow&gt;

<!-- Four data rows for the major outer planets. --&gt;
&lt;TableRow&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;Jupiter&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;778,330,000 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;142,984 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;1.900e27 kg&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;/TableRow&gt;
&lt;TableRow Background="lightgray"&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;Saturn&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;1,429,400,000 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;120,536 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;5.68e26 kg&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;/TableRow&gt;
&lt;TableRow&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;Uranus&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;2,870,990,000 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;51,118 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;8.683e25 kg&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;/TableRow&gt;
&lt;TableRow Background="lightgray"&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;Neptune&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;4,504,000,000 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;49,532 km&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;TableCell&gt;&lt;Paragraph&gt;1.0247e26 kg&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;/TableRow&gt;

<!-- Footer row for the table. --&gt;
&lt;TableRow&gt;
&lt;TableCell ColumnSpan="4"&gt;&lt;Paragraph FontSize="10pt" FontStyle="Italic"&gt;
Information from the
&lt;Hyperlink NavigateUri="http://encarta.msn.com/encnet/refpages/artcenter.aspx"&gt;Encarta&lt;/Hyperlink&gt;
web site.
&lt;/Paragraph&gt;&lt;/TableCell&gt;
&lt;/TableRow&gt;

&lt;/TableRowGroup&gt;
&lt;/Table&gt;
&lt;/FlowDocument&gt;
&lt;/FlowDocumentReader&gt;
</pre>

```

The following figure shows how the table defined in this example renders:

Planetary Information			
Planet	Mean Distance from Sun	Mean Diameter	Approximate Mass
<b>The Inner Planets</b>			
Mercury	57,910,000 km	4,880 km	3.30e23 kg
Venus	108,200,000 km	12,103.6 km	4.869e24 kg
Earth	149,600,000 km	12,756.3 km	5.972e24 kg
Mars	227,940,000 km	6,794 km	6.4219e23 kg
<b>The Major Outer Planets</b>			
Jupiter	778,330,000 km	142,984 km	1.900e27 kg
Saturn	1,429,400,000 km	120,536 km	5.68e26 kg
Uranus	2,870,990,000 km	51,118 km	8.683e25 kg

# How-to: Alter the Typography of Text

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to set the **Typography** attribute, using **Paragraph** as the example element.

## Example

```
<Paragraph  
    TextAlignment="Left"  
    FontSize="18"  
    FontFamily="Palatino Linotype"  
    Typography.NumeralStyle="OldStyle"  
    Typography.Fraction="Stacked"  
    Typography.Variants="Inferior"  
>  
<Run>  
    This text has some altered typography characteristics. Note  
    that use of an open type font is necessary for most typographic  
    properties to be effective.  
</Run>  
<LineBreak/><LineBreak/>  
<Run>  
    0123456789 10 11 12 13  
</Run>  
<LineBreak/><LineBreak/>  
<Run>  
    1/2 2/3 3/4  
</Run>  
</Paragraph>
```

The following figure shows how this example renders.

This text has some altered typography characteristics. Note that use of an open type font is necessary for most typographic properties to be effective.

0123456789 10 11 12 13

$\frac{1}{2}$   $\frac{2}{3}$   $\frac{3}{4}$

In contrast, the following figure shows how a similar example with default typographic properties renders.

This text has some altered typography characteristics. Note that use of an open type font is necessary for most typographic properties to be effective.

0123456789 10 11 12 13

1/2 2/3 3/4

## Example

The following example shows how to set the **Typography** property programmatically.

```

Paragraph par = new Paragraph();

Run runText = new Run(
    "This text has some altered typography characteristics. Note" +
    "that use of an open type font is necessary for most typographic" +
    "properties to be effective.");
Run runNumerals = new Run("0123456789 10 11 12 13");
Run runFractions = new Run("1/2 2/3 3/4");

par.Inlines.Add(runText);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runNumerals);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runFractions);

par.TextAlignment = TextAlignment.Left;
par.FontSize = 18;
par.FontFamily = new FontFamily("Palatino Linotype");

par.Typography.NumeralStyle = FontNumeralStyle.OldStyle;
par.Typography.Fraction = FontFraction.Stacked;
par.Typography.Variants = FontVariants.Inferior;

```

```

Dim par As New Paragraph()

Dim runText As New Run("This text has some altered typography characteristics. Note" & "that use of an open
type font is necessary for most typographic" & "properties to be effective.")
Dim runNumerals As New Run("0123456789 10 11 12 13")
Dim runFractions As New Run("1/2 2/3 3/4")

par.Inlines.Add(runText)
par.Inlines.Add(New LineBreak())
par.Inlines.Add(New LineBreak())
par.Inlines.Add(runNumerals)
par.Inlines.Add(New LineBreak())
par.Inlines.Add(New LineBreak())
par.Inlines.Add(runFractions)

par.TextAlignment = TextAlignment.Left
par.FontSize = 18
par.FontFamily = New FontFamily("Palatino Linotype")

par.Typography.NumeralStyle = FontNumeralStyle.OldStyle
par.Typography.Fraction = FontFraction.Stacked
par.Typography.Variants = FontVariants.Inferior

```

## See also

- [Flow Document Overview](#)

# How to: Enable Text Trimming

3/6/2019 • 2 minutes to read • [Edit Online](#)

This example demonstrates the usage and effects of the values available in the [TextTrimming](#) enumeration.

## Example

The following example defines a [TextBlock](#) element with the [TextTrimming](#) attribute set.

```
<TextBlock  
    Name="myTextBlock"  
    Margin="20" Background="LightGoldenrodYellow"  
    TextTrimming="WordEllipsis" TextWrapping="NoWrap"  
    FontSize="14"  
>  
    One<LineBreak/>  
    two two<LineBreak/>  
    Three Three<LineBreak/>  
    four four four<LineBreak/>  
    Five Five Five Five<LineBreak/>  
    six six six six six<LineBreak/>  
    Seven Seven Seven Seven Seven Seven  
</TextBlock>
```

Setting the corresponding [TextTrimming](#) property in code is demonstrated below.

```
myTextBlock.TextTrimming = TextTrimming.CharacterEllipsis;
```

```
myTextBlock.TextTrimming = TextTrimming.CharacterEllipsis
```

There are currently three options for trimming text: **CharacterEllipsis**, **WordEllipsis**, and **None**.

When [TextTrimming](#) is set to **CharacterEllipsis**, text is trimmed and continued with an ellipsis at the character closest to the trimming edge. This setting tends to trim text to fit more closely to the trimming boundary, but may result in words being partially trimmed. The following figure shows the effect of this setting on a [TextBlock](#) similar to the one defined above.

```
One  
two two  
Three Thre...  
four four fo...  
Five Five Fi...  
six six six si...  
Seven Seve...
```

When [TextTrimming](#) is set to **WordEllipsis**, text is trimmed and continued with an ellipsis at the end of the first full word closest to the trimming edge. This setting will not show partially trimmed words, but tends not to trim text as closely to the trimming edge as the **CharacterEllipsis** setting. The following figure shows the effect of this setting on the [TextBlock](#) defined above.

```
One  
two two  
Three...  
four four...  
Five Five...  
six six six...  
Seven...
```

When [TextTrimming](#) is set to **None**, no text trimming is performed. In this case, text is simply cropped to the boundary of the parent text container. The following figure shows the effect of this setting on a [TextBlock](#) similar to the one defined above.

```
One  
two two  
Three Three T  
four four four  
Five Five Five  
six six six six s  
Seven Seven S
```

# How to: Insert an Element Into Text Programmatically

4/8/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to use two [TextPointer](#) objects to specify a range within text to apply a [Span](#) element to.

## Example

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class InsertInlineIntoTextExample : Page
    {
        public InsertInlineIntoTextExample()
        {

            // Create a paragraph with a short sentence
            Paragraph myParagraph = new Paragraph(new Run("Neptune has 72 times Earth's volume..."));

            // Create two TextPointers that will specify the text range the Span will cover
            TextPointer myTextPointer1 = myParagraph.ContentStart.GetPositionAtOffset(10);
            TextPointer myTextPointer2 = myParagraph.ContentEnd.GetPositionAtOffset(-5);

            // Create a Span that covers the range between the two TextPointers.
            Span mySpan = new Span(myTextPointer1, myTextPointer2);
            mySpan.Background = Brushes.Red;

            // Create a FlowDocument with the paragraph as its initial content.
            FlowDocument myFlowDocument = new FlowDocument(myParagraph);

            this.Content = myFlowDocument;

        }
    }
}
```

```
Imports System.Windows
Imports System.Windows.Media
Imports System.Windows.Controls
Imports System.Windows.Documents

Namespace SDKSample
    Partial Public Class InsertInlineIntoTextExample
        Inherits Page
        Public Sub New()

            ' Create a paragraph with a short sentence
            Dim myParagraph As New Paragraph(New Run("Neptune has 72 times Earth's volume..."))

            ' Create two TextPointers that will specify the text range the Span will cover
            Dim myTextPointer1 As TextPointer = myParagraph.ContentStart.GetPositionAtOffset(10)
            Dim myTextPointer2 As TextPointer = myParagraph.ContentEnd.GetPositionAtOffset(-5)

            ' Create a Span that covers the range between the two TextPointers.
            Dim mySpan As New Span(myTextPointer1, myTextPointer2)
            mySpan.Background = Brushes.Red

            ' Create a FlowDocument with the paragraph as its initial content.
            Dim myFlowDocument As New FlowDocument(myParagraph)

            Me.Content = myFlowDocument

        End Sub
    End Class
End Namespace
```

The illustration below shows what this example looks like.

**Neptune has 72 times Earth's volume...**

## See also

- [Flow Document Overview](#)

# How to: Manipulate Flow Content Elements through the Blocks Property

4/28/2019 • 2 minutes to read • [Edit Online](#)

These examples demonstrate some of the more common operations that can be performed on flow content elements through the **Blocks** property. This property is used to add and remove items from [BlockCollection](#). Flow content elements that feature a **Blocks** property include:

- [Figure](#)
- [Floater](#)
- [ListItem](#)
- [Section](#)
- [TableCell](#)

These examples happen to use [Section](#) as the flow content element, but these techniques are applicable to all elements that host a flow content element collection.

## Example

The following example creates a new [Section](#) and then uses the **Add** method to add a new Paragraph to the [Section](#) contents.

```
Section secx = new Section();
secx.Blocks.Add(new Paragraph(new Run("A bit of text content...")));
```

```
Dim secx As New Section()
secx.Blocks.Add(New Paragraph(New Run("A bit of text content...")))
```

## Example

The following example creates a new [Paragraph](#) element and inserts it at the beginning of the [Section](#).

```
Paragraph parx = new Paragraph(new Run("Text to insert..."));
secx.Blocks.InsertBefore(secx.Blocks.FirstBlock, parx);
```

```
Dim parx As New Paragraph(New Run("Text to insert..."))
secx.Blocks.InsertBefore(secx.Blocks.FirstBlock, parx)
```

## Example

The following example gets the number of top-level [Block](#) elements contained in the [Section](#).

```
int countTopLevelBlocks = secx.Blocks.Count;
```

```
Dim countTopLevelBlocks As Integer = secx.Blocks.Count
```

## Example

The following example deletes the last [Block](#) element in the [Section](#).

```
secx.Blocks.Remove(secx.Blocks.LastBlock);
```

```
secx.Blocks.Remove(secx.Blocks.LastBlock)
```

## Example

The following example clears all of the contents ([Block](#) elements) from the [Section](#).

```
secx.Blocks.Clear();
```

```
secx.Blocks.Clear()
```

## See also

- [BlockCollection](#)
- [InlineCollection](#)
- [ListItemCollection](#)
- [Flow Document Overview](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)
- [Manipulate a Table's Columns through the Columns Property](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)

# How to: Manipulate Flow Content Elements through the Inlines Property

4/28/2019 • 2 minutes to read • [Edit Online](#)

These examples demonstrate some of the more common operations that can be performed on inline flow content elements (and containers of such elements, such as [TextBlock](#)) through the **Inlines** property. This property is used to add and remove items from [InlineCollection](#). Flow content elements that feature an **Inlines** property include:

- [Bold](#)
- [Hyperlink](#)
- [Italic](#)
- [Paragraph](#)
- [Span](#)
- [Underline](#)

These examples happen to use [Span](#) as the flow content element, but these techniques are applicable to all elements or controls that host an [InlineCollection](#) collection.

## Example

The following example creates a new [Span](#) object, and then uses the **Add** method to add two text runs as content children of the [Span](#).

```
Span spanx = new Span();
spanx.Inlines.Add(new Run("A bit of text content..."));
spanx.Inlines.Add(new Run("A bit more text content..."));
```

```
Dim spanx As New Span()
spanx.Inlines.Add(New Run("A bit of text content..."))
spanx.Inlines.Add(New Run("A bit more text content..."))
```

## Example

The following example creates a new [Run](#) element and inserts it at the beginning of the [Span](#).

```
Run runx = new Run("Text to insert...");
spanx.Inlines.InsertBefore(spanx.Inlines.FirstInline, runx);
```

```
Dim runx As New Run("Text to insert...")
spanx.Inlines.InsertBefore(spanx.Inlines.FirstInline, runx)
```

## Example

The following example gets the number of top-level [Inline](#) elements contained in the [Span](#).

```
int countTopLevelInlines = spanx.Inlines.Count;
```

```
Dim countTopLevelInlines As Integer = spanx.Inlines.Count
```

## Example

The following example deletes the last [Inline](#) element in the [Span](#).

```
spanx.Inlines.Remove(spanx.Inlines.LastInline);
```

```
spanx.Inlines.Remove(spanx.Inlines.LastInline)
```

## Example

The following example clears all of the contents ([Inline](#) elements) from the [Span](#).

```
spanx.Inlines.Clear();
```

```
spanx.Inlines.Clear()
```

## See also

- [BlockCollection](#)
- [InlineCollection](#)
- [ListItemCollection](#)
- [Flow Document Overview](#)
- [Manipulate a FlowDocument through the Blocks Property](#)
- [Manipulate a Table's Columns through the Columns Property](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)

# How to: Manipulate a FlowDocument through the Blocks Property

4/8/2019 • 2 minutes to read • [Edit Online](#)

These examples demonstrate some of the more common operations that can be performed on a [FlowDocument](#) through the [Blocks](#) property.

## Example

The following example creates a new [FlowDocument](#) and then appends a new [Paragraph](#) element to the [FlowDocument](#).

```
FlowDocument flowDoc = new FlowDocument(new Paragraph(new Run("A bit of text content...")));
flowDoc.Blocks.Add(new Paragraph(new Run("Text to append...")));
```

```
Dim flowDoc As New FlowDocument(New Paragraph(New Run("A bit of text content...")))
flowDoc.Blocks.Add(New Paragraph(New Run("Text to append...")))
```

## Example

The following example creates a new [Paragraph](#) element and inserts it at the beginning of the [FlowDocument](#).

```
Paragraph p = new Paragraph(new Run("Text to insert..."));
flowDoc.Blocks.InsertBefore(flowDoc.Blocks.FirstBlock, p);
```

```
Dim p As New Paragraph(New Run("Text to insert..."))
flowDoc.Blocks.InsertBefore(flowDoc.Blocks.FirstBlock, p)
```

## Example

The following example gets the number of top-level [Block](#) elements contained in the [FlowDocument](#).

```
int countTopLevelBlocks = flowDoc.Blocks.Count;
```

```
Dim countTopLevelBlocks As Integer = flowDoc.Blocks.Count
```

## Example

The following example deletes the last [Block](#) element in the [FlowDocument](#).

```
flowDoc.Blocks.Remove(flowDoc.Blocks.LastBlock);
```

```
flowDoc.Blocks.Remove(flowDoc.Blocks.LastBlock)
```

## Example

The following example clears all of the contents ([Block](#) elements) from the [FlowDocument](#).

```
flowDoc.Blocks.Clear();
```

```
flowDoc.Blocks.Clear()
```

## See also

- [Manipulate a Table's Row Groups through the RowGroups Property](#)
- [Manipulate a Table's Columns through the Columns Property](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)

# How to: Manipulate a Table's Columns through the Columns Property

8/22/2019 • 2 minutes to read • [Edit Online](#)

This example demonstrates some of the more common operations that can be performed on a table's columns through the [Columns](#) property.

## Example

The following example creates a new table and then uses the [Add](#) method to add columns to the table's [Columns](#) collection.

```
Table tbl = new Table();
int columnsToAdd = 4;
for (int x = 0; x < columnsToAdd; x++)
    tbl.Columns.Add(new TableColumn());
```

```
Dim tbl As New Table()
Dim columnsToAdd As Integer = 4
For x As Integer = 0 To columnsToAdd - 1
    tbl.Columns.Add(New TableColumn())
Next x
```

## Example

The following example inserts a new [TableColumn](#). The new column is inserted at index position 0, making it the new first column in the table.

### NOTE

The [TableColumnCollection](#) collection uses standard zero-based indexing.

```
tbl.Columns.Insert(0, new TableColumn());
```

```
tbl.Columns.Insert(0, New TableColumn())
```

## Example

The following example accesses some arbitrary properties on columns in the [TableColumnCollection](#) collection, referring to particular columns by index.

```
tbl.Columns[0].Width = new GridLength(20);
tbl.Columns[1].Background = Brushes.AliceBlue;
tbl.Columns[2].Width = new GridLength(20);
tbl.Columns[3].Background = Brushes.AliceBlue;
```

```
tbl.Columns(0).Width = New GridLength(20)
tbl.Columns(1).Background = Brushes.AliceBlue
tbl.Columns(2).Width = New GridLength(20)
tbl.Columns(3).Background = Brushes.AliceBlue
```

## Example

The following example gets the number of columns currently hosted by the table.

```
int columns = tbl.Columns.Count;
```

```
Dim columns As Integer = tbl.Columns.Count
```

## Example

The following example removes a particular column by reference.

```
tbl.Columns.Remove(tbl.Columns[3]);
```

```
tbl.Columns.Remove(tbl.Columns(3))
```

## Example

The following example removes a particular column by index.

```
tbl.Columns.RemoveAt(2);
```

```
tbl.Columns.RemoveAt(2)
```

## Example

The following example removes all columns from the table's columns collection.

```
tbl.Columns.Clear();
```

```
tbl.Columns.Clear()
```

## See also

- [Table Overview](#)
- [Define a Table with XAML](#)
- [Build a Table Programmatically](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)
- [Manipulate a FlowDocument through the Blocks Property](#)
- [Manipulate a Table's Row Groups through the RowGroups Property](#)



# How to: Manipulate a Table's Row Groups through the RowGroups Property

8/22/2019 • 2 minutes to read • [Edit Online](#)

This example demonstrates some of the more common operations that can be performed on a table's row groups through the [RowGroups](#) property.

## Example

The following example creates a new table and then uses the [Add](#) method to add columns to the table's [RowGroups](#) collection.

```
Table tbl = new Table();
int rowGroupsToAdd = 4;
for (int x = 0; x < rowGroupsToAdd; x++)
    tbl.RowGroups.Add(new TableRowGroup());
```

```
Dim tbl As New Table()
Dim rowGroupsToAdd As Integer = 4
For x As Integer = 0 To rowGroupsToAdd - 1
    tbl.RowGroups.Add(New TableRowGroup())
Next x
```

## Example

The following example inserts a new [TableRowGroup](#). The new column is inserted at index position 0, making it the new first row group in the table.

### NOTE

The [TableRowGroupCollection](#) collection uses standard zero-based indexing.

```
tbl.RowGroups.Insert(0, new TableRowGroup());
```

```
tbl.RowGroups.Insert(0, New TableRowGroup())
```

## Example

The following example adds several rows to a particular [TableRowGroup](#) (specified by index) in the table.

```
int rowsToAdd = 10;
for (int x = 0; x < rowsToAdd; x++)
    tbl.RowGroups[0].Rows.Add(new TableRow());
```

```

Dim rowsToAdd As Integer = 10
For x As Integer = 0 To rowsToAdd - 1
    tbl.RowGroups(0).Rows.Add(New TableRow())
Next x

```

## Example

The following example accesses some arbitrary properties on rows in the first row group in the table.

```

// Alias the working TableRowGroup for ease in referencing.
TableRowGroup trg = tbl.RowGroups[0];
trg.Rows[0].Background = Brushes.CornflowerBlue;
trg.Rows[1].FontSize = 24;
trg.Rows[2].ToolTip = "This row's tooltip";

```

```

' Alias the working TableRowGroup for ease in referencing.
Dim trg As TableRowGroup = tbl.RowGroups(0)
trg.Rows(0).Background = Brushes.CornflowerBlue
trg.Rows(1).FontSize = 24
trg.Rows(2).ToolTip = "This row's tooltip"

```

## Example

The following example adds several cells to a particular [TableRow](#) (specified by index) in the table.

```

int cellsToAdd = 10;
for (int x = 0; x < cellsToAdd; x++)
    tbl.RowGroups[0].Rows[0].Cells.Add(new TableCell(new Paragraph(new Run("Cell " + (x + 1)))));

```

```

Dim cellsToAdd As Integer = 10
For x As Integer = 0 To cellsToAdd - 1
    tbl.RowGroups(0).Rows(0).Cells.Add(New TableCell(New Paragraph(New Run("Cell " & (x + 1)))))
Next x

```

## Example

The following example access some arbitrary methods and properties on cells in the first row in the first row group.

```

// Alias the working for for ease in referencing.
TableRow row = tbl.RowGroups[0].Rows[0];
row.Cells[0].Background = Brushes.PapayaWhip;
row.Cells[1].FontStyle = FontStyles.Italic;
// This call clears all of the content from this cell.
row.Cells[2].Blocks.Clear();

```

```

' Alias the working for for ease in referencing.
Dim row As TableRow = tbl.RowGroups(0).Rows(0)
row.Cells(0).Background = Brushes.PapayaWhip
row.Cells(1).FontStyle = FontStyles.Italic
' This call clears all of the content from this cell.
row.Cells(2).Blocks.Clear()

```

## Example

The following example returns the number of [TableRowGroup](#) elements hosted by the table.

```
int rowGroups = tbl.RowGroups.Count;
```

```
Dim rowGroups As Integer = tbl.RowGroups.Count
```

## Example

The following example removes a particular row group by reference.

```
tbl.RowGroups.Remove(tbl.RowGroups[0]);
```

```
tbl.RowGroups.Remove(tbl.RowGroups(0))
```

## Example

The following example removes a particular row group by index.

```
tbl.RowGroups.RemoveAt(0);
```

```
tbl.RowGroups.RemoveAt(0)
```

## Example

The following example removes all row groups from the table's row groups collection.

```
tbl.RowGroups.Clear();
```

```
tbl.RowGroups.Clear()
```

## See also

- [How-to: Manipulate Flow Content Elements through the Inlines Property](#)
- [Manipulate a FlowDocument through the Blocks Property](#)
- [Manipulate a Table's Columns through the Columns Property](#)

# How to: Use Flow Content Elements

4/28/2019 • 2 minutes to read • [Edit Online](#)

The following example demonstrates declarative usage for various flow content elements and associated attributes. Elements and attributes demonstrated include:

- [Bold](#) element
- [BreakPageBefore](#) attribute
- [FontSize](#) attribute
- [Italic](#) element
- [LineBreak](#) element
- [List](#) element
- [ListItem](#) element
- [Paragraph](#) element
- [Run](#) element
- [Section](#) element
- [Span](#) element
- [Variants](#) attribute (superscript and subscript)
- [Underline](#) element

## Example

```
<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>
    <Paragraph FontSize="18">Flow Format Example</Paragraph>

    <Paragraph>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy
        nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi
        enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis
        nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
    </Paragraph>
    <Paragraph>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh
        euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim
        ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl
        ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
    </Paragraph>

    <Paragraph FontSize="18">More flow elements</Paragraph>
    <Paragraph FontSize="15">Inline, font type and weight, and a List</Paragraph>

    <List>
        <ListItem><Paragraph>ListItem 1</Paragraph></ListItem>
        <ListItem><Paragraph>ListItem 2</Paragraph></ListItem>
        <ListItem><Paragraph>ListItem 3</Paragraph></ListItem>
    </List>
```

```
<List><List Item="1">List Item 4</List Item>
<List Item="2">List Item 5</List Item>
</List>

<Paragraph><Bold>Bolded</Bold></Paragraph>
<Paragraph><Underline>Underlined</Underline></Paragraph>
<Paragraph><Bold><Underline>Bolded and Underlined</Underline></Bold></Paragraph>
<Paragraph><Italic>Italic</Italic></Paragraph>

<Paragraph><Span>The Span element, no inherent rendering</Span></Paragraph>
<Paragraph><Run>The Run element, no inherent rendering</Run></Paragraph>

<Paragraph FontSize="15">Subscript, Superscript</Paragraph>

<Paragraph>
    <Run Typography.Variants="Superscript">This text is Superscripted.</Run> This text isn't.
</Paragraph>
<Paragraph>
    <Run Typography.Variants="Subscript">This text is Subscripted.</Run> This text isn't.
</Paragraph>
<Paragraph>
    If a font does not support a particular form (such as Superscript) a default font form will be
    displayed.
</Paragraph>

<Paragraph FontSize="15">Blocks, breaks, paragraph</Paragraph>

<Section><Paragraph>A block section of text</Paragraph></Section>
<Section><Paragraph>Another block section of text</Paragraph></Section>

<Paragraph><LineBreak/></Paragraph>
<Section><Paragraph>... and another section, preceded by a LineBreak</Paragraph></Section>

<Section BreakPageBefore="True"/>
<Section><Paragraph>... and another section, preceded by a PageBreak</Paragraph></Section>

<Paragraph>Finally, a paragraph. Note the break between this paragraph ...</Paragraph>
<Paragraph TextIndent="25">... and this paragraph, and also the left indentation.</Paragraph>

<Paragraph><LineBreak/></Paragraph>

</FlowDocument>
```

# How to: Use FlowDocument Column-Separating Attributes

3/25/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use the column-separating features of a [FlowDocument](#).

## Example

The following example defines a [FlowDocument](#), and sets the [ColumnGap](#), [ColumnRuleBrush](#), and [ColumnRuleWidth](#) attributes. The [FlowDocument](#) contains a single paragraph of sample content.

```
<FlowDocumentReader>
  <FlowDocument
    ColumnGap="20.0"
    ColumnRuleBrush="DodgerBlue"
    ColumnRuleWidth="5.0"
    ColumnWidth="140.0"
  >
    <Paragraph Background="AntiqueWhite" TextAlignment="Left">
      This paragraph has the background set to antique white to make its
      boundaries obvious.

      The column gap is the space between columns; this FlowDocument will
      have a column gap of 20 device-independend pixels. The column rule
      is a vertical line drawn in the column gap, and is used to visually
      separate columns; this FlowDocument a Dodger-blue column rule that
      is 5 pixels wide.

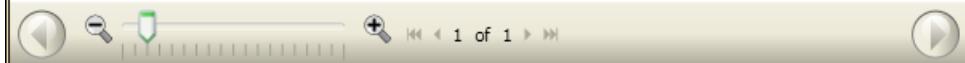
      The column rule and column gap both take space between columns. In
      this case, a column gap width of 20 plus a column rule of width of 5
      results in the space between columns being 25 pixels wide, 5 pixels
      for the column rule, and 10 pixels of column gap on either side of the column rule.
    </Paragraph>
  </FlowDocument>
</FlowDocumentReader>
```

The following figure shows the effects of the [ColumnGap](#), [ColumnRuleBrush](#), and [ColumnRuleWidth](#) attributes in a rendered [FlowDocument](#).

This paragraph has the background set to antique white to make its boundaries obvious. The column gap is the space between columns; this FlowDocument will have a column gap of 20 device-independend pixels. The column rule is a vertical line drawn

in the column gap, and is used to visually separate columns; this FlowDocument a Dodger-blue column rule that is 5 pixels wide. The column rule and column gap both take space between columns. In this case, a column gap width of 20 plus a column rule of

width of 5 results in the space between columns being 25 pixels wide, 5 pixels for the column rule, and 10 pixels of column gap on either side of the column rule.



# Typography

10/7/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) includes support for rich presentation of text content. Text in WPF is rendered using Microsoft ClearType, which enhances the clarity and readability of text. WPF also supports OpenType fonts, which provide additional capabilities beyond those defined by the TrueType® format.

## In This Section

[Typography in WPF](#)

[ClearType Overview](#)

[ClearType Registry Settings](#)

[Drawing Formatted Text](#)

[Advanced Text Formatting](#)

[Fonts](#)

[Glyphs](#)

[How-to Topics](#)

## See also

- [Typography](#)
- [Documents in WPF](#)
- [OpenType Font Features](#)
- [Optimizing WPF Application Performance](#)

# Typography in WPF

11/7/2019 • 6 minutes to read • [Edit Online](#)

This topic introduces the major typographic features of WPF. These features include improved quality and performance of text rendering, OpenType typography support, enhanced international text, enhanced font support, and new text application programming interfaces (APIs).

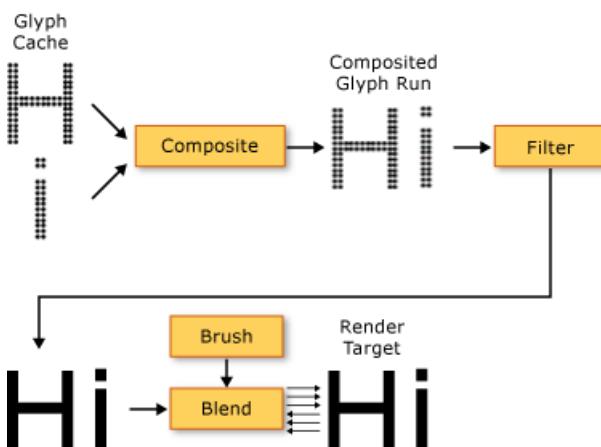
## Improved Quality and Performance of Text

Text in WPF is rendered using Microsoft ClearType, which enhances the clarity and readability of text. ClearType is a software technology developed by Microsoft that improves the readability of text on existing LCDs (Liquid Crystal Displays), such as laptop screens, Pocket PC screens and flat panel monitors. ClearType uses sub-pixel rendering which allows text to be displayed with a greater fidelity to its true shape by aligning characters on a fractional part of a pixel. The extra resolution increases the sharpness of the tiny details in text display, making it much easier to read over long durations. Another improvement of ClearType in WPF is y-direction anti-aliasing, which smoothes the tops and bottoms of shallow curves in text characters. For more details on ClearType features, see [ClearType Overview](#).



Text with ClearType y-direction antialiasing

The entire text rendering pipeline can be hardware-accelerated in WPF provided your machine meets the minimum level of hardware required. Rendering that cannot be performed using hardware falls back to software rendering. Hardware-acceleration affects all phases of the text rendering pipeline—from storing individual glyphs, compositing glyphs into glyph runs, applying effects, to applying the ClearType blending algorithm to the final displayed output. For more information on hardware acceleration, see [Graphics Rendering Tiers](#).



In addition, animated text, whether by character or glyph, takes full advantage of the graphics hardware capability enabled by WPF. This results in smooth text animation.

## Rich Typography

The OpenType font format is an extension of the TrueType® font format. The OpenType font format was developed jointly by Microsoft and Adobe, and provides a rich assortment of advanced typographic features. The [Typography](#) object exposes many of the advanced features of OpenType fonts, such as stylistic alternates and

swashes. The Windows SDK provides a set of sample OpenType fonts that are designed with rich features, such as the Pericles and Pescadero fonts. For more information, see [Sample OpenType Font Pack](#).

The Pericles OpenType font contains additional glyphs that provide stylistic alternates to the standard set of glyphs. The following text displays stylistic alternate glyphs.

## ANCIENT GREEK MYTHOLOGY

Swashes are decorative glyphs that use elaborate ornamentation often associated with calligraphy. The following text displays standard and swash glyphs for the Pescadero font.

A B C D E F G H I J K L M N  
ΑΒ ΚΔΕΦ ΓΗΙΚΛΜΝ

For more details on OpenType features, see [OpenType Font Features](#).

## Enhanced International Text Support

WPF provides enhanced international text support by providing the following features:

- Automatic line-spacing in all writing systems, using adaptive measurement.
- Broad support for international text. For more information, see [Globalization for WPF](#).
- Language-guided line breaking, hyphenation, and justification.

## Enhanced Font Support

WPF provides enhanced font support by providing the following features:

- Unicode for all text. Font behavior and selection no longer require charset or codepage.
- Font behavior independent of global settings, such as system locale.
- Separate [FontWeight](#), [FontStretch](#), and [FontStyle](#) types for defining a [FontFamily](#). This provides greater flexibility than in Win32 programming, in which Boolean combinations of italic and bold are used to define a font family.
- Writing direction (horizontal versus vertical) handled independent of font name.
- Font linking and font fallback in a portable XML file, using composite font technology. Composite fonts allow for the construction of full range multilingual fonts. Composite fonts also provide a mechanism that avoids displaying missing glyphs. For more information, see the remarks in the [FontFamily](#) class.
- International fonts built from composite fonts, using a group of single-language fonts. This saves on resource costs when developing fonts for multiple languages.
- Composite fonts embedded in a document, thereby providing document portability. For more information, see the remarks in the [FontFamily](#) class.

## New Text Application Programming Interfaces (APIs)

WPF provides several text APIs for developers to use when including text in their applications. These APIs are grouped into three categories:

- **Layout and user interface.** The common text controls for the graphical user interface (GUI).

- **Lightweight text drawing.** Allows you to draw text directly to objects.
- **Advanced text formatting.** Allows you to implement a custom text engine.

## Layout and User Interface

At the highest level of functionality, the text APIs provide common user interface (UI) controls such as [Label](#), [TextBlock](#), and [TextBox](#). These controls provide the basic UI elements within an application, and offer an easy way to present and interact with text. Controls such as [RichTextBox](#) and [PasswordBox](#) enable more advanced or specialized text-handling. And classes such as [TextRange](#), [TextSelection](#), and [TextPointer](#) enable useful text manipulation. These UI controls provide properties such as [FontFamily](#), [FontSize](#), and [FontStyle](#), which enable you to control the font that is used to render the text.

### Using Bitmap Effects, Transforms, and Text Effects

WPF allows you to create visually interesting uses of text by uses features such as bitmap effects, transforms, and text effects. The following example shows a typical type of a drop shadow effect applied to text.

## Shadow Text

The following example shows a drop shadow effect and noise applied to text.

## Shadow Text

The following example shows an outer glow effect applied to text.

## Shadow Text

The following example shows a blur effect applied to text.

## Shadow Text

The following example shows the second line of text scaled by 150% along the x-axis, and the third line of text scaled by 150% along the y-axis.

## Scaled Text

## Scaled Text

## Scaled Text

The following example shows text skewed along the x-axis.

## Skewed Text

## Skewed Text

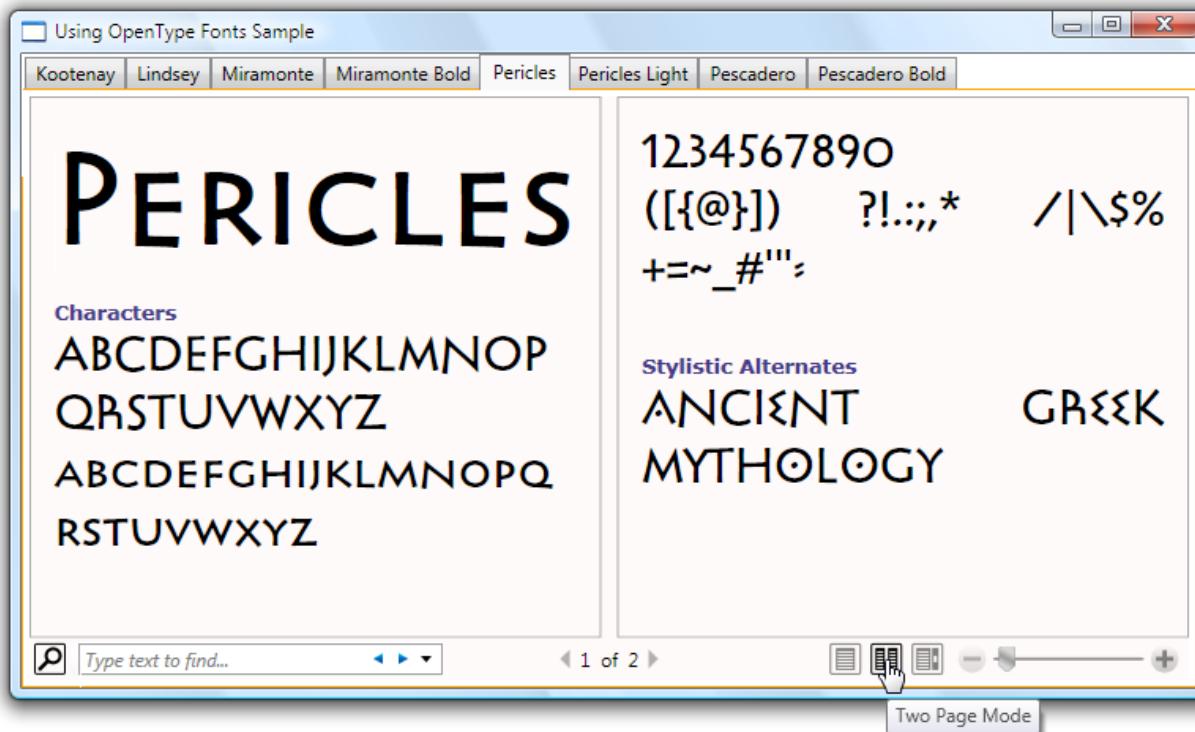
A [TextEffect](#) object is a helper object that allows you to treat text as one or more groups of characters in a text string. The following example shows an individual character being rotated. Each character is rotated independently at 1-second intervals.

# Windows Vista

## Using Flow Documents

In addition to the common UI controls, WPF offers a layout control for text presentation—the [FlowDocument](#) element. The [FlowDocument](#) element, in conjunction with the [DocumentViewer](#) element, provides a control for large amounts of text with varying layout requirements. Layout controls provide access to advanced typography through the [Typography](#) object and font-related properties of other UI controls.

The following example shows text content hosted in a [FlowDocumentReader](#), which provides search, navigation, pagination, and content scaling support.



For more information, see [Documents in WPF](#).

## Lightweight Text Drawing

You can draw text directly to WPF objects by using the [DrawText](#) method of the [DrawingContext](#) object. To use this method, you create a [FormattedText](#) object. This object allows you to draw multi-line text, in which each character in the text can be individually formatted. The functionality of the [FormattedText](#) object contains much of the functionality of the [DrawText](#) flags in the Windows API. In addition, the [FormattedText](#) object contains functionality such as ellipsis support, in which an ellipsis is displayed when text exceeds its bounds. The following example shows text that has several formats applied to it, including a linear gradient on the second and third words.

**L**orem **i**psum  
**d**olor sit amet,  
*c*onsectetur  
*a*dipisicing elit,  
sed do eiusmod...

You can convert formatted text into [Geometry](#) objects, allowing you to create other types of visually interesting text. For example, you could create a [Geometry](#) object based on the outline of a text string.

# Spectrum Outline

The following examples illustrate several ways of creating interesting visual effects by modifying the stroke, fill, and highlight of converted text.

*Fancy Outlined Text*

BUTTERFLIES

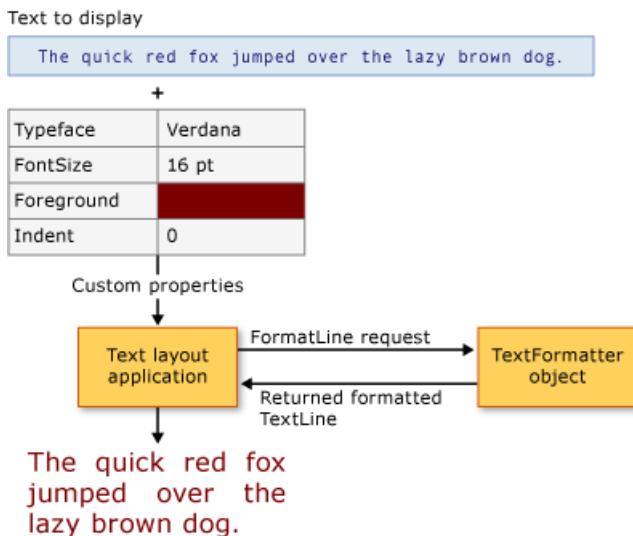
WILDFIRE

For more information on the [FormattedText](#) object, see [Drawing Formatted Text](#).

## Advanced Text Formatting

At the most advanced level of the text APIs, WPF offers you the ability to create custom text layout by using the [TextFormatter](#) object and other types in the [System.Windows.Media.TextFormatting](#) namespace. The [TextFormatter](#) and associated classes allow you to implement custom text layout that supports your own definition of character formats, paragraph styles, line breaking rules, and other layout features for international text. There are very few cases in which you would want to override the default implementation of the WPF text layout support. However, if you were creating a text editing control or application, you might require a different implementation than the default WPF implementation.

Unlike a traditional text API, the [TextFormatter](#) interacts with a text layout client through a set of callback methods. It requires the client to provide these methods in an implementation of the [TextSource](#) class. The following diagram illustrates the text layout interaction between the client application and [TextFormatter](#).



For more details on creating custom text layout, see [Advanced Text Formatting](#).

## See also

- [FormattedText](#)
- [TextFormatter](#)
- [ClearType Overview](#)
- [OpenType Font Features](#)
- [Drawing Formatted Text](#)
- [Advanced Text Formatting](#)
- [Text](#)
- [Microsoft Typography](#)

# ClearType Overview

8/24/2019 • 3 minutes to read • [Edit Online](#)

This topic provides an overview of the Microsoft ClearType technology found in the Windows Presentation Foundation (WPF).

## Technology Overview

ClearType is a software technology developed by Microsoft that improves the readability of text on existing LCDs (Liquid Crystal Displays), such as laptop screens, Pocket PC screens and flat panel monitors. ClearType works by accessing the individual vertical color stripe elements in every pixel of an LCD screen. Before ClearType, the smallest level of detail that a computer could display was a single pixel, but with ClearType running on an LCD monitor, we can now display features of text as small as a fraction of a pixel in width. The extra resolution increases the sharpness of the tiny details in text display, making it much easier to read over long durations.

The ClearType available in Windows Presentation Foundation (WPF) is the latest generation of ClearType which has several improvements over version found in Microsoft Windows Graphics Device Interface (GDI).

## Sub-pixel Positioning

A significant improvement over the previous version of ClearType is the use of sub-pixel positioning. Unlike the ClearType implementation found in GDI, the ClearType found in Windows Presentation Foundation (WPF) allows glyphs to start within the pixel and not just the beginning boundary of the pixel. Because of this extra resolution in positioning glyphs, the spacing and proportions of the glyphs is more precise and consistent.

The following two examples show how glyphs may begin on any sub-pixel boundary when sub-pixel positioning is used. The example on the left is rendered using the earlier version of the ClearType renderer, which did not employ sub-pixel positioning. The example on the right is rendered using the new version of the ClearType renderer, using sub-pixel positioning. Note how each **e** and **I** in the right-hand image is rendered slightly differently because each starts on a different sub-pixel. When viewing the text at its normal size on the screen, this difference is not noticeable because of the high contrast of the glyph image. This is only possible because of sophisticated color filtering that is incorporated in ClearType.

Earlier version of ClearType



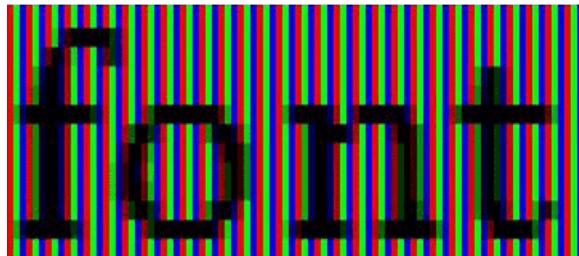
Later version of ClearType



Text displayed with earlier and later versions of ClearType

The following two examples compare output from the earlier ClearType renderer with the new version of the ClearType renderer. The subpixel positioning, shown on the right, greatly improves the spacing of type on screen, especially at small sizes where the difference between a sub-pixel and a whole pixel represents a significant proportion of glyph width. Note that spacing between the letters is more even in the second image. The cumulative benefit of sub-pixel positioning to the overall appearance of a screen of text is greatly increased, and represents a significant evolution in ClearType technology.

Earlier version of ClearType



Later version of ClearType

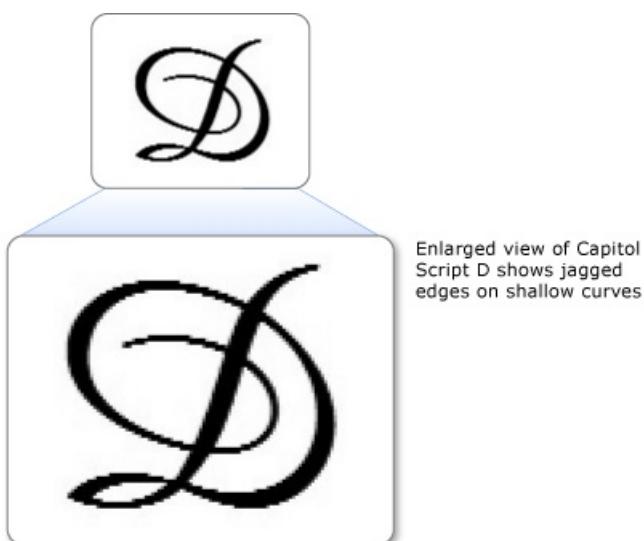


Text with earlier and later versions of ClearType

## Y-Direction Antialiasing

Another improvement of ClearType in Windows Presentation Foundation (WPF) is y-direction anti-aliasing. The ClearType in GDI without y-direction anti-aliasing provides better resolution on the x-axis but not the y-axis. On the tops and bottoms of shallow curves, the jagged edges detract from its readability.

The following example shows the effect of having no y-direction antialiasing. In this case, the jagged edges on the top and bottom of the letter are apparent.

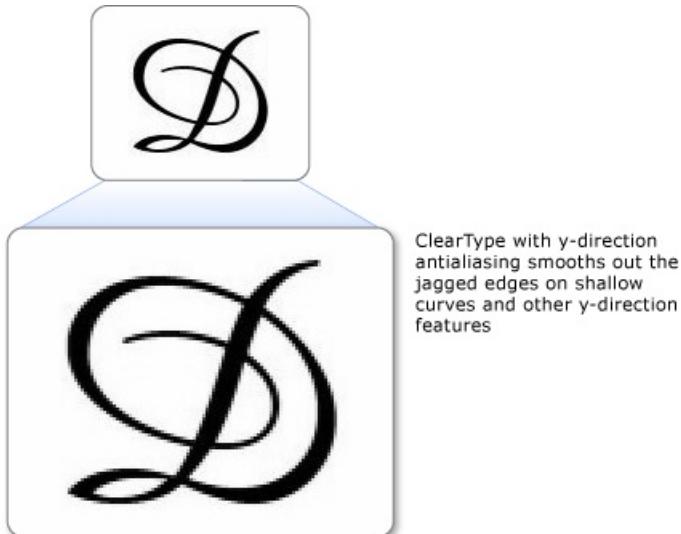


Text with jagged edges on shallow curves

ClearType in Windows Presentation Foundation (WPF) provides antialiasing on the y-direction level to smooth out any jagged edges. This is particularly important for improving the readability of East Asian languages where ideographs have an almost equal amount of horizontal and vertical shallow curves.

The following example shows the effect of y-direction antialiasing. In this case, the top and bottom of the letter

show a smooth curve.



Text with ClearType y-direction antialiasing

## Hardware Acceleration

ClearType in Windows Presentation Foundation (WPF) can take advantage of hardware acceleration for better performance and to reduce CPU load and system memory requirements. By using the pixel shaders and video memory of a graphics card, ClearType provides faster rendering of text, particularly when animation is used.

ClearType in Windows Presentation Foundation (WPF) does not modify the system-wide ClearType settings. Disabling ClearType in Windows sets Windows Presentation Foundation (WPF) antialiasing to grayscale mode. In addition, ClearType in Windows Presentation Foundation (WPF) does not modify the settings of the [ClearType Tuner PowerToy](#).

One of the Windows Presentation Foundation (WPF) architectural design decisions is to have resolution independent layout better support higher resolution DPI monitors, which are becoming more widespread. This has the consequence of Windows Presentation Foundation (WPF) not supporting aliased text rendering or the bitmaps in some East Asian fonts because they are both resolution dependent.

## Further Information

[ClearType Information](#)

[ClearType Tuner PowerToy](#)

## See also

- [ClearType Registry Settings](#)

# ClearType Registry Settings

9/20/2019 • 4 minutes to read • [Edit Online](#)

This topic provides an overview of the Microsoft ClearType registry settings that are used by WPF applications.

## Technology Overview

WPF applications that render text to a display device use ClearType features to provide an enhanced reading experience. ClearType is a software technology developed by Microsoft that improves the readability of text on existing LCDs (Liquid Crystal Displays), such as laptop screens, Pocket PC screens and flat panel monitors.

ClearType works by accessing the individual vertical color stripe elements in every pixel of an LCD screen. For more information on ClearType, see [ClearType Overview](#).

Text that is rendered with ClearType can appear significantly different when viewed on various display devices. For example, a small number of monitors implement the color stripe elements in blue, green, red order rather than the more common red, green, blue (RGB) order.

Text that is rendered with ClearType can also appear significantly different when viewed by individuals with varying levels of color sensitivity. Some individuals can detect slight differences in color better than others.

In each of these cases, ClearType features need to be modified in order to provide the best reading experience for each individual.

## Registry Settings

WPF specifies four registry settings for controlling ClearType features:

SETTING	DESCRIPTION
ClearType level	Describes the level of ClearType color clarity.
Gamma level	Describes the level of the pixel color component for a display device.
Pixel structure	Describes the arrangement of pixels for a display device.
Text contrast level	Describes the level of contrast for displayed text.

These settings can be accessed by an external configuration utility that knows how to reference the identified WPFClearType registry settings. These settings can also be created or modified by accessing the values directly by using the Windows Registry Editor.

If the WPFClearType registry settings are not set (which is the default state), the WPF application queries the Windows system parameters information for font smoothing settings.

### NOTE

For information on enumerating display device names, see the `SystemParametersInfo` Win32 function.

## ClearType Level

The ClearType level allows you to adjust the rendering of text based on the color sensitivity and perception of an individual. For some individuals, the rendering of text that uses ClearType at its highest level does not produce the best reading experience.

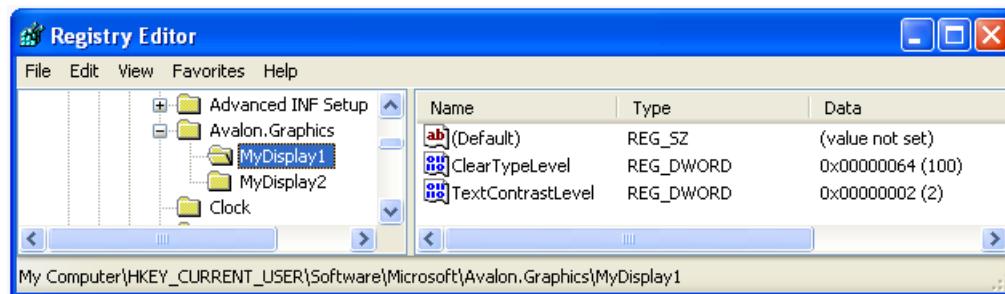
The ClearType level is an integer value that ranges from 0 to 100. The default level is 100, which means ClearType uses the maximum capability of the color stripe elements of the display device. However, a ClearType level of 0 renders text as gray scale. By setting the ClearType level somewhere between 0 and 100, you can create an intermediate level that is suitable to an individual's color sensitivity.

### Registry Setting

The registry setting location for the ClearType level is an individual user setting that corresponds to a specific display device name:

HKEY\_CURRENT\_USER\Software\Microsoft\Avalon.Graphics\<displayName>

For each display device name for a user, a `ClearTypeLevel` DWORD value is defined. The following screenshot shows the Registry Editor setting for the ClearType level.



#### NOTE

WPF applications render text in one of either two modes, with and without ClearType. When text is rendered without ClearType, it is referred to as gray scale rendering.

## Gamma Level

The gamma level refers to the nonlinear relationship between a pixel value and luminance. The gamma level setting should correspond to the physical characteristics of the display device; otherwise, distortions in rendered output may occur. For example, text may appear too wide or too narrow, or color fringes may appear on the edges of vertical stems of glyphs.

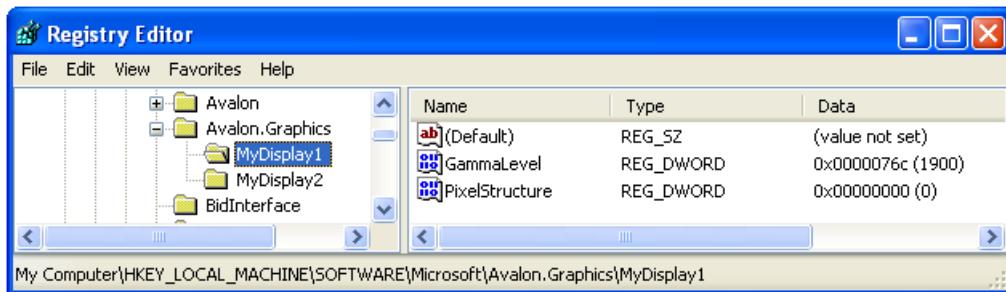
The gamma level is an integer value that ranges from 1000 to 2200. The default level is 1900.

### Registry Setting

The registry setting location for the gamma level is a local machine setting that corresponds to a specific display device name:

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Avalon.Graphics\<displayName>

For each display device name for a user, a `GammaLevel` DWORD value is defined. The following screenshot shows the Registry Editor setting for the gamma level.



## Pixel Structure

The pixel structure describes the type of pixels that make up a display device. The pixel structure is defined as one of three types:

Type	Value	Description
Flat	0	The display device has no pixel structure. This means that light sources for each color are spread equally on the pixel area—this is referred to as gray scale rendering. This is how a standard display device works. ClearType is never applied to the rendered text.
RGB	1	The display device has pixels that consist of three stripes in the following order: red, green, and blue. ClearType is applied to the rendered text.
BGR	2	The display device has pixels that consist of three stripes in the following order: blue, green, and red. ClearType is applied to the rendered text. Notice how the order is inverted from the RGB type.

The pixel structure corresponds to an integer value that ranges from 0 to 2. The default level is 0, which represents a flat pixel structure.

### Note

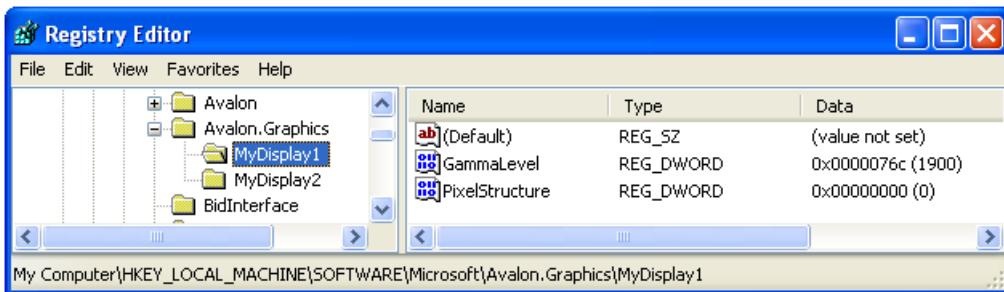
For information on enumerating display device names, see the [EnumDisplayDevices](#) Win32 function.

## Registry Setting

The registry setting location for the pixel structure is a local machine setting that corresponds to a specific display device name:

`HKEY_LOCAL_MACHINE\Software\Microsoft\Avalon.Graphics\<display Name>`

For each display device name for a user, a `PixelStructure` DWORD value is defined. The following screenshot shows the Registry Editor setting for the pixel structure.



## Text Contrast Level

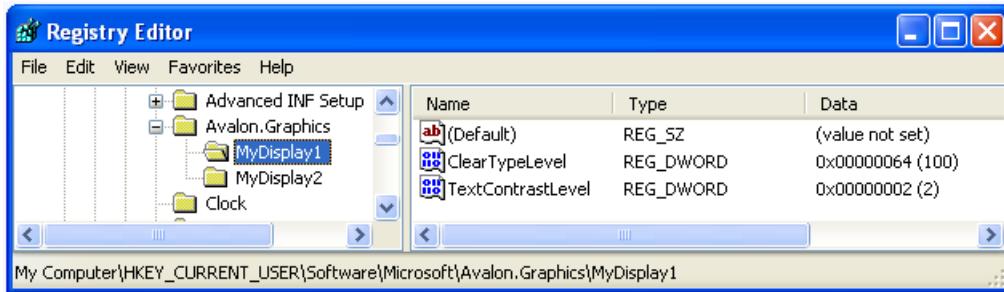
The text contrast level allows you to adjust the rendering of text based on the stem widths of glyphs. The text contrast level is an integer value that ranges from 0 to 6—the larger the integer value, the wider the stem. The default level is 1.

### Registry Setting

The registry setting location for the text contrast level is an individual user setting that corresponds to a specific display device name:

`HKEY_CURRENT_USER\Software\Microsoft\Avalon.Graphics\<displayName>`

For each display device name for a user, a `TextContrastLevel` DWORD value is defined. The following screenshot shows the Registry Editor setting for the text contrast level.



## See also

- [ClearType Overview](#)
- [ClearType Antialiasing](#)

# Drawing Formatted Text

8/22/2019 • 7 minutes to read • [Edit Online](#)

This topic provides an overview of the features of the [FormattedText](#) object. This object provides low-level control for drawing text in Windows Presentation Foundation (WPF) applications.

## Technology Overview

The [FormattedText](#) object allows you to draw multi-line text, in which each character in the text can be individually formatted. The following example shows text that has several formats applied to it.

**Lorem ipsum**  
**dolor sit amet,**  
*consectetur*  
*adipisicing elit,*  
sed do eiusmod...

### NOTE

For those developers migrating from the Win32 API, the table in the [Win32 Migration](#) section lists the Win32 `DrawText` flags and the approximate equivalent in Windows Presentation Foundation (WPF).

## Reasons for Using Formatted Text

WPF includes multiple controls for drawing text to the screen. Each control is targeted to a different scenario and has its own list of features and limitations. In general, the [TextBlock](#) element should be used when limited text support is required, such as a brief sentence in a user interface (UI). [Label](#) can be used when minimal text support is required. For more information, see [Documents in WPF](#).

The [FormattedText](#) object provides greater text formatting features than Windows Presentation Foundation (WPF) text controls, and can be useful in cases where you want to use text as a decorative element. For more information, see the following section [Converting Formatted Text to a Geometry](#).

In addition, the [FormattedText](#) object is useful for creating text-oriented [DrawingVisual](#)-derived objects. [DrawingVisual](#) is a lightweight drawing class that is used to render shapes, images, or text. For more information, see [Hit Test Using DrawingVisuals Sample](#).

## Using the FormattedText Object

To create formatted text, call the [FormattedText](#) constructor to create a [FormattedText](#) object. Once you have created the initial formatted text string, you can apply a range of formatting styles.

Use the [MaxTextWidth](#) property to constrain the text to a specific width. The text will automatically wrap to avoid exceeding the specified width. Use the [MaxTextHeight](#) property to constrain the text to a specific height. The text will display an ellipsis, "..." for the text that exceeds the specified height.

→  
Lorem ipsum dolor  
sit amet,  
consectetur...

Text wordwraps when it  
exceeds the width

Text shows ellipsis when  
it exceeds the height

adipisicing elit, sed do eiusmod tempor

You can apply multiple formatting styles to one or more characters. For example, you could call both the [SetFontSize](#) and [SetForegroundBrush](#) methods to change the formatting of the first five characters in the text.

The following code example creates a [FormattedText](#) object and then applies several formatting styles to the text.

```
protected override void OnRender(DrawingContext drawingContext)
{
    string testString = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor";

    // Create the initial formatted text string.
    FormattedText formattedText = new FormattedText(
        testString,
        CultureInfo.GetCultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface("Verdana"),
        32,
        Brushes.Black);

    // Set a maximum width and height. If the text overflows these values, an ellipsis "..." appears.
    formattedText.MaxTextWidth = 300;
    formattedText.MaxTextHeight = 240;

    // Use a larger font size beginning at the first (zero-based) character and continuing for 5 characters.
    // The font size is calculated in terms of points -- not as device-independent pixels.
    formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5);

    // Use a Bold font weight beginning at the 6th character and continuing for 11 characters.
    formattedText.SetFontWeight(FontWeights.Bold, 6, 11);

    // Use a linear gradient brush beginning at the 6th character and continuing for 11 characters.
    formattedText.SetForegroundBrush(
        new LinearGradientBrush(
            Colors.Orange,
            Colors.Teal,
            90.0),
        6, 11);

    // Use an Italic font style beginning at the 28th character and continuing for 28 characters.
    formattedText.SetFontStyle(FontStyles.Italic, 28, 28);

    // Draw the formatted text string to the DrawingContext of the control.
    drawingContext.DrawText(formattedText, new Point(10, 0));
}
```

```

Protected Overrides Sub OnRender(ByVal drawingContext As DrawingContext)
    Dim testString As String = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor"

    ' Create the initial formatted text string.
    Dim formattedText As New FormattedText(testString, CultureInfo.GetCultureInfo("en-us"),
    FlowDirection.LeftToRight, New Typeface("Verdana"), 32, Brushes.Black)

    ' Set a maximum width and height. If the text overflows these values, an ellipsis "..." appears.
    formattedText.MaxTextWidth = 300
    formattedText.MaxTextHeight = 240

    ' Use a larger font size beginning at the first (zero-based) character and continuing for 5 characters.
    ' The font size is calculated in terms of points -- not as device-independent pixels.
    formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5)

    ' Use a Bold font weight beginning at the 6th character and continuing for 11 characters.
    formattedText.SetFontWeight(FontWeights.Bold, 6, 11)

    ' Use a linear gradient brush beginning at the 6th character and continuing for 11 characters.
    formattedText.SetForegroundBrush(New LinearGradientBrush(Colors.Orange, Colors.Teal, 90.0), 6, 11)

    ' Use an Italic font style beginning at the 28th character and continuing for 28 characters.
    formattedText.SetFontStyle(FontStyles.Italic, 28, 28)

    ' Draw the formatted text string to the DrawingContext of the control.
    drawingContext.DrawText(formattedText, New Point(10, 0))
End Sub

```

## Font Size Unit of Measure

As with other text objects in Windows Presentation Foundation (WPF) applications, the [FormattedText](#) object uses device-independent pixels as the unit of measure. However, most Win32 applications use points as the unit of measure. If you want to use display text in units of points in Windows Presentation Foundation (WPF) applications, you need to convert device-independent units (1/96th inch per unit) to points. The following code example shows how to perform this conversion.

```
// The font size is calculated in terms of points -- not as device-independent pixels.
formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5);
```

```
' The font size is calculated in terms of points -- not as device-independent pixels.
formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5)
```

## Converting Formatted Text to a Geometry

You can convert formatted text into [Geometry](#) objects, allowing you to create other types of visually interesting text. For example, you could create a [Geometry](#) object based on the outline of a text string.

# Spectrum Outline

The following examples illustrate several ways of creating interesting visual effects by modifying the stroke, fill, and highlight of converted text.

# Fancy Outlined Text

# BUTTERFLIES

# WILDFIRE

When text is converted to a [Geometry](#) object, it is no longer a collection of characters—you cannot modify the characters in the text string. However, you can affect the appearance of the converted text by modifying its stroke and fill properties. The stroke refers to the outline of the converted text; the fill refers to the area inside the outline of the converted text. For more information, see [Create Outlined Text](#).

You can also convert formatted text to a [PathGeometry](#) object, and use the object for highlighting the text. For example, you could apply an animation to the [PathGeometry](#) object so that the animation follows the outline of the formatted text.

The following example shows formatted text that has been converted to a [PathGeometry](#) object. An animated ellipse follows the path of the strokes of the rendered text.

# Hello World!

Sphere following the path geometry of text

For more information, see [How to: Create a PathGeometry Animation for Text](#).

You can create other interesting uses for formatted text once it has been converted to a [PathGeometry](#) object. For example, you can clip video to display inside it.



## Win32 Migration

The features of [FormattedText](#) for drawing text are similar to the features of the Win32 `DrawText` function. For those developers migrating from the Win32 API, the following table lists the Win32 `DrawText` flags and the approximate equivalent in Windows Presentation Foundation (WPF).

DRAW TEXT FLAG	WPF EQUIVALENT	NOTES
<code>DT_BOTTOM</code>	<code>Height</code>	Use the <a href="#">Height</a> property to compute an appropriate Win32 <code>DrawText</code> 'y' position.

DRAW TEXT FLAG	WPF EQUIVALENT	NOTES
DT_CALCRECT	Height, Width	Use the <a href="#">Height</a> and <a href="#">Width</a> properties to calculate the output rectangle.
DT_CENTER	TextAlignment	Use the <a href="#">TextAlignment</a> property with the value set to <a href="#">Center</a> .
DT_EDITCONTROL	None	Not required. Space width and last line rendering are the same as in the framework edit control.
DT_END_ELLIPSIS	Trimming	Use the <a href="#">Trimming</a> property with the value <a href="#">CharacterEllipsis</a> .  Use <a href="#">WordEllipsis</a> to get Win32 DT_END_ELLIPSIS with DT_WORD_ELLIPSIS end ellipsis—in this case, character ellipsis only occurs on words that do not fit on a single line.
DT_EXPAND_TABS	None	Not required. Tabs are automatically expanded to stops every 4 ems, which is approximately the width of 8 language-independent characters.
DT_EXTERNALLEADING	None	Not required. External leading is always included in line spacing. Use the <a href="#">LineHeight</a> property to create user-defined line spacing.
DT_HIDEPREFIX	None	Not supported. Remove the '&' from the string before constructing the <a href="#">FormattedText</a> object.
DT_LEFT	TextAlignment	This is the default text alignment. Use the <a href="#">TextAlignment</a> property with the value set to <a href="#">Left</a> . (WPF only)
DT_MODIFYSTRING	None	Not supported.
DT_NOCLIP	VisualClip	Clipping does not happen automatically. If you want to clip text, use the <a href="#">VisualClip</a> property.
DT_NOFULLWIDTHCHARBREAK	None	Not supported.
DT_NOPREFIX	None	Not required. The '&' character in strings is always treated as a normal character.
DT_PATHELLIPSIS	None	Use the <a href="#">Trimming</a> property with the value <a href="#">WordEllipsis</a> .

DRAW TEXT FLAG	WPF EQUIVALENT	NOTES
DT_PREFIX	None	Not supported. If you want to use underscores for text, such as an accelerator key or link, use the <a href="#">SetTextDecorations</a> method.
DT_PREFIXONLY	None	Not supported.
DT_RIGHT	<a href="#">TextAlignment</a>	Use the <a href="#">TextAlignment</a> property with the value set to <a href="#">Right</a> . (WPF only)
DT_RTLREADING	<a href="#">FlowDirection</a>	Set the <a href="#">FlowDirection</a> property to <a href="#">RightToLeft</a> .
DT_SINGLELINE	None	Not required. <a href="#">FormattedText</a> objects behave as a single line control, unless either the <a href="#">MaxTextWidth</a> property is set or the text contains a carriage return/line feed (CR/LF).
DT_TABSTOP	None	No support for user-defined tab stop positions.
DT_TOP	<a href="#">Height</a>	Not required. Top justification is the default. Other vertical positioning values can be defined by using the <a href="#">Height</a> property to compute an appropriate Win32 DrawText 'y' position.
DT_VCENTER	<a href="#">Height</a>	Use the <a href="#">Height</a> property to compute an appropriate Win32 DrawText 'y' position.
DT_WORDBREAK	None	Not required. Word breaking happens automatically with <a href="#">FormattedText</a> objects. You cannot disable it.
DT_WORD_ELLIPSIS	<a href="#">Trimming</a>	Use the <a href="#">Trimming</a> property with the value <a href="#">WordEllipsis</a> .

## See also

- [FormattedText](#)
- [Documents in WPF](#)
- [Typography in WPF](#)
- [Create Outlined Text](#)
- [How to: Create a PathGeometry Animation for Text](#)

# Advanced Text Formatting

10/16/2019 • 7 minutes to read • [Edit Online](#)

The Windows Presentation Foundation (WPF) provides a robust set of APIs for including text in your application. Layout and user interface (UI) APIs, such as [TextBlock](#), provide the most common and general use elements for text presentation. Drawing APIs, such as [GlyphRunDrawing](#) and [FormattedText](#), provide a means for including formatted text in drawings. At the most advanced level, WPF provides an extensible text formatting engine to control every aspect of text presentation, such as text store management, text run formatting management, and embedded object management.

This topic provides an introduction to WPF text formatting. It focuses on client implementation and use of the WPF text formatting engine.

**NOTE**

All code examples within this document can be found in the [Advanced Text Formatting Sample](#).

## Prerequisites

This topic assumes that you are familiar with the higher level APIs used for text presentation. Most user scenarios will not require the advanced text formatting APIs discussed in this topic. For an introduction to the different text APIs, see [Documents in WPF](#).

## Advanced Text Formatting

The text layout and UI controls in WPF provide formatting properties that allow you to easily include formatted text in your application. These controls expose a number of properties to handle the presentation of text, which includes its typeface, size, and color. Under ordinary circumstances, these controls can handle the majority of text presentation in your application. However, some advanced scenarios require the control of text storage as well as text presentation. WPF provides an extensible text formatting engine for this purpose.

The advanced text formatting features found in WPF consist of a text formatting engine, a text store, text runs, and formatting properties. The text formatting engine, [TextFormatter](#), creates lines of text to be used for presentation. This is achieved by initiating the line formatting process and calling the text formatter's [FormatLine](#). The text formatter retrieves text runs from your text store by calling the store's [GetTextRun](#) method. The [TextRun](#) objects are then formed into [TextLine](#) objects by the text formatter and given to your application for inspection or display.

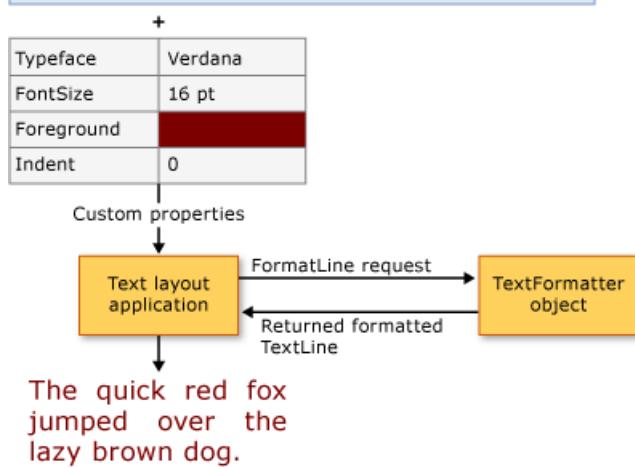
## Using the Text Formatter

[TextFormatter](#) is the WPF text formatting engine and provides services for formatting and breaking text lines. The text formatter can handle different text character formats and paragraph styles, and includes support for international text layout.

Unlike a traditional text API, the [TextFormatter](#) interacts with a text layout client through a set of callback methods. It requires the client to provide these methods in an implementation of the [TextSource](#) class. The following diagram illustrates the text layout interaction between the client application and [TextFormatter](#).

Text to display

```
The quick red fox jumped over the lazy brown dog.
```



The text formatter is used to retrieve formatted text lines from the text store, which is an implementation of [TextSource](#). This is done by first creating an instance of the text formatter by using the [Create](#) method. This method creates an instance of the text formatter and sets the maximum line height and width values. As soon as an instance of the text formatter is created, the line creation process is started by calling the [FormatLine](#) method. [TextFormatter](#) calls back to the text source to retrieve the text and formatting parameters for the runs of text that form a line.

In the following example, the process of formatting a text store is shown. The [TextFormatter](#) object is used to retrieve text lines from the text store and then format the text line for drawing into the [DrawingContext](#).

```
// Create a DrawingGroup object for storing formatted text.
textDest = new DrawingGroup();
DrawingContext dc = textDest.Open();

// Update the text store.
_textStore.Text = textToFormat.Text;
_textStore.FontRendering = _currentRendering;

// Create a TextFormatter object.
TextFormatter formatter = TextFormatter.Create();

// Format each line of text from the text store and draw it.
while (textStorePosition < _textStore.Text.Length)
{
    // Create a textline from the text store using the TextFormatter object.
    using (TextLine myTextLine = formatter.FormatLine(
        _textStore,
        textStorePosition,
        96*6,
        new GenericTextParagraphProperties(_currentRendering),
        null))
    {
        // Draw the formatted text into the drawing context.
        myTextLine.Draw(dc, linePosition, InvertAxes.None);

        // Update the index position in the text store.
        textStorePosition += myTextLine.Length;

        // Update the line position coordinate for the displayed line.
        linePosition.Y += myTextLine.Height;
    }
}

// Persist the drawn text content.
dc.Close();

// Display the formatted text in the DrawingGroup object.
myDrawingBrush.Drawing = textDest;
```

```

' Create a DrawingGroup object for storing formatted text.
textDest = New DrawingGroup()
Dim dc As DrawingContext = textDest.Open()

' Update the text store.
_textStore.Text = textToFormat.Text
_textStore.FontRendering = _currentRendering

' Create a TextFormatter object.
Dim formatter As TextFormatter = TextFormatter.Create()

' Format each line of text from the text store and draw it.
Do While textStorePosition < _textStore.Text.Length
    ' Create a textline from the text store using the TextFormatter object.
    Using myTextLine As TextLine = formatter.FormatLine(_textStore, textStorePosition, 96*6, New
GenericTextParagraphProperties(_currentRendering), Nothing)
        ' Draw the formatted text into the drawing context.
        myTextLine.Draw(dc, linePosition, InvertAxes.None)

        ' Update the index position in the text store.
        textStorePosition += myTextLine.Length

        ' Update the line position coordinate for the displayed line.
        linePosition.Y += myTextLine.Height
    End Using
Loop

' Persist the drawn text content.
dc.Close()

' Display the formatted text in the DrawingGroup object.
myDrawingBrush.Drawing = textDest

```

## Implementing the Client Text Store

When you extend the text formatting engine, you are required to implement and manage all aspects of the text store. This is not a trivial task. The text store is responsible for tracking text run properties, paragraph properties, embedded objects, and other similar content. It also provides the text formatter with individual [TextRun](#) objects which the text formatter uses to create [TextLine](#) objects.

To handle the virtualization of the text store, the text store must be derived from [TextSource](#). [TextSource](#) defines the method the text formatter uses to retrieve text runs from the text store. [GetTextRun](#) is the method used by the text formatter to retrieve text runs used in line formatting. The call to [GetTextRun](#) is repeatedly made by the text formatter until one of the following conditions occurs:

- A [TextEndOfLine](#) or a subclass is returned.
- The accumulated width of text runs exceeds the maximum line width specified in either the call to create the text formatter or the call to the text formatter's [FormatLine](#) method.
- A Unicode newline sequence, such as "CF", "LF", or "CRLF", is returned.

## Providing Text Runs

The core of the text formatting process is the interaction between the text formatter and the text store. Your implementation of [TextSource](#) provides the text formatter with the [TextRun](#) objects and the properties with which to format the text runs. This interaction is handled by the [GetTextRun](#) method, which is called by the text formatter.

The following table shows some of the predefined [TextRun](#) objects.

TEXTRUN TYPE	USAGE
TextCharacters	The specialized text run used to pass a representation of character glyphs back to the text formatter.
TextEmbeddedObject	The specialized text run used to provide content in which measuring, hit testing, and drawing is done in whole, such as a button or image within the text.
TextEndOfLine	The specialized text run used to mark the end of a line.
TextEndOfParagraph	The specialized text run used to mark the end of a paragraph.
TextEndOfSegment	The specialized text run used to mark the end of a segment, such as to end the scope affected by a previous <a href="#">TextModifier</a> run.
TextHidden	The specialized text run used to mark a range of hidden characters.
TextModifier	The specialized text run used to modify properties of text runs in its scope. The scope extends to the next matching <a href="#">TextEndOfSegment</a> text run, or the next <a href="#">TextEndOfParagraph</a> .

Any of the predefined [TextRun](#) objects can be subclassed. This allows your text source to provide the text formatter with text runs that include custom data.

The following example demonstrates a [GetTextRun](#) method. This text store returns [TextRun](#) objects to the text formatter for processing.

```
// Used by the TextFormatter object to retrieve a run of text from the text source.
public override TextRun GetTextRun(int textSourceCharacterIndex)
{
    // Make sure text source index is in bounds.
    if (textSourceCharacterIndex < 0)
        throw new ArgumentOutOfRangeException("textSourceCharacterIndex", "Value must be greater than 0.");
    if (textSourceCharacterIndex >= _text.Length)
    {
        return new TextEndOfParagraph(1);
    }

    // Create TextCharacters using the current font rendering properties.
    if (textSourceCharacterIndex < _text.Length)
    {
        return new TextCharacters(
            _text,
            textSourceCharacterIndex,
            _text.Length - textSourceCharacterIndex,
            new GenericTextRunProperties(_currentRendering));
    }

    // Return an end-of-paragraph if no more text source.
    return new TextEndOfParagraph(1);
}
```

```

' Used by the TextFormatter object to retrieve a run of text from the text source.
Public Overrides Function GetTextRun(ByVal textSourceCharacterIndex As Integer) As TextRun
    ' Make sure text source index is in bounds.
    If textSourceCharacterIndex < 0 Then
        Throw New ArgumentOutOfRangeException("textSourceCharacterIndex", "Value must be greater than 0.")
    End If
    If textSourceCharacterIndex >= _text.Length Then
        Return New TextEndOfParagraph(1)
    End If

    ' Create TextCharacters using the current font rendering properties.
    If textSourceCharacterIndex < _text.Length Then
        Return New TextCharacters(_text, textSourceCharacterIndex, _text.Length - textSourceCharacterIndex, New
GenericTextRunProperties(_currentRendering))
    End If

    ' Return an end-of-paragraph if no more text source.
    Return New TextEndOfParagraph(1)
End Function

```

#### **NOTE**

In this example, the text store provides the same text properties to all of the text. Advanced text stores would need to implement their own span management to allow individual characters to have different properties.

## Specifying Formatting Properties

[TextRun](#) objects are formatted by using properties provided by the text store. These properties come in two types, [TextParagraphProperties](#) and [TextRunProperties](#). [TextParagraphProperties](#) handle paragraph inclusive properties such as [TextAlignment](#) and [FlowDirection](#). [TextRunProperties](#) are properties that can be different for each text run within a paragraph, such as foreground brush, [Typeface](#), and font size. To implement custom paragraph and custom text run property types, your application must create classes that derive from [TextParagraphProperties](#) and [TextRunProperties](#) respectively.

## See also

- [Typography in WPF](#)
- [Documents in WPF](#)

# Fonts (WPF)

8/16/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) includes support for rich presentation of text using OpenType fonts. A sample pack of OpenType fonts is included with the Windows SDK.

## In This Section

[OpenType Font Features](#)

[Packaging Fonts with Applications](#)

[Sample OpenType Font Pack](#)

[How-to Topics](#)

## See also

- [FontStyle](#)
- [SystemFonts](#)
- [Documents in WPF](#)
- [Typography in WPF](#)

# OpenType Font Features

10/7/2019 • 11 minutes to read • [Edit Online](#)

This topic provides an overview of some of the key features of OpenType font technology in Windows Presentation Foundation (WPF).

## OpenType Font Format

The OpenType font format is an extension of the TrueType® font format, adding support for PostScript font data. The OpenType font format was developed jointly by Microsoft and Adobe Corporation. OpenType fonts and the operating system services which support OpenType fonts provide users with a simple way to install and use fonts, whether the fonts contain TrueType outlines or CFF (PostScript) outlines.

The OpenType font format addresses the following developer challenges:

- Broader multi-platform support.
- Better support for international character sets.
- Better protection for font data.
- Smaller file sizes to make font distribution more efficient.
- Broader support for advanced typographic control.

### NOTE

The Windows SDK contains a set of sample OpenType fonts that you can use with Windows Presentation Foundation (WPF) applications. These fonts provide most of the features illustrated in the rest of this topic. For more information, see [Sample OpenType Font Pack](#).

See the [OpenType Specification](#) for details of the OpenType font format.

### Advanced Typographic Extensions

The Advanced Typographic tables (OpenType Layout tables) extend the functionality of fonts with either TrueType or CFF outlines. OpenType Layout fonts contain additional information that extends the capabilities of the fonts to support high-quality international typography. Most OpenType fonts expose only a subset of the total OpenType features available. OpenType fonts provide the following features.

- Rich mapping between characters and glyphs that support ligatures, positional forms, alternates, and other font substitutions.
- Support for two-dimensional positioning and glyph attachment.
- Explicit script and language information contained in font, so a text-processing application can adjust its behavior accordingly.

The OpenType Layout tables are described in more detail in the "[Font File Tables](#)" section of the OpenType specification.

The remainder of this overview introduces the breadth and flexibility of some of the visually-interesting OpenType features that are exposed by the properties of the [Typography](#) object. For more information on this object, see [Typography Class](#).

# Variants

Variants are used to render different typographic styles, such as superscripts and subscripts.

## Superscripts and Subscripts

The [Variants](#) property allows you to set superscript and subscript values for an OpenType font.

The following text displays superscripts for the Palatino Linotype font.

2<sup>3</sup> 14<sup>th</sup>

The following markup example shows how to define superscripts for the Palatino Linotype font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Palatino Linotype">
  2<Run Typography.Variants="Superscript">3</Run>
  14<Run Typography.Variants="Superscript">th</Run>
</Paragraph>
```

The following text displays subscripts for the Palatino Linotype font.

H<sub>2</sub>O Footnote<sub>4</sub>

The following markup example shows how to define subscripts for the Palatino Linotype font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Palatino Linotype">
  H<Run Typography.Variants="Subscript">2</Run>O
  Footnote<Run Typography.Variants="Subscript">4</Run>
</Paragraph>
```

## Decorative Uses of Superscripts and Subscripts

You can also use superscripts and subscripts to create decorative effects of mixed case text. The following text displays superscript and subscript text for the Palatino Linotype font. Note that the capitals are not affected.

Chapter One

Chapter One

The following markup example shows how to define superscripts and subscripts for a font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Palatino Linotype" Typography.Variants="Superscript">
  Chapter One
</Paragraph>
<Paragraph FontFamily="Palatino Linotype" Typography.Variants="Subscript">
  Chapter One
</Paragraph>
```

# Capitals

Capitals are a set of typographical forms that render text in capital-styled glyphs. Typically, when text is rendered as all capitals, the spacing between letters can appear too tight, and the weight and proportion of the letters too

heavy. OpenType supports a number of styling formats for capitals, including small capitals, petite capitals, titling, and capital spacing. These styling formats allow you to control the appearance of capitals.

The following text displays standard capital letters for the Pescadero font, followed by the letters styled as "SmallCaps" and "AllSmallCaps". In this case, the same font size is used for all three words.

# CAPITALS CAPITALS CAPITALS

The following markup example shows how to define capitals for the Pescadero font, using properties of the [Typography](#) object. When the "SmallCaps" format is used, any leading capital letter is ignored.

```
<Paragraph FontFamily="Pescadero" FontSize="48">
  <Run>CAPITALS</Run>
  <Run Typography.Capitals="SmallCaps">Capitals</Run>
  <Run Typography.Capitals="AllSmallCaps">Capitals</Run>
</Paragraph>
```

## Titling Capitals

Titling capitals are lighter in weight and proportion and designed to give a more elegant look than normal capitals. Titling capitals are typically used in larger font sizes as headings. The following text displays normal and titling capitals for the Pescadero font. Notice the narrower stem widths of the text on the second line.

# CHAPTER ONE CHAPTER ONE

The following markup example shows how to define titling capitals for the Pescadero font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Pescadero">
  <Run Typography.Capitals="Titling">chapter one</Run>
</Paragraph>
```

## Capital Spacing

Capital spacing is a feature that allows you to provide more spacing when using all capitals in text. Capital letters are typically designed to blend with lowercase letters. Spacing that appears attractive between and a capital letter and a lowercase letter may look too tight when all capital letters are used. The following text displays normal and capital spacing for the Pescadero font.

# CHAPTER ONE CHAPTER ONE

The following markup example shows how to define capital spacing for the Pescadero font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Pescadero">
  <Run Typography.CapitalSpacing="True">CHAPTER ONE</Run>
</Paragraph>
```

# Ligatures

Ligatures are two or more glyphs that are formed into a single glyph in order to create more readable or attractive text. OpenType fonts support four types of ligatures:

- **Standard ligatures.** Designed to enhance readability. Standard ligatures include "fi", "ff", and "fl".
- **Contextual ligatures.** Designed to enhance readability by providing better joining behavior between the characters that make up the ligature.
- **Discretionary ligatures.** Designed to be ornamental, and not specifically designed for readability.
- **Historical ligatures.** Designed to be historical, and not specifically designed for readability.

The following text displays standard ligature glyphs for the Pericles font.

FI FL TH TT TV TW TY VT WT YT

The following markup example shows how to define standard ligature glyphs for the Pericles font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Pericles" Typography.StandardLigatures="True">
  <Run Typography.StylisticAlternates="1">FI</Run>
  <Run Typography.StylisticAlternates="1">FL</Run>
  <Run Typography.StylisticAlternates="1">TH</Run>
  <Run Typography.StylisticAlternates="1">TT</Run>
  <Run Typography.StylisticAlternates="1">TV</Run>
  <Run Typography.StylisticAlternates="1">TW</Run>
  <Run Typography.StylisticAlternates="1">TY</Run>
  <Run Typography.StylisticAlternates="1">VT</Run>
  <Run Typography.StylisticAlternates="1">WT</Run>
  <Run Typography.StylisticAlternates="1">YT</Run>
</Paragraph>
```

The following text displays discretionary ligature glyphs for the Pericles font.

CO LA LE LI LL LO LU

The following markup example shows how to define discretionary ligature glyphs for the Pericles font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Pericles" Typography.DiscretionaryLigatures="True">
  <Run Typography.StylisticAlternates="1">CO</Run>
  <Run Typography.StylisticAlternates="1">LA</Run>
  <Run Typography.StylisticAlternates="1">LE</Run>
  <Run Typography.StylisticAlternates="1">LI</Run>
  <Run Typography.StylisticAlternates="1">LL</Run>
  <Run Typography.StylisticAlternates="1">LO</Run>
  <Run Typography.StylisticAlternates="1">LU</Run>
</Paragraph>
```

By default, OpenType fonts in Windows Presentation Foundation (WPF) enable standard ligatures. For example, if you use the Palatino Linotype font, the standard ligatures "fi", "ff", and "fl" appear as a combined character glyph. Notice that the pair of characters for each standard ligature touch each other.

fi ff fl

However, you can disable standard ligature features so that a standard ligature such as "ff" displays as two separate glyphs, rather than as a combined character glyph.

fi ff fl

The following markup example shows how to disable standard ligature glyphs for the Palatino Linotype font, using properties of the [Typography](#) object.

```
<!-- Set standard ligatures to false in order to disable feature. -->
<Paragraph Typography.StandardLigatures="False" FontFamily="Palatino Linotype" FontSize="72">
    fi ff fl
</Paragraph>
```

## Swashes

Swashes are decorative glyphs that use elaborate ornamentation often associated with calligraphy. The following text displays standard and swash glyphs for the Pescadero font.

A B C D E F G H I J K L M N  
A<sup>B</sup> C<sup>D</sup>E<sup>F</sup> G<sup>H</sup>I<sup>J</sup>K<sup>L</sup>M<sup>N</sup>

Swashes are often used as decorative elements in short phrases such as event announcements. The following text uses swashes to emphasize the capital letters of the name of the event.

Wishing you a  
Happy New Year!

The following markup example shows how to define swashes for a font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Pescadero" TextBlock.TextAlignment="Center">
    Wishing you a<LineBreak/>
    <Run Typography.StandardSwashes="1" FontSize="36">Happy New Year!</Run>
</Paragraph>
```

## Contextual Swashes

Certain combinations of swash glyphs can cause an unattractive appearance, such as overlapping descenders on adjacent letters. Using a contextual swash allows you to use a substitute swash glyph that produces a better appearance. The following text shows the same word before and after a contextual swash is applied.

Lyon Lyon

The following markup example shows how to define a contextual swash for the Pescadero font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Pescadero" Typography.StandardSwashes="1">
    Lyon <Run Typography.ContextualSwashes="1">L</Run>yon
</Paragraph>
```

## Alternates

Alternates are glyphs that can be substituted for a standard glyph. OpenType fonts, such as the Pericles font used in the following examples, can contain alternate glyphs that you can use to create different appearances for text. The following text displays standard glyphs for the Pericles font.

# ANCIENT GREEK MYTHOLOGY

The Pericles OpenType font contains additional glyphs that provide stylistic alternates to the standard set of glyphs. The following text displays stylistic alternate glyphs.

# ANCIENT GREEK MYTHOLOGY

The following markup example shows how to define stylistic alternate glyphs for the Pericles font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Pericles">
  <Run Typography.StylisticAlternates="1">A</Run>NCIENT
  GR<Run Typography.StylisticAlternates="1">EE</Run>K
  MYTH<Run Typography.StylisticAlternates="1">O</Run>LOGY
</Paragraph>
```

The following text displays several other stylistic alternate glyphs for the Pericles font.

A A Ä Ç E G Ø Q R R S Y

The following markup example shows how to define these other stylistic alternate glyphs.

```
<Paragraph FontFamily="Pericles">
  <Run Typography.StylisticAlternates="1">A</Run>
  <Run Typography.StylisticAlternates="2">A</Run>
  <Run Typography.StylisticAlternates="3">A</Run>
  <Run Typography.StylisticAlternates="1">C</Run>
  <Run Typography.StylisticAlternates="1">E</Run>
  <Run Typography.StylisticAlternates="1">G</Run>
  <Run Typography.StylisticAlternates="1">O</Run>
  <Run Typography.StylisticAlternates="1">Q</Run>
  <Run Typography.StylisticAlternates="1">R</Run>
  <Run Typography.StylisticAlternates="2">R</Run>
  <Run Typography.StylisticAlternates="1">S</Run>
  <Run Typography.StylisticAlternates="1">Y</Run>
</Paragraph>
```

### Random Contextual Alternates

Random contextual alternates provide multiple substitute glyphs for a single character. When implemented with script-type fonts, this feature can simulate handwriting by using of a set of randomly chosen glyphs with slight differences in appearance. The following text uses random contextual alternates for the Lindsey font. Notice that the letter "a" varies slightly in appearance

a banana in a cabana

The following markup example shows how to define random contextual alternates for the Lindsey font, using properties of the [Typography](#) object.

```
<TextBlock FontFamily="Lindsey">
  <Run Typography.ContextualAlternates="True">
    a banana in a cabana
  </Run>
</TextBlock>
```

## Historical Forms

Historical forms are typographic conventions that were common in the past. The following text displays the phrase, "Boston, Massachusetts", using an historical form of glyphs for the Palatino Linotype font.

Bofton, Maffachufettf

The following markup example shows how to define historical forms for the Palatino Linotype font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Palatino Linotype">
  <Run Typography.HistoricalForms="True">Boston, Massachusetts</Run>
</Paragraph>
```

## Numerical Styles

OpenType fonts support a large number of features that can be used with numerical values in text.

### Fractions

OpenType fonts support styles for fractions, including slashed and stacked.

The following text displays fraction styles for the Palatino Linotype font.

1/8 1/4 3/8 1/2 5/8 3/4 7/8

$\frac{1}{8}$   $\frac{1}{4}$   $\frac{3}{8}$   $\frac{1}{2}$   $\frac{5}{8}$   $\frac{3}{4}$   $\frac{7}{8}$

The following markup example shows how to define fraction styles for the Palatino Linotype font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Palatino Linotype" Typography.Fraction="Slashed">
  1/8 1/4 3/8 1/2 5/8 3/4 7/8
</Paragraph>
<Paragraph FontFamily="Palatino Linotype" Typography.Fraction="Stacked">
  1/8 1/4 3/8 1/2 5/8 3/4 7/8
</Paragraph>
```

### Old Style Numerals

OpenType fonts support an old style numeral format. This format is useful for displaying numerals in styles that are no longer standard. The following text displays an 18th century date in standard and old style numeral formats for the Palatino Linotype font.

July 4, 1776      July 4, 1776

The following text displays standard numerals for the Palatino Linotype font, followed by old style numerals.

1234567890 1234567890

The following markup example shows how to define old style numerals for the Palatino Linotype font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Palatino Linotype">
  <Run Typography.NumeralStyle="Normal">1234567890</Run>
  <Run Typography.NumeralStyle="OldStyle">1234567890</Run>
</Paragraph>
```

## Proportional and Tabular Figures

OpenType fonts support a proportional and tabular figure feature to control the alignment of widths when using numerals. Proportional figures treat each numeral as having a different width—"1" is narrower than "5". Tabular figures are treated as equal-width numerals so that they align vertically, which increases the readability of financial type information.

The following text displays two proportional figures in the first column using the Miramonte font. Note the difference in width between the numerals "5" and "1". The second column shows the same two numeric values with the widths adjusted by using the tabular figure feature.

The image shows two columns of numbers on lined paper. The first column contains the numbers '550,689' and '114,131'. The second column contains the same numbers, but the '5' is wider than the '1', demonstrating proportional figure widths. This is followed by a second row of the same numbers, where the '5' and '1' have the same width, demonstrating tabular figure widths.

550,689	550,689
114,131	114,131

The following markup example shows how to define proportional and tabular figures for the Miramonte font, using properties of the [Typography](#) object.

```
<TextBlock FontFamily="Miramonte">
  <Run Typography.NumeralAlignment="Proportional">114,131</Run>
</TextBlock>
<TextBlock FontFamily="Miramonte">
  <Run Typography.NumeralAlignment="Tabular">114,131</Run>
</TextBlock>
```

## Slashed Zero

OpenType fonts support a slashed zero numeral format to emphasize the difference between the letter "O" and the numeral "0". The slashed zero numeral is often used for identifiers in financial and business information.

The following text displays a sample order identifier using the Miramonte font. The first line uses standard numerals. The second line used slashed zero numerals to provide better contrast with the uppercase "O" letter.

Order #0048-OTC-390  
Order #0048-OTC-390

The following markup example shows how to define slashed zero numerals for the Miramonte font, using properties of the [Typography](#) object.

```
<Paragraph FontFamily="Miramonte">
  <Run>Order #0048-OTC-390</Run>
  <LineBreak/>
  <Run Typography.SlashedZero="True">Order #0048-OTC-390</Run>
</Paragraph>
```

# Typography Class

The [Typography](#) object exposes the set of features that an OpenType font supports. By setting the properties of [Typography](#) in markup, you can easily author documents that take advantage of OpenType features.

The following text displays standard capital letters for the Pescadero font, followed by the letters styled as "SmallCaps" and "AllSmallCaps". In this case, the same font size is used for all three words.

# CAPITALS CAPITALS CAPITALS

The following markup example shows how to define capitals for the Pescadero font, using properties of the [Typography](#) object. When the "SmallCaps" format is used, any leading capital letter is ignored.

```
<Paragraph FontFamily="Pescadero" FontSize="48">
  <Run>CAPITALS</Run>
  <Run Typography.Capitals="SmallCaps">Capitals</Run>
  <Run Typography.Capitals="AllSmallCaps">Capitals</Run>
</Paragraph>
```

The following code example accomplishes the same task as the previous markup example.

```
MyParagraph.FontFamily = new FontFamily("Pescadero");
MyParagraph.FontSize = 48;

Run run_1 = new Run("CAPITALS ");
MyParagraph.Inlines.Add(run_1);

Run run_2 = new Run("Capitals ");
run_2.Typography.Capitals = FontCapitals.SmallCaps;
MyParagraph.Inlines.Add(run_2);

Run run_3 = new Run("Capitals");
run_3.Typography.Capitals = FontCapitals.AllSmallCaps;
MyParagraph.Inlines.Add(run_3);

MyParagraph.Inlines.Add(new LineBreak());
```

```
MyParagraph.FontFamily = New FontFamily("Pescadero")
MyParagraph.FontSize = 48

Dim run_1 As New Run("CAPITALS ")
MyParagraph.Inlines.Add(run_1)

Dim run_2 As New Run("Capitals ")
run_2.Typography.Capitals = FontCapitals.SmallCaps
MyParagraph.Inlines.Add(run_2)

Dim run_3 As New Run("Capitals")
run_3.Typography.Capitals = FontCapitals.AllSmallCaps
MyParagraph.Inlines.Add(run_3)

MyParagraph.Inlines.Add(New LineBreak())
```

## Typography Class Properties

The following table lists the properties, values, and default settings of the [Typography](#) object.

PROPERTY	VALUE(S)	DEFAULT VALUE
<a href="#">AnnotationAlternates</a>	Numeric value - byte	0

PROPERTY	VALUE(S)	DEFAULT VALUE
Capitals	AllPetiteCaps   AllSmallCaps   Normal   PetiteCaps   SmallCaps   Titling   Unicase	FontCapitals.Normal
CapitalSpacing	Boolean	false
CaseSensitiveForms	Boolean	false
ContextualAlternates	Boolean	true
ContextualLigatures	Boolean	true
ContextualSwashes	Numeric value - byte	0
DiscretionaryLigatures	Boolean	false
EastAsianExpertForms	Boolean	false
EastAsianLanguage	HojoKanji   Jis04   Jis78   Jis83   Jis90   NlcKanji   Normal   Simplified   Traditional   TraditionalNames	FontEastAsianLanguage.Normal
EastAsianWidths	Full   Half   Normal   Proportional   Quarter   Third	FontEastAsianWidths.Normal
Fraction	Normal   Slashed   Stacked	FontFraction.Normal
HistoricalForms	Boolean	false
HistoricalLigatures	Boolean	false
Kerning	Boolean	true
MathematicalGreek	Boolean	false
NumeralAlignment	Normal   Proportional   Tabular	FontNumeralAlignment.Normal
NumeralStyle	Boolean	FontNumeralStyle.Normal
SlashedZero	Boolean	false
StandardLigatures	Boolean	true
StandardSwashes	numeric value – byte	0
StylisticAlternates	numeric value – byte	0
StylisticSet1	Boolean	false
StylisticSet2	Boolean	false

PROPERTY	VALUE(S)	DEFAULT VALUE
StylisticSet3	Boolean	false
StylisticSet4	Boolean	false
StylisticSet5	Boolean	false
StylisticSet6	Boolean	false
StylisticSet7	Boolean	false
StylisticSet8	Boolean	false
StylisticSet9	Boolean	false
StylisticSet10	Boolean	false
StylisticSet11	Boolean	false
StylisticSet12	Boolean	false
StylisticSet13	Boolean	false
StylisticSet14	Boolean	false
StylisticSet15	Boolean	false
StylisticSet16	Boolean	false
StylisticSet17	Boolean	false
StylisticSet18	Boolean	false
StylisticSet19	Boolean	false
StylisticSet20	Boolean	false
Variants	Inferior   Normal   Ordinal   Ruby   Subscript   Superscript	FontVariants.Normal

## See also

- [Typography](#)
- [OpenType Specification](#)
- [Typography in WPF](#)
- [Sample OpenType Font Pack](#)
- [Packaging Fonts with Applications](#)

# Packaging Fonts with Applications

10/18/2019 • 7 minutes to read • [Edit Online](#)

This topic provides an overview of how to package fonts with your Windows Presentation Foundation (WPF) application.

## NOTE

As with most types of software, font files are licensed, rather than sold. Licenses that govern the use of fonts vary from vendor to vendor but in general most licenses, including those covering the fonts Microsoft supplies with applications and Windows, do not allow the fonts to be embedded within applications or otherwise redistributed. Therefore, as a developer it is your responsibility to ensure that you have the required license rights for any font you embed within an application or otherwise redistribute.

## Introduction to Packaging Fonts

You can easily package fonts as resources within your WPF applications to display user interface text and other types of text based content. The fonts can be separate from or embedded within the application's assembly files. You can also create a resource-only font library, which your application can reference.

OpenType and TrueType® fonts contain a type flag, `fsType`, that indicates font embedding licensing rights for the font. However, this type flag only refers to embedded fonts stored in a document—it does not refer to fonts embedded in an application. You can retrieve the font embedding rights for a font by creating a [GlyphTypeface](#) object and referencing its [EmbeddingRights](#) property. Refer to the "OS/2 and Windows Metrics" section of the [OpenType Specification](#) for more information on the `fsType` flag.

The [Microsoft Typography](#) Web site includes contact information that can help you locate a particular font vendor or find a font vendor for custom work.

## Adding Fonts as Content Items

You can add fonts to your application as project content items that are separate from the application's assembly files. This means that content items are not embedded as resources within an assembly. The following project file example shows how to define content items.

```
<Project DefaultTargets="Build"
      xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Other project build settings ... -->

  <ItemGroup>
    <Content Include="Peric.ttf" />
    <Content Include="Peric1.ttf" />
  </ItemGroup>
</Project>
```

In order to ensure that the application can use the fonts at run time, the fonts must be accessible in the application's deployment directory. The `<CopyToOutputDirectory>` element in the application's project file allows you to automatically copy the fonts to the application deployment directory during the build process. The following project file example shows how to copy fonts to the deployment directory.

```
<ItemGroup>
  <Content Include="Peric.ttf">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </Content>
  <Content Include="Pericl.ttf">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </Content>
</ItemGroup>
```

The following code example shows how to reference the application's font as a content item—the referenced content item must be in the same directory as the application's assembly files.

```
<TextBlock FontFamily="./#Pericles Light">
  Aegean Sea
</TextBlock>
```

## Adding Fonts as Resource Items

You can add fonts to your application as project resource items that are embedded within the application's assembly files. Using a separate subdirectory for resources helps to organize the application's project files. The following project file example shows how to define fonts as resource items in a separate subdirectory.

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Other project build settings ... -->

  <ItemGroup>
    <Resource Include="resources\Peric.ttf" />
    <Resource Include="resources\Pericl.ttf" />
  </ItemGroup>
</Project>
```

### NOTE

When you add fonts as resources to your application, make sure you are setting the `<Resource>` element, and not the `<EmbeddedResource>` element in your application's project file. The `<EmbeddedResource>` element for the build action is not supported.

The following markup example shows how to reference the application's font resources.

```
<TextBlock FontFamily=".(resources/#Pericles Light)">
  Aegean Sea
</TextBlock>
```

## Referencing Font Resource Items from Code

In order to reference font resource items from code, you must supply a two-part font resource reference: the base uniform resource identifier (URI); and the font location reference. These values are used as the parameters for the `FontFamily` method. The following code example shows how to reference the application's font resources in the project subdirectory called `resources`.

```
// The font resource reference includes the base URI reference (application directory level),
// and a relative URI reference.
myTextBlock.FontFamily = new FontFamily(new Uri("pack://application:,,,/"),
  ".(resources/#Pericles Light");
```

```
' The font resource reference includes the base URI reference (application directory level),
' and a relative URI reference.
myTextBlock.FontFamily = New FontFamily(New Uri("pack://application:,,,/"), "./resources/#Pericles Light")
```

The base uniform resource identifier (URI) can include the application subdirectory where the font resource resides. In this case, the font location reference would not need to specify a directory, but would have to include a leading "./", which indicates the font resource is in the same directory specified by the base uniform resource identifier (URI). The following code example shows an alternate way of referencing the font resource item—it is equivalent to the previous code example.

```
// The base URI reference can include an application subdirectory.
myTextBlock.FontFamily = new FontFamily(new Uri("pack://application:,,,/resources/"), "./#Pericles Light");
```

```
' The base URI reference can include an application subdirectory.
myTextBlock.FontFamily = New FontFamily(New Uri("pack://application:,,,/resources/"), "./#Pericles Light")
```

## Referencing Fonts from the Same Application Subdirectory

You can place both application content and resource files within the same user-defined subdirectory of your application project. The following project file example shows a content page and font resources defined in the same subdirectory.

```
<ItemGroup>
  <Page Include="pages\HomePage.xaml" />
</ItemGroup>
<ItemGroup>
  <Resource Include="pages\Peric.ttf" />
  <Resource Include="pages\Pericl.ttf" />
</ItemGroup>
```

Since the application content and font are in the same subdirectory, the font reference is relative to the application content. The following examples show how to reference the application's font resource when the font is in the same directory as the application.

```
<TextBlock FontFamily="./#Pericles Light">
  Aegean Sea
</TextBlock>
```

```
// The font resource reference includes the base Uri (application directory level),
// and the file resource location, which is relative to the base Uri.
myTextBlock.FontFamily = new FontFamily(new Uri("pack://application:,,,/"), "/pages/#Pericles Light");
```

```
' The font resource reference includes the base Uri (application directory level),
' and the file resource location, which is relative to the base Uri.
myTextBlock.FontFamily = New FontFamily(New Uri("pack://application:,,,/"), "/pages/#Pericles Light")
```

## Enumerating Fonts in an Application

To enumerate fonts as resource items in your application, use either the [GetFontFamilies](#) or [GetTypefaces](#) method. The following example shows how to use the [GetFontFamilies](#) method to return the collection of [FontFamily](#) objects from the application font location. In this case, the application contains a subdirectory named "resources".

```
foreach (FontFamily fontFamily in Fonts.GetFontFamilies(new Uri("pack://application:,,,/"), "./resources/"))
{
    // Perform action.
}
```

```
For Each fontFamily As FontFamily In Fonts.GetFontFamilies(New Uri("pack://application:,,,/"), "./resources/")
    ' Perform action.
Next fontFamily
```

The following example shows how to use the [GetTypefaces](#) method to return the collection of [Typeface](#) objects from the application font location. In this case, the application contains a subdirectory named "resources".

```
foreach (Typeface typeface in Fonts.GetTypefaces(new Uri("pack://application:,,,/"), "./resources/"))
{
    // Perform action.
}
```

```
For Each typeface As Typeface In Fonts.GetTypefaces(New Uri("pack://application:,,,/"), "./resources/")
    ' Perform action.
Next typeface
```

## Creating a Font Resource Library

You can create a resource-only library that contains only fonts—no code is part of this type of library project. Creating a resource-only library is a common technique for decoupling resources from the application code that uses them. This also allows the library assembly to be included with multiple application projects. The following project file example shows the key portions of a resource-only library project.

```
<PropertyGroup>
    <AssemblyName>FontLibrary</AssemblyName>
    <OutputType>library</OutputType>
    ...
</PropertyGroup>
...
<ItemGroup>
    <Resource Include="Kooten.ttf" />
    <Resource Include="Pesca.ttf" />
</ItemGroup>
```

### Referencing a Font in a Resource Library

To reference a font in a resource library from your application, you must prefix the font reference with the name of the library assembly. In this case, the font resource assembly is "FontLibrary". To separate the assembly name from the reference within the assembly, use a ';' character. Adding the "Component" keyword followed by the reference to the font name completes the full reference to the font library's resource. The following code example shows how to reference a font in a resource library assembly.

```
<Run FontFamily="/FontLibrary;Component/#Kootenay" FontSize="36">
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
</Run>
```

#### NOTE

This SDK contains a set of sample OpenType fonts that you can use with WPF applications. The fonts are defined in a resource-only library. For more information, see [Sample OpenType Font Pack](#).

## Limitations on Font Usage

The following list describes several limitations on the packaging and use of fonts in WPF applications:

- **Font embedding permission bits:** WPF applications do not check or enforce any font embedding permission bits. See the [Introduction\\_to\\_Packing\\_Fonts](#) section for more information.
- **Site of origin fonts:** WPF applications do not allow a font reference to an http or ftp uniform resource identifier (URI).
- **Absolute URI using the pack: notation:** WPF applications do not allow you to create a [FontFamily](#) object programmatically using "pack:" as part of the absolute uniform resource identifier (URI) reference to a font. For example, `"pack://application:,,,/resources/#Pericles_Light"` is an invalid font reference.
- **Automatic font embedding:** During design time, there is no support for searching an application's use of fonts and automatically embedding the fonts in the application's resources.
- **Font subsets:** WPF applications do not support the creation of font subsets for non-fixed documents.
- In cases where there is an incorrect reference, the application falls back to using an available font.

## See also

- [Typography](#)
- [FontFamily](#)
- [Microsoft Typography: Links, News, and Contacts](#)
- [OpenType Specification](#)
- [OpenType Font Features](#)
- [Sample OpenType Font Pack](#)

# Sample OpenType Font Pack

11/7/2019 • 2 minutes to read • [Edit Online](#)

This topic provides an overview of the sample OpenType fonts that are distributed with the Windows SDK. The sample fonts support extended OpenType features that can be used by Windows Presentation Foundation (WPF) applications.

## Fonts in the OpenType Font Pack

The Windows SDK provides a set of sample OpenType fonts that you can use in creating Windows Presentation Foundation (WPF) applications. The sample fonts are supplied under license from Ascender Corporation. These fonts implement only a subset of the total features defined by the OpenType format. The following table lists the names of the sample OpenType fonts.

NAME	FILE
Kootenay	Kooten.ttf
Lindsey	Linds.ttf
Miramonte	Miramo.ttf
Miramonte Bold	Miramob.ttf
Pericles	Peric.ttf
Pericles Light	Pericl.ttf
Pescadero	Pesca.ttf
Pescadero Bold	Pescab.ttf

The following illustration shows what the sample OpenType fonts look like.

Kootenay

Lindsey

Miramonte

**Miramonte Bold**

PERICLES

PERICLES LIGHT

Pescadero

**Pescadero Bold**

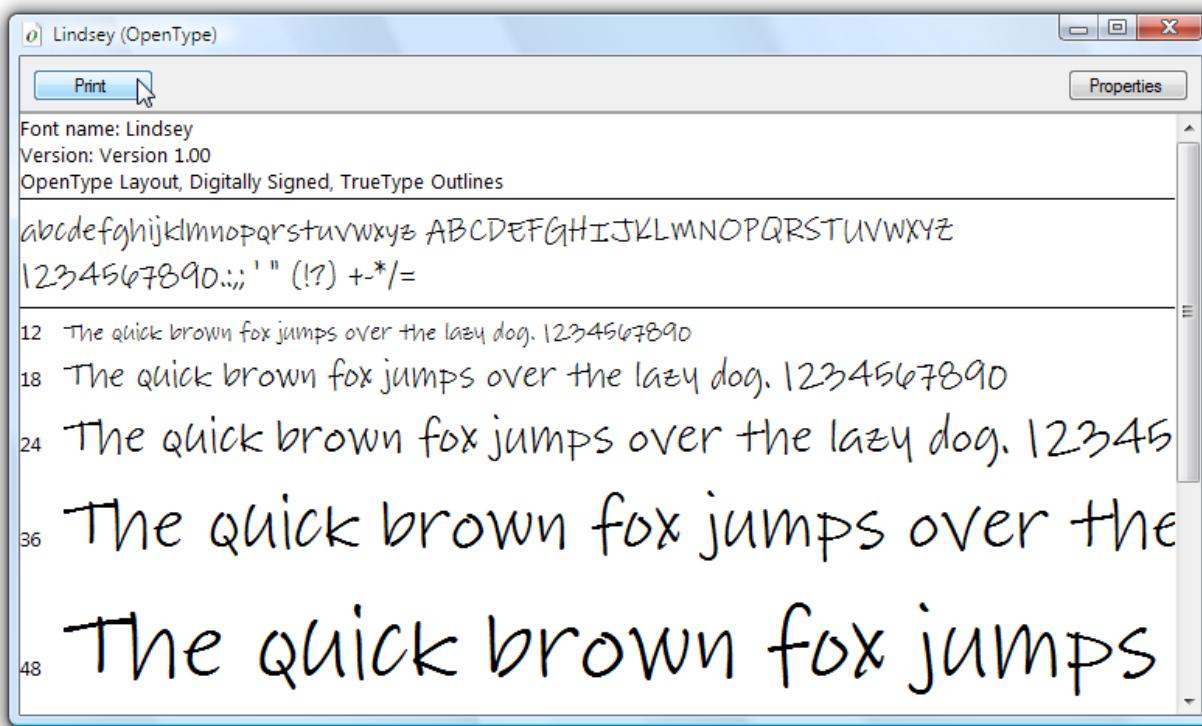
The sample fonts are supplied under license from Ascender Corporation. Ascender is a provider of advanced font products. To license extended or custom versions of the sample fonts, see [Ascender Corporation's Web site](#).

#### NOTE

As a developer it is your responsibility to ensure that you have the required license rights for any font you embed within an application or otherwise redistribute.

## Installing the Fonts

You have the option of installing the sample OpenType fonts to the default Windows Fonts directory, **\WINDOWS\FONTS**. Use the Fonts control panel to install the fonts. Once these fonts are on your computer, they are accessible to all applications that reference default Windows fonts. You can display a representative set of characters in several font sizes by doubling-clicking the font file. The following screenshot shows the Lindsey font file, Linds.ttf.



Displaying the Lindsey font

## Using the Fonts

There are two ways that you can use fonts in your application. You can add fonts to your application as project content items that are not embedded as resources within an assembly. Alternatively, you can add fonts to your application as project resource items that are embedded within the application's assembly files. For more information, see [Packaging Fonts with Applications](#).

## See also

- [Typography](#)
- [OpenType Font Features](#)
- [Packaging Fonts with Applications](#)

# Fonts How-to Topics

4/8/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section demonstrate how to use the font features included with Windows Presentation Foundation (WPF).

## In This Section

[Enumerate System Fonts](#)

[Use the FontSizeConverter Class](#)

## See also

- [FontStyle](#)
- [SystemFonts](#)
- [Documents in WPF](#)
- [Typography in WPF](#)

# How to: Enumerate System Fonts

3/5/2019 • 2 minutes to read • [Edit Online](#)

## Example

The following example shows how to enumerate the fonts in the system font collection. The font family name of each `FontFamily` within `SystemFontFamilies` is added as an item to a combo box.

```
public void FillFontComboBox(ComboBox comboBoxFonts)
{
    // Enumerate the current set of system fonts,
    // and fill the combo box with the names of the fonts.
    foreach (FontFamily fontFamily in Fonts.SystemFontFamilies)
    {
        // FontFamily.Source contains the font family name.
        comboBoxFonts.Items.Add(fontFamily.Source);
    }

    comboBoxFonts.SelectedIndex = 0;
}
```

```
Public Sub FillFontComboBox(ByVal comboBoxFonts As ComboBox)
    ' Enumerate the current set of system fonts,
    ' and fill the combo box with the names of the fonts.
    For Each fontFamily As FontFamily In Fonts.SystemFontFamilies
        ' FontFamily.Source contains the font family name.
        comboBoxFonts.Items.Add(fontFamily.Source)
    Next fontFamily

    comboBoxFonts.SelectedIndex = 0
End Sub
```

If multiple versions of the same font family reside in the same directory, the Windows Presentation Foundation (WPF) font enumeration returns the most recent version of the font. If the version information does not provide resolution, the font with latest timestamp is returned. If the timestamp information is equivalent, the font file that is first in alphabetical order is returned.

# How to: Use the FontSizeConverter Class

4/8/2019 • 2 minutes to read • [Edit Online](#)

## Example

This example shows how to create an instance of [FontSizeConverter](#) and use it to change a font size.

The example defines a custom method called `changeSize` that converts the contents of a [ListBoxItem](#), as defined in a separate Extensible Application Markup Language (XAML) file, to an instance of [Double](#), and later into a [String](#). This method passes the [ListBoxItem](#) to a [FontSizeConverter](#) object, which converts the [Content](#) of a [ListBoxItem](#) to an instance of [Double](#). This value is then passed back as the value of the [FontSize](#) property of the [TextBlock](#) element.

This example also defines a second custom method that is called `changeFamily`. This method converts the [Content](#) of the [ListBoxItem](#) to a [String](#), and then passes that value to the [FontFamily](#) property of the [TextBlock](#) element.

This example does not run.

```
private void changeSize(object sender, SelectionChangedEventArgs args)
{
    ListBoxItem li = ((sender as ListBox).SelectedItem as ListBoxItem);
    FontSizeConverter myFontSizeConverter = new FontSizeConverter();
    text1.FontSize = (Double)myFontSizeConverter.ConvertFromString(li.Content.ToString());
}

private void changeFamily(object sender, SelectionChangedEventArgs args)
{
    ListBoxItem li2 = ((sender as ListBox).SelectedItem as ListBoxItem);
    text1.FontFamily = new System.Windows.Media.FontFamily(li2.Content.ToString());
}
```

## See also

- [FontSizeConverter](#)

# Glyphs

4/8/2019 • 2 minutes to read • [Edit Online](#)

Glyphs are a low-level depiction of a character to be drawn on-screen. Windows Presentation Foundation (WPF) provides direct access to glyphs for customers who want to intercept and persist text after formatting.

## In This Section

[Introduction to the GlyphRun Object and Glyphs Element](#)

[How to: Draw Text Using Glyphs](#)

## See also

- [GlyphRun](#)
- [DrawText](#)
- [Glyphs](#)
- [Documents in WPF](#)
- [Typography in WPF](#)

# Introduction to the GlyphRun Object and Glyphs Element

10/18/2019 • 3 minutes to read • [Edit Online](#)

This topic describes the [GlyphRun](#) object and the [Glyphs](#) element.

## Introduction to GlyphRun

Windows Presentation Foundation (WPF) provides advanced text support including glyph-level markup with direct access to [Glyphs](#) for customers who want to intercept and persist text after formatting. These features provide critical support for the different text rendering requirements in each of the following scenarios.

1. Screen display of fixed-format documents.
2. Print scenarios.
  - Extensible Application Markup Language (XAML) as a device printer language.
  - Microsoft XPS Document Writer.
  - Previous printer drivers, output from Win32 applications to the fixed format.
  - Print spool format.
3. Fixed-format document representation, including clients for previous versions of Windows and other computing devices.

### NOTE

[Glyphs](#) and [GlyphRun](#) are designed for fixed-format document presentation and print scenarios. Windows Presentation Foundation (WPF) provides several elements for general layout and user interface (UI) scenarios such as [Label](#) and [TextBlock](#). For more information on layout and UI scenarios, see the [Typography in WPF](#).

## The GlyphRun Object

The [GlyphRun](#) object represents a sequence of glyphs from a single face of a single font at a single size, and with a single rendering style.

[GlyphRun](#) includes both font details such as glyph [Indices](#) and individual glyph positions. It also includes the original Unicode code points the run was generated from, character-to-glyph buffer offset mapping information, and per-character and per-glyph flags.

[GlyphRun](#) has a corresponding high-level [FrameworkElement](#), [Glyphs](#). [Glyphs](#) can be used in the element tree and in XAML markup to represent [GlyphRun](#) output.

## The Glyphs Element

The [Glyphs](#) element represents the output of a [GlyphRun](#) in XAML. The following markup syntax is used to describe the [Glyphs](#) element.

```

<!-- The example shows how to use a Glyphs object. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <StackPanel Background="PowderBlue">

        <Glyphs
            FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
            FontRenderingEmSize = "100"
            StyleSimulations   = "BoldSimulation"
            UnicodeString      = "Hello World!"
            Fill               = "Black"
            OriginX           = "100"
            OriginY           = "200"
        />

    </StackPanel>
</Page>

```

The following property definitions correspond to the first four attributes in the sample markup.

PROPERTY	DESCRIPTION
FontUri	Specifies a resource identifier: file name, Web uniform resource identifier (URI), or resource reference in the application .exe or container.
FontRenderingEmSize	Specifies the font size in drawing surface units (default is .96 inches).
StyleSimulations	Specifies flags for bold and Italic styles.
BidiLevel	Specifies the bidirectional layout level. Even-numbered and zero values imply left-to-right layout; odd-numbered values imply right-to-left layout.

## Indices property

The [Indices](#) property is a string of glyph specifications. Where a sequence of glyphs forms a single cluster, the specification of the first glyph in the cluster is preceded by a specification of how many glyphs and how many code points combine to form the cluster. The [Indices](#) property collects in one string the following properties.

- Glyph indices
- Glyph advance widths
- Combining glyph attachment vectors
- Cluster mapping from code points to glyphs
- Glyph flags

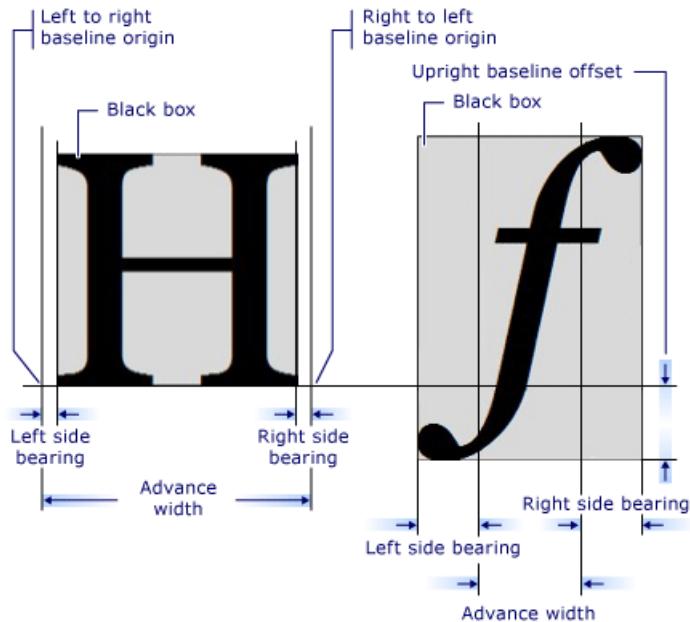
Each glyph specification has the following form.

[[GlyphIndex](#)] [, [Advance](#)] [, [uOffset](#)] [, [vOffset](#)] [, [Flags](#)]]]

## Glyph Metrics

Each glyph defines metrics that specify how it aligns with other [Glyphs](#). The following graphic defines the various

typographic qualities of two different glyph characters.



## Glyphs Markup

The following code example shows how to use various properties of the [Glyphs](#) element in XAML.

```
<!-- The example shows how to use different property settings of Glyphs objects. -->
<Canvas
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Background="PowderBlue"
>

<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
    FontRenderingEmSize = "36"
    StyleSimulations = "ItalicSimulation"
    UnicodeString    = "Hello World!"
    Fill             = "SteelBlue"
    OriginX          = "50"
    OriginY          = "75"
/>

<!-- "Hello World!" with default kerning -->
<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
    FontRenderingEmSize = "36"
    UnicodeString    = "Hello World!"
    Fill             = "Maroon"
    OriginX          = "50"
    OriginY          = "150"
/>

<!-- "Hello World!" with explicit character widths for proportional font -->
<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
    FontRenderingEmSize = "36"
    UnicodeString    = "Hello World!"
    Indices          = ",80;,80;,80;,80;,80;,80;,80;,80;,80;,80;,80"
    Fill             = "Maroon"
    OriginX          = "50"
    OriginY          = "225"
/>

<!-- "Hello World!" with fixed-width font -->
<Glyphs
```

```

<Typings>
  FontUri          = "C:\WINDOWS\Fonts\COUR.TTF"
  FontRenderingEmSize = "36"
  StyleSimulations   = "BoldSimulation"
  UnicodeString     = "Hello World!"
  Fill              = "Maroon"
  OriginX           = "50"
  OriginY           = "300"
/>

<!-- "Open file" without "fi" ligature -->
<Glyphs
  FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
  FontRenderingEmSize = "36"
  StyleSimulations   = "BoldSimulation"
  UnicodeString     = "Open file"
  Fill              = "SlateGray"
  OriginX           = "400"
  OriginY           = "75"
/>

<!-- "Open file" with "fi" ligature -->
<Glyphs
  FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
  FontRenderingEmSize = "36"
  StyleSimulations   = "BoldSimulation"
  UnicodeString     = "Open file"
  Indices           = ";;;;;(2:1)191"
  Fill              = "SlateGray"
  OriginX           = "400"
  OriginY           = "150"
/>

</Canvas>

```

## See also

- [Typography in WPF](#)
- [Documents in WPF](#)
- [Text](#)

# Draw Text Using Glyphs

4/8/2019 • 2 minutes to read • [Edit Online](#)

This topic explains how to use the low-level [Glyphs](#) object to display text in Extensible Application Markup Language (XAML).

## Example

The following examples show how to define properties for a [Glyphs](#) object in Extensible Application Markup Language (XAML). The [Glyphs](#) object represents the output of a [GlyphRun](#) in XAML. The examples assume that the Arial, Courier New, and Times New Roman fonts are installed in the C:\WINDOWS\Fonts folder on the local computer.

```
<!-- The example shows how to use a Glyphs object. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <StackPanel Background="PowderBlue">

        <Glyphs
            FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
            FontRenderingEmSize = "100"
            StyleSimulations   = "BoldSimulation"
            UnicodeString     = "Hello World!"
            Fill              = "Black"
            OriginX           = "100"
            OriginY           = "200"
        />

    </StackPanel>
</Page>
```

This example shows how to define other properties of [Glyphs](#) objects in XAML.

```
<!-- The example shows how to use different property settings of Glyphs objects. -->
<Canvas
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Background="PowderBlue"
    >

    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
        FontRenderingEmSize = "36"
        StyleSimulations   = "ItalicSimulation"
        UnicodeString     = "Hello World!"
        Fill              = "SteelBlue"
        OriginX           = "50"
        OriginY           = "75"
    />

    <!-- "Hello World!" with default kerning -->
    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
        FontRenderingEmSize = "36"
        UnicodeString     = "Hello World!"
```

```

        Fill          = "Maroon"
        OriginX      = "50"
        OriginY      = "150"
    />

<!-- "Hello World!" with explicit character widths for proportional font -->
<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
    FontRenderingEmSize = "36"
    UnicodeString     = "Hello World!"
    Indices           = ",80;,80;,80;,80;,80;,80;,80;,80;,80;,80;"
    Fill              = "Maroon"
    OriginX          = "50"
    OriginY          = "225"
/>

<!-- "Hello World!" with fixed-width font -->
<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\COUR.TTF"
    FontRenderingEmSize = "36"
    StyleSimulations = "BoldSimulation"
    UnicodeString     = "Hello World!"
    Fill              = "Maroon"
    OriginX          = "50"
    OriginY          = "300"
/>

<!-- "Open file" without "fi" ligature -->
<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
    FontRenderingEmSize = "36"
    StyleSimulations = "BoldSimulation"
    UnicodeString     = "Open file"
    Fill              = "SlateGray"
    OriginX          = "400"
    OriginY          = "75"
/>

<!-- "Open file" with "fi" ligature -->
<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
    FontRenderingEmSize = "36"
    StyleSimulations = "BoldSimulation"
    UnicodeString     = "Open file"
    Indices           = ";;;;;(2:1)191"
    Fill              = "SlateGray"
    OriginX          = "400"
    OriginY          = "150"
/>

</Canvas>

```

## See also

- [Typography in WPF](#)

# Typography How-to Topics

4/8/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to use Windows Presentation Foundation (WPF) support for rich presentation of text in your applications.

## In This Section

[Create a Text Decoration](#)

[Specify Whether a Hyperlink is Underlined](#)

[Apply Transforms to Text](#)

[Apply Animations to Text](#)

[Create Text with a Shadow](#)

[Create Outlined Text](#)

[Draw Text to a Control's Background](#)

[Draw Text to a Visual](#)

[Use Special Characters in XAML](#)

## See also

- [Typography](#)
- [Documents in WPF](#)
- [OpenType Font Features](#)

# How to: Create a Text Decoration

4/8/2019 • 3 minutes to read • [Edit Online](#)

A [TextDecoration](#) object is a visual ornamentation you can add to text. There are four types of text decorations: underline, baseline, strikethrough, and overline. The following example shows the locations of the text decorations relative to the text.



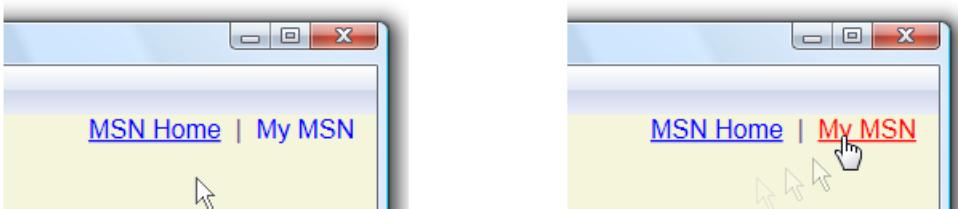
To add a text decoration to text, create a [TextDecoration](#) object and modify its properties. Use the [Location](#) property to specify where the text decoration appears, such as underline. Use the [Pen](#) property to specify the appearance of the text decoration, such as a solid fill or gradient color. If you do not specify a value for the [Pen](#) property, the decorations defaults to the same color as the text. Once you have defined a [TextDecoration](#) object, add it to the [TextDecorations](#) collection of the desired text object.

The following example shows a text decoration that has been styled with a linear gradient brush and a dashed pen.

emphasis

The [Hyperlink](#) object is an inline-level flow content element that allows you to host hyperlinks within the flow content. By default, [Hyperlink](#) uses a [TextDecoration](#) object to display an underline. [TextDecoration](#) objects can be performance intensive to instantiate, particularly if you have many [Hyperlink](#) objects. If you make extensive use of [Hyperlink](#) elements, you may want to consider showing an underline only when triggering an event, such as the [MouseEnter](#) event.

In the following example, the underline for the "My MSN" link is dynamic—it only appears when the [MouseEnter](#) event is triggered.



For more information, see [Specify Whether a Hyperlink is Underlined](#).

## Example

In the following code example, an underline text decoration uses the default font value.

```
// Use the default font values for the strikethrough text decoration.  
private void SetDefaultStrikethrough()  
{  
    // Set the underline decoration directly to the text block.  
    TextBlock1.TextDecorations = TextDecorations.Strikethrough;  
}
```

```
' Use the default font values for the strikethrough text decoration.
Private Sub SetDefaultStrikethrough()
    ' Set the underline decoration directly to the text block.
    TextBlock1.TextDecorations = TextDecorations.Strikethrough
End Sub
```

```
<!-- Use the default font values for the strikethrough text decoration. -->
<TextBlock
    TextDecorations="Strikethrough"
    FontSize="36" >
    The quick red fox
</TextBlock>
```

In the following code example, an underline text decoration is created with a solid color brush for the pen.

```
// Use a Red pen for the underline text decoration.
private void SetRedUnderline()
{
    // Create an underline text decoration. Default is underline.
    TextDecoration myUnderline = new TextDecoration();

    // Create a solid color brush pen for the text decoration.
    myUnderline.Pen = new Pen(Brushes.Red, 1);
    myUnderline.PenThicknessUnit = TextDecorationUnit.FontRecommended;

    // Set the underline decoration to a TextDecorationCollection and add it to the text block.
    TextDecorationCollection myCollection = new TextDecorationCollection();
    myCollection.Add(myUnderline);
    TextBlock2.TextDecorations = myCollection;
}
```

```
' Use a Red pen for the underline text decoration.
Private Sub SetRedUnderline()
    ' Create an underline text decoration. Default is underline.
    Dim myUnderline As New TextDecoration()

    ' Create a solid color brush pen for the text decoration.
    myUnderline.Pen = New Pen(Brushes.Red, 1)
    myUnderline.PenThicknessUnit = TextDecorationUnit.FontRecommended

    ' Set the underline decoration to a TextDecorationCollection and add it to the text block.
    Dim myCollection As New TextDecorationCollection()
    myCollection.Add(myUnderline)
    TextBlock2.TextDecorations = myCollection
End Sub
```

```

<!-- Use a Red pen for the underline text decoration -->
<TextBlock
    FontSize="36" >
    jumps over
<TextBlock.TextDecorations>
    <TextDecorationCollection>
        <TextDecoration
            PenThicknessUnit="FontRecommended">
            <TextDecoration.Pen>
                <Pen Brush="Red" Thickness="1" />
            </TextDecoration.Pen>
        </TextDecoration>
    </TextDecorationCollection>
</TextBlock.TextDecorations>
</TextBlock>

```

In the following code example, an underline text decoration is created with a linear gradient brush for the dashed pen.

```

// Use a linear gradient pen for the underline text decoration.
private void SetLinearGradientUnderline()
{
    // Create an underline text decoration. Default is underline.
    TextDecoration myUnderline = new TextDecoration();

    // Create a linear gradient pen for the text decoration.
    Pen myPen = new Pen();
    myPen.Brush = new LinearGradientBrush(Colors.Yellow, Colors.Red, new Point(0, 0.5), new Point(1, 0.5));
    myPen.Opacity = 0.5;
    myPen.Thickness = 1.5;
    myPen.DashStyle = DashStyles.Dash;
    myUnderline.Pen = myPen;
    myUnderline.PenThicknessUnit = TextDecorationUnit.FontRecommended;

    // Set the underline decoration to a TextDecorationCollection and add it to the text block.
    TextDecorationCollection myCollection = new TextDecorationCollection();
    myCollection.Add(myUnderline);
    TextBlock3.TextDecorations = myCollection;
}

```

```

' Use a linear gradient pen for the underline text decoration.
Private Sub SetLinearGradientUnderline()
    ' Create an underline text decoration. Default is underline.
    Dim myUnderline As New TextDecoration()

    ' Create a linear gradient pen for the text decoration.
    Dim myPen As New Pen()
    myPen.Brush = New LinearGradientBrush(Colors.Yellow, Colors.Red, New Point(0, 0.5), New Point(1, 0.5))
    myPen.Opacity = 0.5
    myPen.Thickness = 1.5
    myPen.DashStyle = DashStyles.Dash
    myUnderline.Pen = myPen
    myUnderline.PenThicknessUnit = TextDecorationUnit.FontRecommended

    ' Set the underline decoration to a TextDecorationCollection and add it to the text block.
    Dim myCollection As New TextDecorationCollection()
    myCollection.Add(myUnderline)
    TextBlock3.TextDecorations = myCollection
End Sub

```

```
<!-- Use a linear gradient pen for the underline text decoration. -->
<TextBlock FontSize="36">the lazy brown dog.
<TextBlock.TextDecorations>
  <TextDecorationCollection>
    <TextDecoration
      PenThicknessUnit="FontRecommended">
      <TextDecoration.Pen>
        <Pen Thickness="1.5">
          <Pen.Brush>
            <LinearGradientBrush Opacity="0.5"
              StartPoint="0,0.5" EndPoint="1,0.5">
              <LinearGradientBrush.GradientStops>
                <GradientStop Color="Yellow" Offset="0" />
                <GradientStop Color="Red" Offset="1" />
              </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
          </Pen.Brush>
          <Pen.DashStyle>
            <DashStyle Dashes="2"/>
          </Pen.DashStyle>
        </Pen>
      </TextDecoration.Pen>
    </TextDecoration>
  </TextDecorationCollection>
</TextBlock.TextDecorations>
</TextBlock>
```

## See also

- [TextDecoration](#)
- [Hyperlink](#)
- [Specify Whether a Hyperlink is Underlined](#)

# How to: Specify Whether a Hyperlink is Underlined

4/8/2019 • 2 minutes to read • [Edit Online](#)

The [Hyperlink](#) object is an inline-level flow content element that allows you to host hyperlinks within the flow content. By default, [Hyperlink](#) uses a [TextDecoration](#) object to display an underline. [TextDecoration](#) objects can be performance intensive to instantiate, particularly if you have many [Hyperlink](#) objects. If you make extensive use of [Hyperlink](#) elements, you may want to consider showing an underline only when triggering an event, such as the [MouseEnter](#) event.

In the following example, the underline for the "My MSN" link is dynamic, that is, it only appears when the [MouseEnter](#) event is triggered.



## Example

The following markup sample shows a [Hyperlink](#) defined with and without an underline:

```
<!-- Hyperlink with default underline. -->
<Hyperlink NavigateUri="http://www.msn.com">
    MSN Home
</Hyperlink>

<Run Text=" | " />

<!-- Hyperlink with no underline. -->
<Hyperlink Name="myHyperlink" TextDecorations="None"
    MouseEnter="OnMouseEnter"
    MouseLeave="OnMouseLeave"
    NavigateUri="http://www.msn.com">
    My MSN
</Hyperlink>
```

The following code sample shows how to create an underline for the [Hyperlink](#) on the [MouseEnter](#) event, and remove it on the [MouseLeave](#) event.

```
// Display the underline on only the MouseEnter event.
private void OnMouseEnter(object sender, EventArgs e)
{
    myHyperlink.TextDecorations = TextDecorations.Underline;
}

// Remove the underline on the MouseLeave event.
private void OnMouseLeave(object sender, EventArgs e)
{
    myHyperlink.TextDecorations = null;
}
```

```
' Display the underline on only the MouseEnter event.  
Private Overloads Sub OnMouseEnter(ByVal sender As Object, ByVal e As EventArgs)  
    myHyperlink.TextDecorations = TextDecorations.Underline  
End Sub  
  
' Remove the underline on the MouseLeave event.  
Private Overloads Sub OnMouseLeave(ByVal sender As Object, ByVal e As EventArgs)  
    myHyperlink.TextDecorations = Nothing  
End Sub
```

## See also

- [TextDecoration](#)
- [Hyperlink](#)
- [Optimizing WPF Application Performance](#)
- [Create a Text Decoration](#)

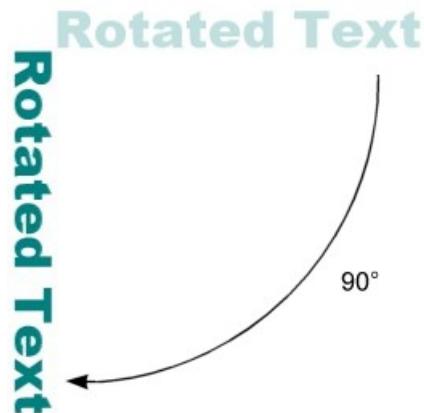
# How to: Apply Transforms to Text

8/22/2019 • 2 minutes to read • [Edit Online](#)

Transforms can alter the display of text in your application. The following examples use different types of rendering transforms to affect the display of text in a [TextBlock](#) control.

## Example

The following example shows text rotated about a specified point in the two-dimensional x-y plane.



The following code example uses a [RotateTransform](#) to rotate text. An [Angle](#) value of 90 rotates the element 90 degrees clockwise.

```
<!-- Rotate the text 90 degrees using a RotateTransform. -->
<TextBlock FontFamily="Arial Black" FontSize="64" Foreground="Moccasin" Margin ="80, 10, 0, 0">
    Text Transforms
    <TextBlock.RenderTransform>
        <RotateTransform Angle="90" />
    </TextBlock.RenderTransform>
</TextBlock>
```

The following example shows the second line of text scaled by 150% along the x-axis, and the third line of text scaled by 150% along the y-axis.

**Scaled Text**  
**Scaled Text**  
**Scaled Text**

The following code example uses a [ScaleTransform](#) to scale text from its original size.

```

<!-- Scale the text using a ScaleTransform. -->
<TextBlock
    Name="textblockScaleMaster"
    FontSize="32"
    Foreground="SteelBlue"
    Text="Scaled Text"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="0">
</TextBlock>
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="SteelBlue"
    Text="{Binding Path=Text, ElementName=textblockScaleMaster}"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="1">
    <TextBlock.RenderTransform>
        <ScaleTransform ScaleX="1.5" ScaleY="1.0" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="SteelBlue"
    Text="{Binding Path=Text, ElementName=textblockScaleMaster}"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="2">
    <TextBlock.RenderTransform>
        <ScaleTransform ScaleX="1.0" ScaleY="1.5" />
    </TextBlock.RenderTransform>
</TextBlock>

```

#### NOTE

Scaling text is not the same as increasing the font size of text. Font sizes are calculated independently of each other in order to provide the best resolution at different sizes. Scaled text, on the other hand, preserves the proportions of the original-sized text.

The following example shows text skewed along the x-axis.

*Skewed Text*  
*Skewed Text*

The following code example uses a [SkewTransform](#) to skew text. A skew, also known as a shear, is a transformation that stretches the coordinate space in a non-uniform manner. In this example, the two text strings are skewed -30° and 30° along the x-coordinate.

```

<!-- Skew the text using a SkewTransform. -->
<TextBlock
    Name="textblockSkewMaster"
    FontSize="32"
    FontWeight="Bold"
    Foreground="Maroon"
    Text="Skewed Text"
    Margin="125, 0, 0, 0"
    Grid.Column="0" Grid.Row="0">
    <TextBlock.RenderTransform>
        <SkewTransform AngleX="-30" AngleY="0" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="Maroon"
    Text="{Binding Path=Text, ElementName=textblockSkewMaster}"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="1">
    <TextBlock.RenderTransform>
        <SkewTransform AngleX="30" AngleY="0" />
    </TextBlock.RenderTransform>
</TextBlock>

```

The following example shows text translated, or moved, along the x- and y-axis.

## Translated Text

The following code example uses a [TranslateTransform](#) to offset text. In this example, a slightly offset copy of text below the primary text creates a shadow effect.

```

<!-- Skew the text using a TranslateTransform. -->
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="Black"
    Text="{Binding Path=Text, ElementName=textblockTranslateMaster}"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="0">
    <TextBlock.RenderTransform>
        <TranslateTransform X="2" Y="2" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock
    Name="textblockTranslateMaster"
    FontSize="32"
    FontWeight="Bold"
    Foreground="Coral"
    Text="Translated Text"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="0"/>

```

### NOTE

The [DropShadowBitmapEffect](#) provides a rich set of features for providing shadow effects. For more information, see [Create Text with a Shadow](#).

**See also**

- [Apply Animations to Text](#)

# How to: Apply Animations to Text

4/8/2019 • 2 minutes to read • [Edit Online](#)

Animations can alter the display and appearance of text in your application. The following examples use different types of animations to affect the display of text in a [TextBlock](#) control.

## Example

The following example uses a [DoubleAnimation](#) to animate the width of the text block. The width value changes from the width of the text block to 0 over a duration of 10 seconds, and then reverses the width values and continues. This type of animation creates a wipe effect.

```
<TextBlock  
    Name="MyWipedText"  
    Margin="20"  
    Width="480" Height="100" FontSize="48" FontWeight="Bold" Foreground="Maroon">  
    This is wiped text  
  
    <!-- Animates the text block's width. -->  
    <TextBlock.Triggers>  
        <EventTrigger RoutedEvent="TextBlock.Loaded">  
            <BeginStoryboard>  
                <Storyboard>  
                    <DoubleAnimation  
                        Storyboard.TargetName="MyWipedText"  
                        Storyboard.TargetProperty="(TextBlock.Width)"  
                        To="0.0" Duration="0:0:10"  
                        AutoReverse="True" RepeatBehavior="Forever" />  
                </Storyboard>  
            </BeginStoryboard>  
        </EventTrigger>  
    </TextBlock.Triggers>  
</TextBlock>
```

The following example uses a [DoubleAnimation](#) to animate the opacity of the text block. The opacity value changes from 1.0 to 0 over a duration of 5 seconds, and then reverses the opacity values and continues.

```

<TextBlock
    Name="MyFadingText"
    Margin="20"
    Width="640" Height="100" FontSize="48" FontWeight="Bold" Foreground="Maroon">
    This is fading text

    <!-- Animates the text block's opacity. -->
    <TextBlock.Triggers>
        <EventTrigger RoutedEvent="TextBlock.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Storyboard.TargetName="MyFadingText"
                        Storyboard.TargetProperty="(TextBlock.Opacity)"
                        From="1.0" To="0.0" Duration="0:0:5"
                        AutoReverse="True" RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </TextBlock.Triggers>
</TextBlock>

```

The following diagram shows the effect of the [TextBlock](#) control changing its opacity from `1.00` to `0.00` during the 5-second interval defined by the [Duration](#).

Opacity	
<b>This is fading text</b>	<b>1.00</b>
<b>This is fading text</b>	<b>0.80</b>
<b>This is fading text</b>	<b>0.60</b>
<b>This is fading text</b>	<b>0.40</b>
<b>This is fading text</b>	<b>0.20</b>
	<b>0.00</b>

The following example uses a [ColorAnimation](#) to animate the foreground color of the text block. The foreground color value changes from one color to a second color over a duration of 5 seconds, and then reverses the color values and continues.

```

<TextBlock
    Name="MyChangingColorText"
    Margin="20"
    Width="640" Height="100" FontSize="48" FontWeight="Bold">
    This is changing color text
    <TextBlock.Foreground>
        <SolidColorBrush x:Name="MySolidColorBrush" Color="Maroon" />
    </TextBlock.Foreground>

    <!-- Animates the text block's color. -->
    <TextBlock.Triggers>
        <EventTrigger RoutedEvent="TextBlock.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <ColorAnimation
                        Storyboard.TargetName="MySolidColorBrush"
                        Storyboard.TargetProperty="Color"
                        From="DarkOrange" To="SteelBlue" Duration="0:0:5"
                        AutoReverse="True" RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </TextBlock.Triggers>
</TextBlock>

```

The following example uses a [DoubleAnimation](#) to rotate the text block. The text block performs a full rotation over a duration of 20 seconds, and then continues to repeat the rotation.

```

<TextBlock
    Name="MyRotatingText"
    Margin="20"
    Width="640" Height="100" FontSize="48" FontWeight="Bold" Foreground="Teal"
    >
    This is rotating text
    <TextBlock.RenderTransform>
        <RotateTransform x:Name="MyRotateTransform" Angle="0" CenterX="230" CenterY="25"/>
    </TextBlock.RenderTransform>

    <!-- Animates the text block's rotation. -->
    <TextBlock.Triggers>
        <EventTrigger RoutedEvent="TextBlock.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Storyboard.TargetName="MyRotateTransform"
                        Storyboard.TargetProperty="(RotateTransform.Angle)"
                        From="0.0" To="360" Duration="0:0:10"
                        RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </TextBlock.Triggers>
</TextBlock>

```

## See also

- [Animation Overview](#)

# How to: Create Text with a Shadow

8/22/2019 • 2 minutes to read • [Edit Online](#)

The examples in this section show how to create a shadow effect for displayed text.

## Example

The [DropShadowEffect](#) object allows you to create a variety of drop shadow effects for Windows Presentation Foundation (WPF) objects. The following example shows a drop shadow effect applied to text. In this case, the shadow is a soft shadow, which means the shadow color blurs.

## Shadow Text

You can control the width of a shadow by setting the [ShadowDepth](#) property. A value of `4.0` indicates a shadow width of 4 pixels. You can control the softness, or blur, of a shadow by modifying the [BlurRadius](#) property. A value of `0.0` indicates no blurring. The following code example shows how to create a soft shadow.

```
<!-- Soft single shadow. -->
<TextBlock
    Text="Shadow Text"
    Foreground="Teal">
    <TextBlock.Effect>
        <DropShadowEffect
            ShadowDepth="4"
            Direction="330"
            Color="Black"
            Opacity="0.5"
            BlurRadius="4"/>
    </TextBlock.Effect>
</TextBlock>
```

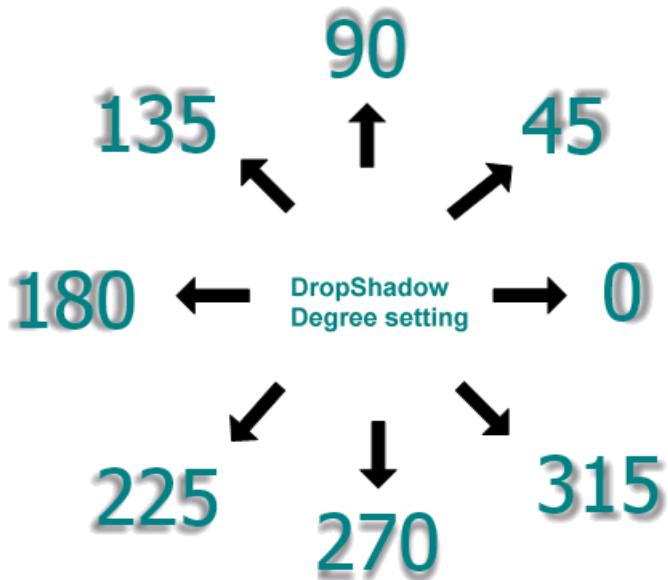
### NOTE

These shadow effects do not go through the Windows Presentation Foundation (WPF) text rendering pipeline. As a result, ClearType is disabled when using these effects.

The following example shows a hard drop shadow effect applied to text. In this case, the shadow is not blurred.

## Shadow Text

You can create a hard shadow by setting the [BlurRadius](#) property to `0.0`, which indicates that no blurring is used. You can control the direction of the shadow by modifying the [Direction](#) property. Set the directional value of this property to a degree between `0` and `360`. The following illustration shows the directional values of the [Direction](#) property setting.



The following code example shows how to create a hard shadow.

```
<!-- Hard single shadow. -->
<TextBlock
    Text="Shadow Text"
    Foreground="Maroon">
    <TextBlock.Effect>
        <DropShadowEffect
            ShadowDepth="6"
            Direction="135"
            Color="Maroon"
            Opacity="0.35"
            BlurRadius="0.0" />
    </TextBlock.Effect>
</TextBlock>
```

## Using a Blur Effect

A [BlurBitmapEffect](#) can be used to create a shadow-like effect that can be placed behind a text object. A blur bitmap effect applied to text blurs the text evenly in all directions.

The following example shows a blur effect applied to text.

# Shadow Text

The following code example shows how to create a blur effect.

```
<!-- Shadow effect by creating a blur. -->
<TextBlock
    Text="Shadow Text"
    Foreground="Green"
    Grid.Column="0" Grid.Row="0" >
    <TextBlock.Effect>
        <BlurEffect
            Radius="8.0"
            KernelType="Box"/>
    </TextBlock.Effect>
</TextBlock>
<TextBlock
    Text="Shadow Text"
    Foreground="Maroon"
    Grid.Column="0" Grid.Row="0" />
```

## Using a Translate Transform

A [TranslateTransform](#) can be used to create a shadow-like effect that can be placed behind a text object.

The following code example uses a [TranslateTransform](#) to offset text. In this example, a slightly offset copy of text below the primary text creates a shadow effect.

# Shadow Text

The following code example shows how to create a transform for a shadow effect.

```
<!-- Shadow effect by creating a transform. -->
<TextBlock
    Foreground="Black"
    Text="Shadow Text"
    Grid.Column="0" Grid.Row="0">
    <TextBlock.RenderTransform>
        <TranslateTransform X="3" Y="3" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock
    Foreground="Coral"
    Text="Shadow Text"
    Grid.Column="0" Grid.Row="0" >
</TextBlock>
```

# How to: Create Outlined Text

4/8/2019 • 3 minutes to read • [Edit Online](#)

In most cases, when you are adding ornamentation to text strings in your Windows Presentation Foundation (WPF) application, you are using text in terms of a collection of discrete characters, or glyphs. For example, you could create a linear gradient brush and apply it to the [Foreground](#) property of a [TextBox](#) object. When you display or edit the text box, the linear gradient brush is automatically applied to the current set of characters in the text string.

## Spectrum Foreground

However, you can also convert text into [Geometry](#) objects, allowing you to create other types of visually rich text. For example, you could create a [Geometry](#) object based on the outline of a text string.

## Spectrum Outline

When text is converted to a [Geometry](#) object, it is no longer a collection of characters—you cannot modify the characters in the text string. However, you can affect the appearance of the converted text by modifying its stroke and fill properties. The stroke refers to the outline of the converted text; the fill refers to the area inside the outline of the converted text.

The following examples illustrate several ways of creating visual effects by modifying the stroke and fill of converted text.

### Fancy Outlined Text

### BUTTERFLIES

It is also possible to modify the bounding box rectangle, or highlight, of the converted text. The following example illustrates a way to creating visual effects by modifying the stroke and highlight of converted text.

### WILDFIRE

#### Example

The key to converting text to a [Geometry](#) object is to use the [FormattedText](#) object. Once you have created this object, you can use the [BuildGeometry](#) and [BuildHighlightGeometry](#) methods to convert the text to [Geometry](#) objects. The first method returns the geometry of the formatted text; the second method returns the geometry of the formatted text's bounding box. The following code example shows how to create a [FormattedText](#) object and to retrieve the geometries of the formatted text and its bounding box.

```

/// <summary>
/// Create the outline geometry based on the formatted text.
/// </summary>
public void CreateText()
{
    System.Windows.FontStyle fontStyle = FontStyles.Normal;
    FontWeight fontWeight = FontWeights.Medium;

    if (Bold == true) fontWeight = FontWeights.Bold;
    if (Italic == true) fontStyle = FontStyles.Italic;

    // Create the formatted text based on the properties set.
    FormattedText formattedText = new FormattedText(
        Text,
        CultureInfo.GetCultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface(
            Font,
            fontStyle,
            fontWeight,
            FontStretches.Normal),
        FontSize,
        System.Windows.Media.Brushes.Black // This brush does not matter since we use the geometry of the
text.
    );

    // Build the geometry object that represents the text.
    _textGeometry = formattedText.BuildGeometry(new System.Windows.Point(0, 0));

    // Build the geometry object that represents the text highlight.
    if (Highlight == true)
    {
        _textHighLightGeometry = formattedText.BuildHighlightGeometry(new System.Windows.Point(0, 0));
    }
}

```

```

''' <summary>
''' Create the outline geometry based on the formatted text.
''' </summary>
Public Sub CreateText()
    Dim fontStyle As FontStyle = FontStyles.Normal
    Dim fontWeight As FontWeight = FontWeights.Medium

    If Bold = True Then
        fontWeight = FontWeights.Bold
    End If
    If Italic = True Then
        fontStyle = FontStyles.Italic
    End If

    ' Create the formatted text based on the properties set.
    Dim formattedText As New FormattedText(Text, CultureInfo.GetCultureInfo("en-us"),
    FlowDirection.LeftToRight, New Typeface(Font, fontStyle, fontWeight, FontStretches.Normal), FontSize,
    Brushes.Black) ' This brush does not matter since we use the geometry of the text.

    ' Build the geometry object that represents the text.
    _textGeometry = formattedText.BuildGeometry(New Point(0, 0))

    ' Build the geometry object that represents the text highlight.
    If Highlight = True Then
        _textHighLightGeometry = formattedText.BuildHighlightGeometry(New Point(0, 0))
    End If
End Sub

```

In order to display the retrieved the [Geometry](#) objects, you need to access the [DrawingContext](#) of the object that is displaying the converted text. In these code examples, this is done by creating a custom control object that is derived from a class that supports user-defined rendering.

To display [Geometry](#) objects in the custom control, provide an override for the [OnRender](#) method. Your overridden method should use the [DrawGeometry](#) method to draw the [Geometry](#) objects.

```
/// <summary>
/// OnRender override draws the geometry of the text and optional highlight.
/// </summary>
/// <param name="drawingContext">Drawing context of the OutlineText control.</param>
protected override void OnRender(DrawingContext drawingContext)
{
    // Draw the outline based on the properties that are set.
    drawingContext.DrawGeometry(Fill, new System.Windows.Media.Pen(Stroke, StrokeThickness), _textGeometry);

    // Draw the text highlight based on the properties that are set.
    if (Highlight == true)
    {
        drawingContext.DrawGeometry(null, new System.Windows.Media.Pen(Stroke, StrokeThickness),
        _textHighLightGeometry);
    }
}
```

```
''' <summary>
''' OnRender override draws the geometry of the text and optional highlight.
''' </summary>
''' <param name="drawingContext">Drawing context of the OutlineText control.</param>
Protected Overrides Sub OnRender(ByVal drawingContext As DrawingContext)
    ' Draw the outline based on the properties that are set.
    drawingContext.DrawGeometry(Fill, New Pen(Stroke, StrokeThickness), _textGeometry)

    ' Draw the text highlight based on the properties that are set.
    If Highlight = True Then
        drawingContext.DrawGeometry(Nothing, New Pen(Stroke, StrokeThickness), _textHighLightGeometry)
    End If
End Sub
```

For the source of the example custom user control object, see [OutlineTextControl.cs for C#](#) and [OutlineTextControl.vb for Visual Basic](#).

## See also

- [Drawing Formatted Text](#)

# How to: Draw Text to a Control's Background

11/1/2019 • 2 minutes to read • [Edit Online](#)

You can draw text directly to the background of a control by converting a text string to a [FormattedText](#) object, and then drawing the object to the control's [DrawingContext](#). You can also use this technique for drawing to the background of objects derived from [Panel](#), such as [Canvas](#) and [StackPanel](#).

**My Custom Label**

**Display Text**

Example of controls with custom text backgrounds

## Example

To draw to the background of a control, create a new [DrawingBrush](#) object and draw the converted text to the object's [DrawingContext](#). Then, assign the new [DrawingBrush](#) to the control's [Background](#) property.

The following code example shows how to create a [FormattedText](#) object and draw to the background of a [Label](#) and [Button](#) object.

```

// Handle the WindowLoaded event for the window.
private void WindowLoaded(object sender, EventArgs e)
{
    // Update the background property of the label and button.
    myLabel.Background = new DrawingBrush(DrawMyText("My Custom Label"));
    myButton.Background = new DrawingBrush(DrawMyText("Display Text"));
}

// Convert the text string to a geometry and draw it to the control's DrawingContext.
private Drawing DrawMyText(string textString)
{
    // Create a new DrawingGroup of the control.
    DrawingGroup drawingGroup = new DrawingGroup();

    // Open the DrawingGroup in order to access the DrawingContext.
    using (DrawingContext drawingContext = drawingGroup.Open())
    {
        // Create the formatted text based on the properties set.
        FormattedText formattedText = new FormattedText(
            textString,
            CultureInfo.GetCultureInfo("en-us"),
            FlowDirection.LeftToRight,
            new Typeface("Comic Sans MS Bold"),
            48,
            System.Windows.Media.Brushes.Black // This brush does not matter since we use the geometry of the
text.
        );

        // Build the geometry object that represents the text.
        Geometry textGeometry = formattedText.BuildGeometry(new System.Windows.Point(20, 0));

        // Draw a rounded rectangle under the text that is slightly larger than the text.
        drawingContext.DrawRoundedRectangle(System.Windows.Media.Brushes.PapayaWhip, null, new Rect(new
System.Windows.Size(formattedText.Width + 50, formattedText.Height + 5)), 5.0, 5.0);

        // Draw the outline based on the properties that are set.
        drawingContext.DrawGeometry(System.Windows.Media.Brushes.Gold, new
System.Windows.Media.Pen(System.Windows.Media.Brushes.Maroon, 1.5), textGeometry);

        // Return the updated DrawingGroup content to be used by the control.
        return drawingGroup;
    }
}

```

## See also

- [FormattedText](#)
- [Drawing Formatted Text](#)

# How to: Draw Text to a Visual

8/22/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to draw text to a [DrawingVisual](#) using a [DrawingContext](#) object. A drawing context is returned by calling the [RenderOpen](#) method of a [DrawingVisual](#) object. You can draw graphics and text into a drawing context.

To draw text into the drawing context, use the [DrawText](#) method of a [DrawingContext](#) object. When you are finished drawing content into the drawing context, call the [Close](#) method to close the drawing context and persist the content.

## Example

```
// Create a DrawingVisual that contains text.
private DrawingVisual CreateDrawingVisualText()
{
    // Create an instance of a DrawingVisual.
    DrawingVisual drawingVisual = new DrawingVisual();

    // Retrieve the DrawingContext from the DrawingVisual.
    DrawingContext drawingContext = drawingVisual.RenderOpen();

    // Draw a formatted text string into the DrawingContext.
    drawingContext.DrawText(
        new FormattedText("Click Me!",
            CultureInfo.GetCultureInfo("en-us"),
            FlowDirection.LeftToRight,
            new Typeface("Verdana"),
            36, System.Windows.Media.Brushes.Black),
        new System.Windows.Point(200, 116));

    // Close the DrawingContext to persist changes to the DrawingVisual.
    drawingContext.Close();

    return drawingVisual;
}
```

```
' Create a DrawingVisual that contains text.
Private Function CreateDrawingVisualText() As DrawingVisual
    ' Create an instance of a DrawingVisual.
    Dim drawingVisual As New DrawingVisual()

    ' Retrieve the DrawingContext from the DrawingVisual.
    Dim drawingContext As DrawingContext = drawingVisual.RenderOpen()

    ' Draw a formatted text string into the DrawingContext.
    drawingContext.DrawText(New FormattedText("Click Me!", CultureInfo.GetCultureInfo("en-us"),
        FlowDirection.LeftToRight, New Typeface("Verdana"), 36, Brushes.Black), New Point(200, 116))

    ' Close the DrawingContext to persist changes to the DrawingVisual.
    drawingContext.Close()

    Return drawingVisual
End Function
```

**NOTE**

For the complete code sample from which the preceding code example was extracted, see [Hit Test Using DrawingVisuals Sample](#).

# How to: Use Special Characters in XAML

11/7/2019 • 2 minutes to read • [Edit Online](#)

Markup files that are created in Visual Studio are automatically saved in the Unicode UTF-8 file format, which means that most special characters, such as accent marks, are encoded correctly. However, there is a set of commonly-used special characters that are handled differently. These special characters follow the World Wide Web Consortium (W3C) XML standard for encoding.

The following table shows the syntax for encoding this set of special characters:

CHARACTER	SYNTAX	DESCRIPTION
<	&lt;	Less than symbol.
>	&gt;	Greater than sign.
&	&amp;	Ampersand symbol.
"	&quot;	Double quote symbol.

## NOTE

If you create a markup file using a text editor, such as Windows Notepad, you must save the file in the Unicode UTF-8 file format in order to preserve any encoded special characters.

The following example shows how you can use special characters in text when creating markup.

## Example

```
<!-- Display special characters that require special encoding: < > & " -->
<TextBlock>
    &lt;    <!-- Less than symbol -->
    &gt;    <!-- Greater than symbol -->
    &amp;   <!-- Ampersand symbol -->
    &quot;  <!-- Double quote symbol -->
</TextBlock>

<!-- Display miscellaneous special characters -->
<TextBlock>
    Cæsar  <!-- AE diphthong symbol -->
    © 2006  <!-- Copyright symbol -->
    Español <!-- Tilde symbol -->
    ¥      <!-- Yen symbol -->
</TextBlock>
```

# Printing and Print System Management

11/12/2019 • 2 minutes to read • [Edit Online](#)

Windows Vista and Microsoft .NET Framework introduce a new print path — an alternative to Microsoft Windows Graphics Device Interface (GDI) printing — and a vastly expanded set of print system management APIs.

## In This Section

### [Printing Overview](#)

A discussion of the new print path and APIs.

### [How-to Topics](#)

A set of articles showing how to use the new print path and APIs.

## See also

- [System.Printing](#)
- [System.Printing.IndexedProperties](#)
- [System.Printing.Interop](#)
- [Documents in WPF](#)
- [XPS Documents](#)

# Printing Overview

11/12/2019 • 11 minutes to read • [Edit Online](#)

With Microsoft .NET Framework, application developers using Windows Presentation Foundation (WPF) have a rich new set of printing and print system management APIs. With Windows Vista, some of these print system enhancements are also available to developers creating Windows Forms applications and developers using unmanaged code. At the core of this new functionality is the new XML Paper Specification (XPS) file format and the XPS print path.

This topic contains the following sections.

## About XPS

XPS is an electronic document format, a spool file format and a page description language. It is an open document format that uses XML, Open Packaging Conventions (OPC), and other industry standards to create cross-platform documents. XPS simplifies the process by which digital documents are created, shared, printed, viewed, and archived. For additional information on XPS, see [XPS Documents](#).

Several techniques for printing XPS-based content using WPF are demonstrated in [Programmatically Print XPS Files](#). You may find it useful to reference these samples during review of content contained in this topic. (Unmanaged code developers should see documentation for the [MXDC\\_ESCAPE function](#). Windows Forms developers must use the API in the [System.Drawing.Printing](#) namespace which does not support the full XPS print path, but does support a hybrid GDI-to-XPS print path. See [Print Path Architecture](#) below.)

## XPS Print Path

The XML Paper Specification (XPS) print path is a new Windows feature that redefines how printing is handled in Windows applications. Because XPS can replace a document presentation language (such as RTF), a print spooler format (such as WMF), and a page description language (such as PCL or Postscript); the new print path maintains the XPS format from application publication to the final processing in the print driver or device.

The XPS print path is built upon the XPS printer driver model (XPSSDrv), which provides several benefits for developers such as "what you see is what you get" (WYSIWYG) printing, improved color support, and significantly improved print performance. (For more on XPSSDrv, see the [Windows Driver Kit documentation](#).)

The operation of the print spooler for XPS documents is essentially the same as in previous versions of Windows. However, it has been enhanced to support the XPS print path in addition to the existing GDI print path. The new print path natively consumes an XPS spool file. While user-mode printer drivers written for previous versions of Windows will continue to work, an XPS printer driver (XPSSDrv) is required in order to use the XPS print path.

The benefits of the XPS print path are significant, and include:

- WYSIWYG print support
- Native support of advanced color profiles, which include 32 bits per channel (bpc), CMYK, named-colors, n-inks, and native support of transparency and gradients.
- Improved print performance for both .NET Framework and Win32 based applications.
- Industry standard XPS format.

For basic print scenarios, a simple and intuitive API is available with a single entry point for user interface,

configuration and job submission. For advanced scenarios, an additional support is added for user interface (UI) customization (or no UI at all), synchronous or asynchronous printing, and batch printing capabilities. Both options provide print support in full or partial trust mode.

XPS was designed with extensibility in mind. By using the extensibility framework, features and capabilities can be added to XPS in a modular manner. Extensibility features include:

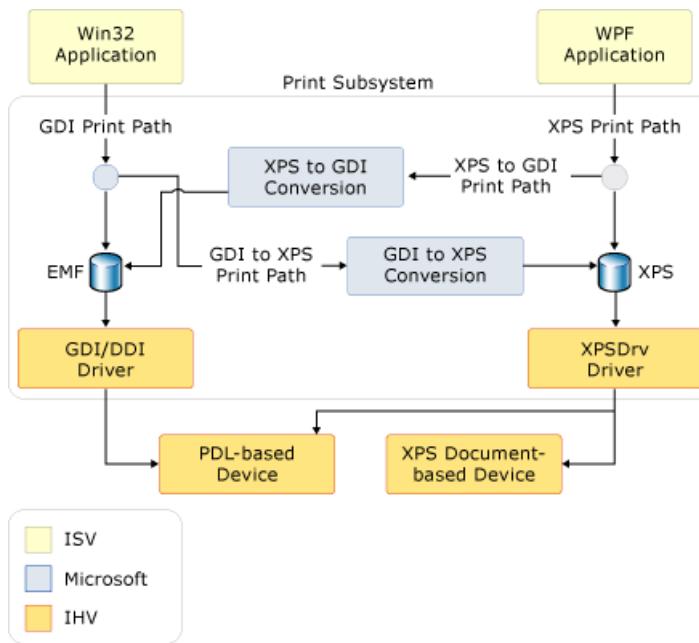
- Print Schema. The public schema is updated regularly and enables rapid extension of device capabilities. (See [PrintTicket and PrintCapabilities](#) below.)
- Extensible Filter Pipeline. The XPS printer driver (XPSDrv) filter pipeline was designed to enable both direct and scalable printing of XPS documents. For more information, see [XPSDrv Printer Drivers](#).

## Print Path Architecture

While both Win32 and .NET Framework applications support XPS, Win32 and Windows Forms applications use a GDI to XPS conversion in order to create XPS formatted content for the XPS printer driver (XPSDrv). These applications are not required to use the XPS print path, and can continue to use Enhanced Metafile (EMF) based printing. However, most XPS features and enhancements are only available to applications that target the XPS print path.

To enable the use of XPSDrv-based printers by Win32 and Windows Forms applications, the XPS printer driver (XPSDrv) supports conversion of GDI to XPS format. The XPSDrv model also provides a converter for XPS to GDI format so that Win32 applications can print XPS Documents. For WPF applications, conversion of XPS to GDI format is done automatically by the [Write](#) and [WriteAsync](#) methods of the [XpsDocumentWriter](#) class whenever the target print queue of the write operation does not have an XPSDrv driver. (Windows Forms applications cannot print XPS Documents.)

The following illustration depicts the print subsystem and defines the portions provided by Microsoft, and the portions defined by software and hardware vendors:



## Basic XPS Printing

WPF defines both a basic and advanced API. For those applications that do not require extensive print customization or access to the complete XPS feature set, basic print support is available. Basic print support is exposed through a print dialog control that requires minimal configuration and features a familiar UI. Many XPS features are available using this simplified print model.

### PrintDialog

The [System.Windows.Controls.PrintDialog](#) control provides a single entry point for UI, configuration, and XPS job submission. For information about how to instantiate and use the control, see [Invoke a Print Dialog](#).

## Advanced XPS Printing

To access the complete set of XPS features, the advanced print API must be used. Several relevant API are described in greater detail below. For a complete list of XPS print path APIs, see the [System.Windows.Xps](#) and [System.Printing](#) namespace references.

### PrintTicket and PrintCapabilities

The [PrintTicket](#) and [PrintCapabilities](#) classes are the foundation of the advanced XPS features. Both types of objects are XML formatted structures of print-oriented features such as collation, two-sided printing, stapling, etc. These structures are defined by the print schema. A [PrintTicket](#) instructs a printer how to process a print job. The [PrintCapabilities](#) class defines the capabilities of a printer. By querying the capabilities of a printer, a [PrintTicket](#) can be created that takes full advantage of a printer's supported features. Similarly, unsupported features can be avoided.

The following example demonstrates how to query the [PrintCapabilities](#) of a printer and create a [PrintTicket](#) using code.

```
// ----- GetPrintTicketFromPrinter -----
/// <summary>
///   Returns a PrintTicket based on the current default printer.</summary>
/// <returns>
///   A PrintTicket for the current local default printer.</returns>
PrintTicket^ GetPrintTicketFromPrinter ()
{
    PrintQueue^ printQueue = nullptr;

    LocalPrintServer^ localPrintServer = gcnew LocalPrintServer();

    // Retrieving collection of local printer on user machine
    PrintQueueCollection^ localPrinterCollection = localPrintServer->GetPrintQueues();

    System::Collections::IEnumerator^ localPrinterEnumerator = localPrinterCollection->GetEnumerator();

    if (localPrinterEnumerator->MoveNext())
    {
        // Get PrintQueue from first available printer
        printQueue = ((PrintQueue^)localPrinterEnumerator->Current);
    }
    else
    {
        return nullptr;
    }

    // Get default PrintTicket from printer
    PrintTicket^ printTicket = printQueue->DefaultPrintTicket;

    PrintCapabilities^ printCapabilites = printQueue->GetPrintCapabilities();

    // Modify PrintTicket
    if (printCapabilites->CollationCapability->Contains(Collation::Collated))
    {
        printTicket->Collation = Collation::Collated;
    }
    if (printCapabilites->DuplexingCapability->Contains(Duplexing::TwoSidedLongEdge))
    {
        printTicket->Duplexing = Duplexing::TwoSidedLongEdge;
    }
    if (printCapabilites->StaplingCapability->Contains(Stapling::StapleDualLeft))
    {
        printTicket->Stapling = Stapling::StapleDualLeft;
    }
    return printTicket;
};// end:GetPrintTicketFromPrinter()
```

```

// ----- GetPrintTicketFromPrinter -----
/// <summary>
///   Returns a PrintTicket based on the current default printer.</summary>
/// <returns>
///   A PrintTicket for the current local default printer.</returns>
private PrintTicket GetPrintTicketFromPrinter()
{
    PrintQueue printQueue = null;

    LocalPrintServer localPrintServer = new LocalPrintServer();

    // Retrieving collection of local printer on user machine
    PrintQueueCollection localPrinterCollection =
        localPrintServer.GetPrintQueues();

    System.Collections.IEnumerator localPrinterEnumerator =
        localPrinterCollection.GetEnumerator();

    if (localPrinterEnumerator.MoveNext())
    {
        // Get PrintQueue from first available printer
        printQueue = (PrintQueue)localPrinterEnumerator.Current;
    }
    else
    {
        // No printer exist, return null PrintTicket
        return null;
    }

    // Get default PrintTicket from printer
    PrintTicket printTicket = printQueue.DefaultPrintTicket;

    PrintCapabilities printCapabilites = printQueue.GetPrintCapabilities();

    // Modify PrintTicket
    if (printCapabilites.CollationCapability.Contains(Collation.Collated))
    {
        printTicket.Collation = Collation.Collated;
    }

    if ( printCapabilites.DuplexingCapability.Contains(
        Duplexing.TwoSidedLongEdge) )
    {
        printTicket.Duplexing = Duplexing.TwoSidedLongEdge;
    }

    if (printCapabilites.StaplingCapability.Contains(Stapling.StapleDualLeft))
    {
        printTicket.Stapling = Stapling.StapleDualLeft;
    }

    return printTicket;
}// end:GetPrintTicketFromPrinter()

```

```

' ----- GetPrintTicketFromPrinter -----
''' <summary>
'''   Returns a PrintTicket based on the current default printer.</summary>
''' <returns>
'''   A PrintTicket for the current local default printer.</returns>
Private Function GetPrintTicketFromPrinter() As PrintTicket
    Dim printQueue As PrintQueue = Nothing

    Dim localPrintServer As New LocalPrintServer()

    ' Retrieving collection of local printer on user machine
    Dim localPrinterCollection As PrintQueueCollection = localPrintServer.GetPrintQueues()

    Dim localPrinterEnumerator As System.Collections.IEnumerator = localPrinterCollection.GetEnumerator()

    If localPrinterEnumerator.MoveNext() Then
        ' Get PrintQueue from first available printer
        printQueue = CType(localPrinterEnumerator.Current, PrintQueue)
    Else
        ' No printer exist, return null PrintTicket
        Return Nothing
    End If

    ' Get default PrintTicket from printer
    Dim printTicket As PrintTicket = printQueue.DefaultPrintTicket

    Dim printCapabilites As PrintCapabilities = printQueue.GetPrintCapabilities()

    ' Modify PrintTicket
    If printCapabilites.CollationCapability.Contains(Collation.Collated) Then
        printTicket.Collation = Collation.Collated
    End If

    If printCapabilites.DuplexingCapability.Contains(Duplexing.TwoSidedLongEdge) Then
        printTicket.Duplexing = Duplexing.TwoSidedLongEdge
    End If

    If printCapabilites.StaplingCapability.Contains(Stapling.StapleDualLeft) Then
        printTicket.Stapling = Stapling.StapleDualLeft
    End If

    Return printTicket
End Function ' end:GetPrintTicketFromPrinter()

```

## PrintServer and PrintQueue

The [PrintServer](#) class represents a network print server and the [PrintQueue](#) class represents a printer and the output job queue associated with it. Together, these APIs allow advanced management of a server's print jobs. A [PrintServer](#), or one of its derived classes, is used to manage a [PrintQueue](#). The [AddJob](#) method is used to insert a new print job into the queue.

The following example demonstrates how to create a [LocalPrintServer](#) and access its default [PrintQueue](#) by using code.

```

// ----- GetPrintXpsDocumentWriter() -----
/// <summary>
///   Returns an XpsDocumentWriter for the default print queue.</summary>
/// <returns>
///   An XpsDocumentWriter for the default print queue.</returns>
private XpsDocumentWriter GetPrintXpsDocumentWriter()
{
    // Create a local print server
    LocalPrintServer ps = new LocalPrintServer();

    // Get the default print queue
    PrintQueue pq = ps.DefaultPrintQueue;

    // Get an XpsDocumentWriter for the default print queue
    XpsDocumentWriter xpsdw = PrintQueue.CreateXpsDocumentWriter(pq);
    return xpsdw;
}// end:GetPrintXpsDocumentWriter()

```

```

' ----- GetPrintXpsDocumentWriter() -----
''' <summary>
'''   Returns an XpsDocumentWriter for the default print queue.</summary>
''' <returns>
'''   An XpsDocumentWriter for the default print queue.</returns>
Private Function GetPrintXpsDocumentWriter() As XpsDocumentWriter
    ' Create a local print server
    Dim ps As New LocalPrintServer()

    ' Get the default print queue
    Dim pq As PrintQueue = ps.DefaultPrintQueue

    ' Get an XpsDocumentWriter for the default print queue
    Dim xpsdw As XpsDocumentWriter = PrintQueue.CreateXpsDocumentWriter(pq)
    Return xpsdw
End Function ' end:GetPrintXpsDocumentWriter()

```

## XpsDocumentWriter

An [XpsDocumentWriter](#), with its many the [Write](#) and [WriteAsync](#) methods, is used to write XPS documents to a [PrintQueue](#). For example, the [Write\(FixedPage, PrintTicket\)](#) method is used to output an XPS document and [PrintTicket](#) synchronously. The [WriteAsync\(FixedDocument, PrintTicket\)](#) method is used to output an XPS document and [PrintTicket](#) asynchronously.

The following example demonstrates how to create an [XpsDocumentWriter](#) using code.

```

// ----- GetPrintXpsDocumentWriter() -----
/// <summary>
///   Returns an XpsDocumentWriter for the default print queue.</summary>
/// <returns>
///   An XpsDocumentWriter for the default print queue.</returns>
private XpsDocumentWriter GetPrintXpsDocumentWriter()
{
    // Create a local print server
    LocalPrintServer ps = new LocalPrintServer();

    // Get the default print queue
    PrintQueue pq = ps.DefaultPrintQueue;

    // Get an XpsDocumentWriter for the default print queue
    XpsDocumentWriter xpsdw = PrintQueue.CreateXpsDocumentWriter(pq);
    return xpsdw;
}// end:GetPrintXpsDocumentWriter()

```

```

' ----- GetPrintXpsDocumentWriter() -----
''' <summary>
'''   Returns an XpsDocumentWriter for the default print queue.</summary>
''' <returns>
'''   An XpsDocumentWriter for the default print queue.</returns>
Private Function GetPrintXpsDocumentWriter() As XpsDocumentWriter
    ' Create a local print server
    Dim ps As New LocalPrintServer()

    ' Get the default print queue
    Dim pq As PrintQueue = ps.DefaultPrintQueue

    ' Get an XpsDocumentWriter for the default print queue
    Dim xpsdw As XpsDocumentWriter = PrintQueue.CreateXpsDocumentWriter(pq)
    Return xpsdw
End Function ' end:GetPrintXpsDocumentWriter()

```

The [AddJob](#) methods also provide ways to print. See [Programmatically Print XPS Files](#). for details.

## GDI Print Path

While WPF applications natively support the XPS print path, Win32 and Windows Forms applications can also take advantage of some XPS features. The XPS printer driver (XPSDrv) can convert GDI based output to XPS format. For advanced scenarios, custom conversion of content is supported using the [Microsoft XPS Document Converter \(MXDC\)](#). Similarly, WPF applications can also output to the GDI print path by calling one of the [Write](#) or [WriteAsync](#) methods of the [XpsDocumentWriter](#) class and designating a non-XpsDrv printer as the target print queue.

For applications that do not require XPS functionality or support, the current GDI print path remains unchanged.

- For additional reference material on the GDI print path and the various XPS conversion options, see [Microsoft XPS Document Converter \(MXDC\)](#) and [XPSDrv Printer Drivers](#).

## XPSDrv Driver Model

The XPS print path improves spooler efficiency by using XPS as the native print spool format when printing to an XPS -enabled printer or driver. The simplified spooling process eliminates the need to generate an intermediate spool file, such as an EMF data file, before the document is spooled. Through smaller spool file sizes, the XPS print path can reduce network traffic and improve print performance.

EMF is a closed format that represents application output as a series of calls into GDI for rendering services. Unlike EMF, the XPS spool format represents the actual document without requiring further interpretation when output to an XPS-based printer driver (XPSDrv). The drivers can operate directly on the data in the format. This capability eliminates the data and color space conversions required when you use EMF files and GDI-based print drivers.

Spool file sizes are usually reduced when you use XPS Documents that target an XPS printer driver (XPSDrv) compared with their EMF equivalents; however, there are exceptions:

- A vector graphic that is very complex, multi-layered, or inefficiently written can be larger than a bitmapped version of the same graphic.
- For screen display purposes, XPS files embed device fonts as well as computer-based fonts; whereas GDI spool files do not embed device fonts. But both kinds of fonts are subsetted (see below) and printer drivers can remove the device fonts before transmitting the file to the printer.

Spool size reduction is performed through several mechanisms:

- **Font subsetting.** Only characters used within the actual document are stored in the XPS file.

- **Advanced Graphics Support.** Native support for transparency and gradient primitives avoids rasterization of content in the XPS Document.
- **Identification of common resources.** Resources that are used multiple times (such as an image that represents a corporate logo) are treated as shared resources and are loaded only once.
- **ZIP compression.** All XPS documents use ZIP compression.

## See also

- [PrintDialog](#)
- [XpsDocumentWriter](#)
- [XpsDocument](#)
- [PrintTicket](#)
- [PrintCapabilities](#)
- [PrintServer](#)
- [PrintQueue](#)
- [How-to Topics](#)
- [Documents in WPF](#)
- [XPS Documents](#)
- [Document Serialization and Storage](#)
- [Microsoft XPS Document Converter \(MXDC\)](#)

# Printing How-to Topics

10/29/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section demonstrate how to use the printing and print system management features included with Windows Presentation Foundation (WPF) as well as the new XML Paper Specification (XPS) print path.

## In This Section

### [Invoke a Print Dialog](#)

Instructions for XAML markup to declare a Microsoft Windows print dialog object and using code to invoke the dialog from within a Windows Presentation Foundation (WPF) application.

### [Clone a Printer](#)

Instructions for how to install a second print queue with exactly the same properties as an existing print queue.

### [Diagnose Problematic Print Job](#)

Instructions for using the properties of print queues and print jobs to diagnose a print job that is not printing.

### [Discover Whether a Print Job Can Be Printed At This Time of Day](#)

Instructions for using the properties of print queues and print jobs to programmatically decide what times of day the job can be printed.

### [Enumerate a Subset of Print Queues](#)

Instructions for generating a list of printers having certain characteristics.

### [Get Print System Object Properties Without Reflection](#)

Instructions for how to discover at runtime print system object's properties and their types.

### [Programmatically Print XPS Files](#)

Instructions for rapid printing of XML Paper Specification (XPS) files without the need for a user interface (UI).

### [Remotely Survey the Status of Printers](#)

Instructions for creating a utility that will survey printers to discover those experiencing a paper jam or other problem.

### [Validate and Merge PrintTickets](#)

Instructions for checking that a print ticket is valid and that it does not request anything that is not supported by the printer.

## See also

- [System.Printing](#)
- [System.Printing.IndexedProperties](#)
- [System.Printing.Interop](#)
- [Printing Overview](#)
- [Documents in WPF](#)
- [XPS Documents](#)

# How to: Invoke a Print Dialog

9/4/2019 • 2 minutes to read • [Edit Online](#)

To provide the ability to print from your application, you can simply create and open a [PrintDialog](#) object.

## Example

The [PrintDialog](#) control provides a single entry point for UI, configuration, and XPS job submission. The control is easy to use and can be instantiated by using Extensible Application Markup Language (XAML) markup or code. The following example demonstrates how to instantiate and open the control in code and how to print from it. It also shows how to ensure that the dialog will give the user the option of setting a specific range of pages. The example code assumes that there is a file FixedDocumentSequence.xps in the root of the C: drive.

```
private void InvokePrint(object sender, RoutedEventArgs e)
{
    // Create the print dialog object and set options
    PrintDialog pDialog = new PrintDialog();
    pDialog.PageRangeSelection = PageRangeSelection.AllPages;
    pDialog.UserPageRangeEnabled = true;

    // Display the dialog. This returns true if the user presses the Print button.
    Nullable<Boolean> print = pDialog.ShowDialog();
    if (print == true)
    {
        XpsDocument xpsDocument = new XpsDocument("C:\\\\FixedDocumentSequence.xps", FileAccess.ReadWrite);
        FixedDocumentSequence fixedDocSeq = xpsDocument.GetFixedDocumentSequence();
        pDialog.PrintDocument(fixedDocSeq.DocumentPaginator, "Test print job");
    }
}
```

```
Private Sub InvokePrint(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Create the print dialog object and set options
    Dim pDialog As New PrintDialog()
    pDialog.PageRangeSelection = PageRangeSelection.AllPages
    pDialog.UserPageRangeEnabled = True

    ' Display the dialog. This returns true if the user presses the Print button.
    Dim print? As Boolean = pDialog.ShowDialog()
    If print = True Then
        Dim xpsDocument As New XpsDocument("C:\\\\FixedDocumentSequence.xps", FileAccess.ReadWrite)
        Dim fixedDocSeq As FixedDocumentSequence = xpsDocument.GetFixedDocumentSequence()
        pDialog.PrintDocument(fixedDocSeq.DocumentPaginator, "Test print job")
    End If
End Sub
```

Once the dialog is open, users will be able to select from the printers installed on their computer. They will also have the option of selecting the [Microsoft XPS Document Writer](#) to create an XML Paper Specification (XPS) file instead of printing.

### NOTE

The [System.Windows.Controls.PrintDialog](#) control of WPF, which is discussed in this topic, should not be confused with the [System.Windows.Forms.PrintDialog](#) component of Windows Forms.

Strictly speaking, you can use the [PrintDocument](#) method without ever opening the dialog. In that sense, the control can be used as an unseen printing component. But for performance reasons, it would be better to use either the [AddJob](#) method or one of the many [Write](#) and [WriteAsync](#) methods of the [XpsDocumentWriter](#). For more about this, see [Programmatically Print XPS Files](#) and .

## See also

- [PrintDialog](#)
- [Documents in WPF](#)
- [Printing Overview](#)
- [Microsoft XPS Document Writer](#)

# How to: Clone a Printer

4/28/2019 • 2 minutes to read • [Edit Online](#)

Most businesses will, at some point, buy multiple printers of the same model. Typically, these are all installed with virtually identical configuration settings. Installing each printer can be time-consuming and error prone. The [System.Printing.IndexedProperties](#) namespace and the [InstallPrintQueue](#) class that are exposed with Microsoft .NET Framework makes it possible to instantly install any number of additional print queues that are cloned from an existing print queue.

## Example

In the example below, a second print queue is cloned from an existing print queue. The second differs from the first only in its name, location, port, and shared status. The major steps for doing this are as follows.

1. Create a [PrintQueue](#) object for the existing printer that is going to be cloned.
2. Create a [PrintPropertyDictionary](#) from the [PropertiesCollection](#) of the [PrintQueue](#). The [Value](#) property of each entry in this dictionary is an object of one of the types derived from [PrintProperty](#). There are two ways to set the value of an entry in this dictionary.
  - Use the dictionary's [Remove](#) and [Add](#) methods to remove the entry and then re-add it with the desired value.
  - Use the dictionary's  [SetProperty](#) method.

The example below illustrates both ways.

3. Create a [PrintBooleanProperty](#) object and set its [Name](#) to "IsShared" and its [Value](#) to `true`.
4. Use the [PrintBooleanProperty](#) object to be the value of the [PrintPropertyDictionary](#)'s "IsShared" entry.
5. Create a [PrintStringProperty](#) object and set its [Name](#) to "ShareName" and its [Value](#) to an appropriate [String](#).
6. Use the [PrintStringProperty](#) object to be the value of the [PrintPropertyDictionary](#)'s "ShareName" entry.
7. Create another [PrintStringProperty](#) object and set its [Name](#) to "Location" and its [Value](#) to an appropriate [String](#).
8. Use the second [PrintStringProperty](#) object to be the value of the [PrintPropertyDictionary](#)'s "Location" entry.
9. Create an array of [Strings](#). Each item is the name of a port on the server.
10. Use [InstallPrintQueue](#) to install the new printer with the new values.

An example is below.

```
LocalPrintServer myLocalPrintServer = new LocalPrintServer(PrintSystemDesiredAccess.AdministrateServer);
PrintQueue sourcePrintQueue = myLocalPrintServer.DefaultPrintQueue;
PrintPropertyDictionary myPrintProperties = sourcePrintQueue.PropertiesCollection;

// Share the new printer using Remove/Add methods
PrintBooleanProperty shared = new PrintBooleanProperty("IsShared", true);
myPrintProperties.Remove("IsShared");
myPrintProperties.Add("IsShared", shared);

// Give the new printer its share name using SetProperty method
PrintStringProperty theShareName = new PrintStringProperty("ShareName", "\"Son of " + sourcePrintQueue.Name
+ "\"");
myPrintProperties SetProperty("ShareName", theShareName);

// Specify the physical location of the new printer using Remove/Add methods
PrintStringProperty theLocation = new PrintStringProperty("Location", "the supply room");
myPrintProperties.Remove("Location");
myPrintProperties.Add("Location", theLocation);

// Specify the port for the new printer
String[] port = new String[] { "COM1:" };

// Install the new printer on the local print server
PrintQueue clonedPrinter = myLocalPrintServer.InstallPrintQueue("My clone of " + sourcePrintQueue.Name, "Xerox
WCP 35 PS", port, "WinPrint", myPrintProperties);
myLocalPrintServer.Commit();

// Report outcome
Console.WriteLine("{0} in {1} has been installed and shared as {2}", clonedPrinter.Name,
clonedPrinter.Location, clonedPrinter.ShareName);
Console.WriteLine("Press Return to continue ...");
Console.ReadLine();
```

```

Dim myLocalPrintServer As New LocalPrintServer(PrintSystemDesiredAccess.AdministrateServer)
Dim sourcePrintQueue As PrintQueue = myLocalPrintServer.DefaultPrintQueue
Dim myPrintProperties As PrintPropertyDictionary = sourcePrintQueue.PropertiesCollection

' Share the new printer using Remove/Add methods
Dim [shared] As New PrintBooleanProperty("IsShared", True)
myPrintProperties.Remove("IsShared")
myPrintProperties.Add("IsShared", [shared])

' Give the new printer its share name using SetProperty method
Dim theShareName As New PrintStringProperty("ShareName", """Son of " & sourcePrintQueue.Name & """")
myPrintProperties SetProperty("ShareName", theShareName)

' Specify the physical location of the new printer using Remove/Add methods
Dim theLocation As New PrintStringProperty("Location", "the supply room")
myPrintProperties.Remove("Location")
myPrintProperties.Add("Location", theLocation)

' Specify the port for the new printer
Dim port() As String = { "COM1:" }

' Install the new printer on the local print server
Dim clonedPrinter As PrintQueue = myLocalPrintServer.InstallPrintQueue("My clone of " & sourcePrintQueue.Name,
"Xerox WCP 35 PS", port, "WinPrint", myPrintProperties)
myLocalPrintServer.Commit()

' Report outcome
Console.WriteLine("{0} in {1} has been installed and shared as {2}", clonedPrinter.Name,
clonedPrinter.Location, clonedPrinter.ShareName)
Console.WriteLine("Press Return to continue ...")
Console.ReadLine()

```

## See also

- [System.Printing.IndexedProperties](#)
- [PrintPropertyDictionary](#)
- [LocalPrintServer](#)
- [PrintQueue](#)
- [DictionaryEntry](#)
- [Documents in WPF](#)
- [Printing Overview](#)

# How to: Diagnose Problematic Print Job

8/22/2019 • 15 minutes to read • [Edit Online](#)

Network administrators often field complaints from users about print jobs that do not print or print slowly. The rich set of print job properties exposed in the APIs of Microsoft .NET Framework provide a means for performing a rapid remote diagnosis of print jobs.

## Example

The major steps for creating this kind of utility are as follows.

1. Identify the print job that the user is complaining about. Users often cannot do this precisely. They may not know the names of the print servers or printers. They may describe the location of the printer in different terminology than was used in setting its [Location](#) property. Accordingly, it is a good idea to generate a list of the user's currently submitted jobs. If there is more than one, then communication between the user and the print system administrator can be used to pinpoint the job that is having problems. The substeps are as follows.
  - a. Obtain a list of all print servers.
  - b. Loop through the servers to query their print queues.
  - c. Within each pass of the server loop, loop through all the server's queues to query their jobs
  - d. Within each pass of the queue loop, loop through its jobs and gather identifying information about those that were submitted by the complaining user.
2. When the problematic print job has been identified, examine relevant properties to see what might be the problem. For example, is job in an error state or did the printer servicing the queue go offline before the job could print?

The code below is series of code examples. The first code example contains the loop through the print queues. (Step 1c above.) The variable `myPrintQueues` is the [PrintQueueCollection](#) object for the current print server.

The code example begins by refreshing the current print queue object with [PrintQueue.Refresh](#). This ensures that the object's properties accurately represent the state of the physical printer that it represents. Then the application gets the collection of print jobs currently in the print queue by using [GetPrintJobInfoCollection](#).

Next the application loops through the [PrintSystemJobInfo](#) collection and compares each [Submitter](#) property with the alias of the complaining user. If they match, the application adds identifying information about the job to the string that will be presented. (The `userName` and `jobList` variables are initialized earlier in the application.)

```

for each (PrintQueue^ pq in myPrintQueues)
{
    pq->Refresh();
    PrintJobInfoCollection^ jobs = pq->GetPrintJobInfoCollection();
    for each (PrintSystemJobInfo^ job in jobs)
    {
        // Since the user may not be able to articulate which job is problematic,
        // present information about each job the user has submitted.
        if (job->Submitter == userName)
        {
            atLeastOne = true;
            jobList = jobList + "\nServer:" + line;
            jobList = jobList + "\n\tQueue:" + pq->Name;
            jobList = jobList + "\n\tLocation:" + pq->Location;
            jobList = jobList + "\n\t\tJob: " + job->JobName + " ID: " + job->JobIdentifier;
        }
    }
}

```

```

foreach (PrintQueue pq in myPrintQueues)
{
    pq.Refresh();
    PrintJobInfoCollection jobs = pq.GetPrintJobInfoCollection();
    foreach (PrintSystemJobInfo job in jobs)
    {
        // Since the user may not be able to articulate which job is problematic,
        // present information about each job the user has submitted.
        if (job.Submitter == userName)
        {
            atLeastOne = true;
            jobList = jobList + "\nServer:" + line;
            jobList = jobList + "\n\tQueue:" + pq.Name;
            jobList = jobList + "\n\tLocation:" + pq.Location;
            jobList = jobList + "\n\t\tJob: " + job.JobName + " ID: " + job.JobIdentifier;
        }
    }
    // end for each print job
}

// end for each print queue

```

```

For Each pq As PrintQueue In myPrintQueues
    pq.Refresh()
    Dim jobs As PrintJobInfoCollection = pq.GetPrintJobInfoCollection()
    For Each job As PrintSystemJobInfo In jobs
        ' Since the user may not be able to articulate which job is problematic,
        ' present information about each job the user has submitted.
        If job.Submitter = userName Then
            atLeastOne = True
            jobList = jobList & vbCrLf & "Server:" & line
            jobList = jobList & vbCrLf & vbTab & "Queue:" & pq.Name
            jobList = jobList & vbCrLf & vbTab & "Location:" & pq.Location
            jobList = jobList & vbCrLf & vbTab & vbTab & "Job: " & job.JobName & " ID: " & job.JobIdentifier
        End If
        Next job ' end for each print job

    Next pq ' end for each print queue

```

The next code example picks up the application at Step 2. (See above.) The problematic job has been identified and the application prompts for the information that will identify it. From this information it creates [PrintServer](#), [PrintQueue](#), and [PrintSystemJobInfo](#) objects.

At this point the application contains a branching structure corresponding to the two ways of checking a print job's status:

- You can read the flags of the [JobStatus](#) property which is of type [PrintJobStatus](#).
- You can read each relevant property such as [IsBlocked](#) and [IsInError](#).

This example demonstrates both methods, so the user was previously prompted as to which method to use and responded with "Y" if he or she wanted to use the flags of the [JobStatus](#) property. See below for the details of the two methods. Finally, the application uses a method called [ReportQueueAndJobAvailability](#) to report on whether the job can be printed at this time of day. This method is discussed in [Discover Whether a Print Job Can Be Printed At This Time of Day](#).

```
// When the problematic print job has been identified, enter information about it.
Console::Write("\nEnter the print server hosting the job (including leading slashes \\\\): " + "\n(press
Return for the current computer \\\{0\}): ", Environment::MachineName);
String^ pServer = Console::ReadLine();
if (pServer == "")
{
    pServer = "\\\\" + Environment::MachineName;
}
Console::Write("\nEnter the print queue hosting the job: ");
String^ pQueue = Console::ReadLine();
Console::Write("\nEnter the job ID: ");
Int16 jobID = Convert::ToInt16(Console::ReadLine());

// Create objects to represent the server, queue, and print job.
PrintServer^ hostingServer = gcnew PrintServer(pServer, PrintSystemDesiredAccess::AdministrateServer);
PrintQueue^ hostingQueue = gcnew PrintQueue(hostingServer, pQueue,
PrintSystemDesiredAccess::AdministratePrinter);
PrintSystemJobInfo^ theJob = hostingQueue->GetJob(jobID);

if (useAttributesResponse == "Y")
{
    TroubleSpotter::SpotTroubleUsingJobAttributes(theJob);
    // TroubleSpotter class is defined in the complete example.
} else
{
    TroubleSpotter::SpotTroubleUsingProperties(theJob);
}

TroubleSpotter::ReportQueueAndJobAvailability(theJob);
```

```

// When the problematic print job has been identified, enter information about it.
Console.WriteLine("\nEnter the print server hosting the job (including leading slashes \\\\"": " +
"\n(press Return for the current computer \\{0}): ", Environment.MachineName);
String pServer = Console.ReadLine();
if (pServer == "")
{
    pServer = "\\\\" + Environment.MachineName;
}
Console.WriteLine("\nEnter the print queue hosting the job: ");
String pQueue = Console.ReadLine();
Console.WriteLine("\nEnter the job ID: ");
Int16 jobID = Convert.ToInt16(Console.ReadLine());

// Create objects to represent the server, queue, and print job.
PrintServer hostingServer = new PrintServer(pServer, PrintSystemDesiredAccess.AdministrateServer);
PrintQueue hostingQueue = new PrintQueue(hostingServer, pQueue, PrintSystemDesiredAccess.AdministratePrinter);
PrintSystemJobInfo theJob = hostingQueue.GetJob(jobID);

if (useAttributesResponse == "Y")
{
    TroubleSpotter.SpotTroubleUsingJobAttributes(theJob);
    // TroubleSpotter class is defined in the complete example.
}
else
{
    TroubleSpotter.SpotTroubleUsingProperties(theJob);
}

TroubleSpotter.ReportQueueAndJobAvailability(theJob);

```

```

' When the problematic print job has been identified, enter information about it.
Console.WriteLine(vbLf & "Enter the print server hosting the job (including leading slashes \\): " & vbCrLf & "
(press Return for the current computer \\{0}): ", Environment.MachineName)
Dim pServer As String = Console.ReadLine()
If pServer = "" Then
    pServer = "\\" & Environment.MachineName
End If
Console.WriteLine(vbLf & "Enter the print queue hosting the job: ")
Dim pQueue As String = Console.ReadLine()
Console.WriteLine(vbLf & "Enter the job ID: ")
Dim jobID As Int16 = Convert.ToInt16(Console.ReadLine())

' Create objects to represent the server, queue, and print job.
Dim hostingServer As New PrintServer(pServer, PrintSystemDesiredAccess.AdministrateServer)
Dim hostingQueue As New PrintQueue(hostingServer, pQueue, PrintSystemDesiredAccess.AdministratePrinter)
Dim theJob As PrintSystemJobInfo = hostingQueue.GetJob(jobID)

If useAttributesResponse = "Y" Then
    TroubleSpotter.SpotTroubleUsingJobAttributes(theJob)
    ' TroubleSpotter class is defined in the complete example.
Else
    TroubleSpotter.SpotTroubleUsingProperties(theJob)
End If

TroubleSpotter.ReportQueueAndJobAvailability(theJob)

```

To check print job status using the flags of the [JobStatus](#) property, you check each relevant flag to see if it is set. The standard way to see if one bit is set in a set of bit flags is to perform a logical AND operation with the set of flags as one operand and the flag itself as the other. Since the flag itself has only one bit set, the result of the logical AND is that, at most, that same bit is set. To find out whether it is or not, just compare the result of the logical AND with the flag itself. For more information, see [PrintJobStatus](#), the [& Operator \(C# Reference\)](#), and [FlagsAttribute](#).

For each attribute whose bit is set, the code reports this to the console screen and sometimes suggests a way to

respond. (The **HandlePausedJob** method that is called if the job or queue is paused is discussed below.)

```
// Check for possible trouble states of a print job using the flags of the JobStatus property
static void SpotTroubleUsingJobAttributes (PrintSystemJobInfo^ theJob)
{
    if (((theJob->JobStatus & PrintJobStatus::Blocked) == PrintJobStatus::Blocked)
    {
        Console::WriteLine("The job is blocked.");
    }
    if (((theJob->JobStatus & PrintJobStatus::Completed) == PrintJobStatus::Completed)
        ||
        ((theJob->JobStatus & PrintJobStatus::Printed) == PrintJobStatus::Printed))
    {
        Console::WriteLine("The job has finished. Have user recheck all output bins and be sure the correct
printer is being checked.");
    }
    if (((theJob->JobStatus & PrintJobStatus::Deleted) == PrintJobStatus::Deleted)
        ||
        ((theJob->JobStatus & PrintJobStatus::Deleting) == PrintJobStatus::Deleting))
    {
        Console::WriteLine("The user or someone with administration rights to the queue has deleted the job. It
must be resubmitted.");
    }
    if ((theJob->JobStatus & PrintJobStatus::Error) == PrintJobStatus::Error)
    {
        Console::WriteLine("The job has errored.");
    }
    if ((theJob->JobStatus & PrintJobStatus::Offline) == PrintJobStatus::Offline)
    {
        Console::WriteLine("The printer is offline. Have user put it online with printer front panel.");
    }
    if ((theJob->JobStatus & PrintJobStatus::PaperOut) == PrintJobStatus::PaperOut)
    {
        Console::WriteLine("The printer is out of paper of the size required by the job. Have user add paper.");
    }
    if (((theJob->JobStatus & PrintJobStatus::Paused) == PrintJobStatus::Paused)
        ||
        ((theJob->HostingPrintQueue->QueueStatus & PrintQueueStatus::Paused) == PrintQueueStatus::Paused))
    {
        HandlePausedJob(theJob);
        //HandlePausedJob is defined in the complete example.
    }

    if ((theJob->JobStatus & PrintJobStatus::Printing) == PrintJobStatus::Printing)
    {
        Console::WriteLine("The job is printing now.");
    }
    if ((theJob->JobStatus & PrintJobStatus::Spooling) == PrintJobStatus::Spooling)
    {
        Console::WriteLine("The job is spooling now.");
    }
    if ((theJob->JobStatus & PrintJobStatus::UserIntervention) == PrintJobStatus::UserIntervention)
    {
        Console::WriteLine("The printer needs human intervention.");
    }
};

};
```

```

// Check for possible trouble states of a print job using the flags of the JobStatus property
internal static void SpotTroubleUsingJobAttributes(PrintSystemJobInfo theJob)
{
    if (((theJob.JobStatus & PrintJobStatus.Blocked) == PrintJobStatus.Blocked)
    {
        Console.WriteLine("The job is blocked.");
    }
    if (((theJob.JobStatus & PrintJobStatus.Completed) == PrintJobStatus.Completed)
        ||
        ((theJob.JobStatus & PrintJobStatus.Printed) == PrintJobStatus.Printed))
    {
        Console.WriteLine("The job has finished. Have user recheck all output bins and be sure the correct
printer is being checked.");
    }
    if (((theJob.JobStatus & PrintJobStatus.Deleted) == PrintJobStatus.Deleted)
        ||
        ((theJob.JobStatus & PrintJobStatus.Deleting) == PrintJobStatus.Deleting))
    {
        Console.WriteLine("The user or someone with administration rights to the queue has deleted the job. It
must be resubmitted.");
    }
    if ((theJob.JobStatus & PrintJobStatus.Error) == PrintJobStatus.Error)
    {
        Console.WriteLine("The job has errored.");
    }
    if ((theJob.JobStatus & PrintJobStatus.Offline) == PrintJobStatus.Offline)
    {
        Console.WriteLine("The printer is offline. Have user put it online with printer front panel.");
    }
    if ((theJob.JobStatus & PrintJobStatus.PaperOut) == PrintJobStatus.PaperOut)
    {
        Console.WriteLine("The printer is out of paper of the size required by the job. Have user add
paper.");
    }

    if (((theJob.JobStatus & PrintJobStatus.Paused) == PrintJobStatus.Paused)
        ||
        ((theJob.HostingPrintQueue.QueueStatus & PrintQueueStatus.Paused) == PrintQueueStatus.Paused))
    {
        HandlePausedJob(theJob);
        //HandlePausedJob is defined in the complete example.
    }

    if ((theJob.JobStatus & PrintJobStatus.Printing) == PrintJobStatus.Printing)
    {
        Console.WriteLine("The job is printing now.");
    }
    if ((theJob.JobStatus & PrintJobStatus.Spooling) == PrintJobStatus.Spooling)
    {
        Console.WriteLine("The job is spooling now.");
    }
    if ((theJob.JobStatus & PrintJobStatus.UserIntervention) == PrintJobStatus.UserIntervention)
    {
        Console.WriteLine("The printer needs human intervention.");
    }
}

//end SpotTroubleUsingJobAttributes

```

```

' Check for possible trouble states of a print job using the flags of the JobStatus property
Friend Shared Sub SpotTroubleUsingJobAttributes(ByVal theJob As PrintSystemJobInfo)
    If (theJob.JobStatus And PrintJobStatus.Blocked) = PrintJobStatus.Blocked Then
        Console.WriteLine("The job is blocked.")
    End If
    If ((theJob.JobStatus And PrintJobStatus.Completed) = PrintJobStatus.Completed) OrElse ((theJob.JobStatus
And PrintJobStatus.Printed) = PrintJobStatus.Printed) Then
        Console.WriteLine("The job has finished. Have user recheck all output bins and be sure the correct
printer is being checked.")
    End If
    If ((theJob.JobStatus And PrintJobStatus.Deleted) = PrintJobStatus.Deleted) OrElse ((theJob.JobStatus And
PrintJobStatus.Deleting) = PrintJobStatus.Deleting) Then
        Console.WriteLine("The user or someone with administration rights to the queue has deleted the job. It
must be resubmitted.")
    End If
    If (theJob.JobStatus And PrintJobStatus.Error) = PrintJobStatus.Error Then
        Console.WriteLine("The job has errored.")
    End If
    If (theJob.JobStatus And PrintJobStatus.Offline) = PrintJobStatus.Offline Then
        Console.WriteLine("The printer is offline. Have user put it online with printer front panel.")
    End If
    If (theJob.JobStatus And PrintJobStatus.PaperOut) = PrintJobStatus.PaperOut Then
        Console.WriteLine("The printer is out of paper or the size required by the job. Have user add paper.")
    End If

    If ((theJob.JobStatus And PrintJobStatus.Paused) = PrintJobStatus.Paused) OrElse
((theJob.HostingPrintQueue.QueueStatus And PrintQueueStatus.Paused) = PrintQueueStatus.Paused) Then
        HandlePausedJob(theJob)
        'HandlePausedJob is defined in the complete example.
    End If

    If (theJob.JobStatus And PrintJobStatus.Printing) = PrintJobStatus.Printing Then
        Console.WriteLine("The job is printing now.")
    End If
    If (theJob.JobStatus And PrintJobStatus.Spooling) = PrintJobStatus.Spooling Then
        Console.WriteLine("The job is spooling now.")
    End If
    If (theJob.JobStatus And PrintJobStatus.UserIntervention) = PrintJobStatus.UserIntervention Then
        Console.WriteLine("The printer needs human intervention.")
    End If

End Sub

```

To check print job status using separate properties, you simply read each property and, if the property is `true`, report to the console screen and possibly suggest a way to respond. (The **HandlePausedJob** method that is called if the job or queue is paused is discussed below.)

```
// Check for possible trouble states of a print job using its properties
static void SpotTroubleUsingProperties (PrintSystemJobInfo^ theJob)
{
    if (theJob->IsBlocked)
    {
        Console::WriteLine("The job is blocked.");
    }
    if (theJob->IsCompleted || theJob->IsPrinted)
    {
        Console::WriteLine("The job has finished. Have user recheck all output bins and be sure the correct
printer is being checked.");
    }
    if (theJob->IsDeleted || theJob->IsDeleting)
    {
        Console::WriteLine("The user or someone with administration rights to the queue has deleted the job. It
must be resubmitted.");
    }
    if (theJob->IsInError)
    {
        Console::WriteLine("The job has errored.");
    }
    if (theJob->IsOffline)
    {
        Console::WriteLine("The printer is offline. Have user put it online with printer front panel.");
    }
    if (theJob->IsPaperOut)
    {
        Console::WriteLine("The printer is out of paper of the size required by the job. Have user add paper.");
    }

    if (theJob->IsPaused || theJob->HostingPrintQueue->IsPaused)
    {
        HandlePausedJob(theJob);
        //HandlePausedJob is defined in the complete example.
    }

    if (theJob->IsPrinting)
    {
        Console::WriteLine("The job is printing now.");
    }
    if (theJob->IsSpooling)
    {
        Console::WriteLine("The job is spooling now.");
    }
    if (theJob->IsUserInterventionRequired)
    {
        Console::WriteLine("The printer needs human intervention.");
    }
};
```

```
// Check for possible trouble states of a print job using its properties
internal static void SpotTroubleUsingProperties(PrintSystemJobInfo theJob)
{
    if (theJob.IsBlocked)
    {
        Console.WriteLine("The job is blocked.");
    }
    if (theJob.IsCompleted || theJob.IsPrinted)
    {
        Console.WriteLine("The job has finished. Have user recheck all output bins and be sure the
correct printer is being checked.");
    }
    if (theJob.IsDeleted || theJob.IsDeleting)
    {
        Console.WriteLine("The user or someone with administration rights to the queue has deleted the
job. It must be resubmitted.");
    }
    if (theJob.IsInError)
    {
        Console.WriteLine("The job has errored.");
    }
    if (theJob.Offline)
    {
        Console.WriteLine("The printer is offline. Have user put it online with printer front panel.");
    }
    if (theJob.IsPaperOut)
    {
        Console.WriteLine("The printer is out of paper of the size required by the job. Have user add
paper.");
    }

    if (theJob.IsPaused || theJob.HostingPrintQueue.IsPaused)
    {
        HandlePausedJob(theJob);
        //HandlePausedJob is defined in the complete example.
    }

    if (theJob.IsPrinting)
    {
        Console.WriteLine("The job is printing now.");
    }
    if (theJob.IsSpooling)
    {
        Console.WriteLine("The job is spooling now.");
    }
    if (theJob.isUserInterventionRequired)
    {
        Console.WriteLine("The printer needs human intervention.");
    }
}

}//end SpotTroubleUsingProperties
```

```

' Check for possible trouble states of a print job using its properties
Friend Shared Sub SpotTroubleUsingProperties(ByVal theJob As PrintSystemJobInfo)
    If theJob.IsBlocked Then
        Console.WriteLine("The job is blocked.")
    End If
    If theJob.IsCompleted OrElse theJob.IsPrinted Then
        Console.WriteLine("The job has finished. Have user recheck all output bins and be sure the correct
printer is being checked.")
    End If
    If theJob.IsDeleted OrElse theJob.IsDeleting Then
        Console.WriteLine("The user or someone with administration rights to the queue has deleted the job. It
must be resubmitted.")
    End If
    If theJob.IsInError Then
        Console.WriteLine("The job has errored.")
    End If
    If theJob.IsOffline Then
        Console.WriteLine("The printer is offline. Have user put it online with printer front panel.")
    End If
    If theJob.IsPaperOut Then
        Console.WriteLine("The printer is out of paper of the size required by the job. Have user add paper.")
    End If

    If theJob.IsPaused OrElse theJob.HostingPrintQueue.IsPaused Then
        HandlePausedJob(theJob)
        'HandlePausedJob is defined in the complete example.
    End If

    If theJob.IsPrinting Then
        Console.WriteLine("The job is printing now.")
    End If
    If theJob.IsSpooling Then
        Console.WriteLine("The job is spooling now.")
    End If
    If theJob.IsUserInterventionRequired Then
        Console.WriteLine("The printer needs human intervention.")
    End If

End Sub

```

The **HandlePausedJob** method enables the application's user to remotely resume paused jobs. Because there might be a good reason why the print queue was paused, the method begins by prompting for a user decision about whether to resume it. If the answer is "Y", then the [PrintQueue.Resume](#) method is called.

Next the user is prompted to decide if the job itself should be resumed, just in case it is paused independently of the print queue. (Compare [PrintQueue.IsPaused](#) and [PrintSystemJobInfo.IsPaused](#).) If the answer is "Y", then [PrintSystemJobInfo.Resume](#) is called; otherwise [Cancel](#) is called.

```

static void HandlePausedJob (PrintSystemJobInfo^ theJob)
{
    // If there's no good reason for the queue to be paused, resume it and
    // give user choice to resume or cancel the job.
    Console::WriteLine("The user or someone with administrative rights to the queue" + "\nhas paused the job or
queue." + "\nResume the queue? (Has no effect if queue is not paused.)" + "\nEnter \"Y\" to resume, otherwise
press return: ");
    String^ resume = Console::ReadLine();
    if (resume == "Y")
    {
        theJob->HostingPrintQueue->Resume();

        // It is possible the job is also paused. Find out how the user wants to handle that.
        Console::WriteLine("Does user want to resume print job or cancel it?" + "\nEnter \"Y\" to resume (any
other key cancels the print job): ");
        String^ userDecision = Console::ReadLine();
        if (userDecision == "Y")
        {
            theJob->Resume();
        } else
        {
            theJob->Cancel();
        }
    }
};


```

```

internal static void HandlePausedJob(PrintSystemJobInfo theJob)
{
    // If there's no good reason for the queue to be paused, resume it and
    // give user choice to resume or cancel the job.
    Console.WriteLine("The user or someone with administrative rights to the queue" +
        "\nhas paused the job or queue." +
        "\nResume the queue? (Has no effect if queue is not paused.)" +
        "\nEnter \"Y\" to resume, otherwise press return: ");
    String resume = Console.ReadLine();
    if (resume == "Y")
    {
        theJob.HostingPrintQueue.Resume();

        // It is possible the job is also paused. Find out how the user wants to handle that.
        Console.WriteLine("Does user want to resume print job or cancel it?" +
            "\nEnter \"Y\" to resume (any other key cancels the print job): ");
        String userDecision = Console.ReadLine();
        if (userDecision == "Y")
        {
            theJob.Resume();
        } else
        {
            theJob.Cancel();
        }
    }
//end if the queue should be resumed

}//end HandlePausedJob

```

```
Friend Shared Sub HandlePausedJob(ByVal theJob As PrintSystemJobInfo)
    ' If there's no good reason for the queue to be paused, resume it and
    ' give user choice to resume or cancel the job.
    Console.WriteLine("The user or someone with administrative rights to the queue" & vbCrLf & "has paused the
job or queue." & vbCrLf & "Resume the queue? (Has no effect if queue is not paused.)" & vbCrLf & "Enter ""Y"" to
resume, otherwise press return: ")
    Dim [resume] As String = Console.ReadLine()
    If [resume] = "Y" Then
        theJob.HostingPrintQueue.Resume()

        ' It is possible the job is also paused. Find out how the user wants to handle that.
        Console.WriteLine("Does user want to resume print job or cancel it?" & vbCrLf & "Enter ""Y"" to resume
(any other key cancels the print job): ")
        Dim userDecision As String = Console.ReadLine()
        If userDecision = "Y" Then
            theJob.Resume()
        Else
            theJob.Cancel()
        End If
    End If 'end if the queue should be resumed

End Sub
```

## See also

- [PrintJobStatus](#)
- [PrintSystemJobInfo](#)
- [FlagsAttribute](#)
- [PrintQueue](#)
- [& Operator \(C# Reference\)](#)
- [Documents in WPF](#)
- [Printing Overview](#)

# How to: Discover Whether a Print Job Can Be Printed At This Time of Day

8/22/2019 • 8 minutes to read • [Edit Online](#)

Print queues are not always available for 24 hours a day. They have start and end time properties that can be set to make them unavailable at certain times of day. This feature can be used, for example, to reserve a printer for the exclusive use of a certain department after 5 P.M.. That department would have a different queue servicing the printer than other departments use. The queue for the other departments would be set to be unavailable after 5 P.M., while queue for the favored department could be set to be available at all times.

Moreover, print jobs themselves can be set to be printable only within a specified span of time.

The [PrintQueue](#) and [PrintSystemJobInfo](#) classes exposed in the APIs of Microsoft .NET Framework provide a means for remotely checking whether a given print job can print on a given queue at the current time.

## Example

The example below is a sample that can diagnose problems with a print job.

There are two major steps for this kind of function as follows.

1. Read the [StartTimeOfDay](#) and [UntilTimeOfDay](#) properties of the [PrintQueue](#) to determine whether the current time is between them.
2. Read the [StartTimeOfDay](#) and [UntilTimeOfDay](#) properties of the [PrintSystemJobInfo](#) to determine whether the current time is between them.

But complications arise from the fact that these properties are not [DateTime](#) objects. Instead they are [Int32](#) objects that express the time of day as the number of minutes since midnight. Moreover, this is not midnight in the current time zone, but midnight UTC (Coordinated Universal Time).

The first code example presents the static method [ReportQueueAndJobAvailability](#), which is passed a [PrintSystemJobInfo](#) and calls helper methods to determine whether the job can print at the current time and, if not, when it can print. Notice that a [PrintQueue](#) is not passed to the method. This is because the [PrintSystemJobInfo](#) includes a reference to the queue in its [HostingPrintQueue](#) property.

The subordinate methods include the overloaded [ReportAvailabilityAtThisTime](#) method which can take either a [PrintQueue](#) or a [PrintSystemJobInfo](#) as a parameter. There is also a [TimeConverter.ConvertToLocalHumanReadableTime](#). All of these methods are discussed below.

The [ReportQueueAndJobAvailability](#) method begins by checking to see if either the queue or the print job is unavailable at this time. If either of them is unavailable, it then checks to see if the queue unavailable. If it is not available, then the method reports this fact and the time when the queue will become available again. It then checks the job and if it is unavailable, it reports the next time span when it can print. Finally, the method reports the earliest time when the job can print. This is the later of following two times.

- The time when the print queue is next available.
- The time when the print job is next available.

When reporting times of day, the [ToShortTimeString](#) method is also called because this method suppresses the years, months, and days from the output. You cannot restrict the availability of either a print queue or a print job to particular years, months, or days.

```
static void ReportQueueAndJobAvailability (PrintSystemJobInfo^ theJob)
{
    if (!(ReportAvailabilityAtThisTime(theJob->HostingPrintQueue) && ReportAvailabilityAtThisTime(theJob)))
    {
        if (!ReportAvailabilityAtThisTime(theJob->HostingPrintQueue))
        {
            Console::WriteLine("\nThat queue is not available at this time of day." + "\nJobs in the queue will
start printing again at {0}", TimeConverter::ConvertToLocalHumanReadableTime(theJob->HostingPrintQueue-
>StartTimeOfDay).ToShortTimeString());
            // TimeConverter class is defined in the complete sample
        }
        if (!ReportAvailabilityAtThisTime(theJob))
        {
            Console::WriteLine("\nThat job is set to print only between {0} and {1}",
TimeConverter::ConvertToLocalHumanReadableTime(theJob->StartTimeOfDay).ToShortTimeString(),
TimeConverter::ConvertToLocalHumanReadableTime(theJob->UntilTimeOfDay).ToShortTimeString());
        }
        Console::WriteLine("\nThe job will begin printing as soon as it reaches the top of the queue after:");
        if (theJob->StartTimeOfDay > theJob->HostingPrintQueue->StartTimeOfDay)
        {
            Console::WriteLine(TimeConverter::ConvertToLocalHumanReadableTime(theJob-
>StartTimeOfDay).ToShortTimeString());
        } else
        {
            Console::WriteLine(TimeConverter::ConvertToLocalHumanReadableTime(theJob->HostingPrintQueue-
>StartTimeOfDay).ToShortTimeString());
        }
    }
};
```

```

internal static void ReportQueueAndJobAvailability(PrintSystemJobInfo theJob)
{
    if (!(ReportAvailabilityAtThisTime(theJob.HostingPrintQueue) && ReportAvailabilityAtThisTime(theJob)))
    {
        if (!ReportAvailabilityAtThisTime(theJob.HostingPrintQueue))
        {
            Console.WriteLine("\nThat queue is not available at this time of day." +
                "\nJobs in the queue will start printing again at {0}",

TimeConverter.ConvertToLocalHumanReadableTime(theJob.HostingPrintQueue.StartTimeOfDay).ToShortTimeString());
            // TimeConverter class is defined in the complete sample
        }

        if (!ReportAvailabilityAtThisTime(theJob))
        {
            Console.WriteLine("\nThat job is set to print only between {0} and {1}",
                TimeConverter.ConvertToLocalHumanReadableTime(theJob.StartTimeOfDay).ToShortTimeString(),
                TimeConverter.ConvertToLocalHumanReadableTime(theJob.UntilTimeOfDay).ToShortTimeString());
        }
        Console.WriteLine("\nThe job will begin printing as soon as it reaches the top of the queue after:");
        if (theJob.StartTimeOfDay > theJob.HostingPrintQueue.StartTimeOfDay)
        {

Console.WriteLine(TimeConverter.ConvertToLocalHumanReadableTime(theJob.StartTimeOfDay).ToShortTimeString());
        }
        else
        {

Console.WriteLine(TimeConverter.ConvertToLocalHumanReadableTime(theJob.HostingPrintQueue.StartTimeOfDay).ToShortTimeString());
        }
    }
}

//end if at least one is not available

}//end ReportQueueAndJobAvailability

```

```

Friend Shared Sub ReportQueueAndJobAvailability(ByVal theJob As PrintSystemJobInfo)
    If Not(ReportAvailabilityAtThisTime(theJob.HostingPrintQueue) AndAlso
ReportAvailabilityAtThisTime(theJob)) Then
        If Not ReportAvailabilityAtThisTime(theJob.HostingPrintQueue) Then
            Console.WriteLine(vbLf & "That queue is not available at this time of day." & vbCrLf & "Jobs in the
queue will start printing again at {0}",

TimeConverter.ConvertToLocalHumanReadableTime(theJob.HostingPrintQueue.StartTimeOfDay).ToShortTimeString())
            ' TimeConverter class is defined in the complete sample
        End If

        If Not ReportAvailabilityAtThisTime(theJob) Then
            Console.WriteLine(vbLf & "That job is set to print only between {0} and {1}",
                TimeConverter.ConvertToLocalHumanReadableTime(theJob.StartTimeOfDay).ToShortTimeString(),
                TimeConverter.ConvertToLocalHumanReadableTime(theJob.UntilTimeOfDay).ToShortTimeString())
            End If
            Console.WriteLine(vbLf & "The job will begin printing as soon as it reaches the top of the queue
after:")
            If theJob.StartTimeOfDay > theJob.HostingPrintQueue.StartTimeOfDay Then

Console.WriteLine(TimeConverter.ConvertToLocalHumanReadableTime(theJob.StartTimeOfDay).ToShortTimeString())
        Else

Console.WriteLine(TimeConverter.ConvertToLocalHumanReadableTime(theJob.HostingPrintQueue.StartTimeOfDay).ToShortTimeString())
        End If
    End If 'end if at least one is not available

End Sub

```

The two overloads of the **ReportAvailabilityAtThisTime** method are identical except for the type passed to them, so only the [PrintQueue](#) version is presented below.

#### NOTE

The fact that the methods are identical except for type raises the question of why the sample does not create a generic method **ReportAvailabilityAtThisTime<T>**. The reason is that such a method would have to be restricted to a class that has the **StartTimeOfDay** and **UntilTimeOfDay** properties that the method calls, but a generic method can only be restricted to a single class and the only class common to both [PrintQueue](#) and [PrintSystemJobInfo](#) in the inheritance tree is [PrintSystemObject](#) which has no such properties.

The **ReportAvailabilityAtThisTime** method (presented in the code example below) begins by initializing a [Boolean](#) sentinel variable to `true`. It will be reset to `false`, if the queue is not available.

Next, the method checks to see if the start and "until" times are identical. If they are, the queue is always available, so the method returns `true`.

If the queue is not available all the time, the method uses the static [UtcNow](#) property to get the current time as a [DateTime](#) object. (We do not need local time because the **StartTimeOfDay** and **UntilTimeOfDay** properties are themselves in UTC time.)

However, these two properties are not [DateTime](#) objects. They are [Int32](#)s expressing the time as the number of minutes-after-UTC-midnight. So we do have to convert our [DateTime](#) object to minutes-after-midnight. When that is done, the method simply checks to see whether "now" is between the queue's start and "until" times, sets the sentinel to false if "now" is not between the two times, and returns the sentinel.

```
static Boolean ReportAvailabilityAtThisTime (PrintQueue^ pq)
{
    Boolean available = true;
    if (pq->StartTimeOfDay != pq->UntilTimeOfDay)
    {
        DateTime utcNow = DateTime::UtcNow;
        Int32 utcNowAsMinutesAfterMidnight = (utcNow.TimeOfDay.Hours * 60) + utcNow.TimeOfDay.Minutes;

        // If now is not within the range of available times . . .
        if (!((pq->StartTimeOfDay < utcNowAsMinutesAfterMidnight) && (utcNowAsMinutesAfterMidnight < pq->UntilTimeOfDay)))
        {
            available = false;
        }
    }
    return available;
};
```

```

private static Boolean ReportAvailabilityAtThisTime(PrintQueue pq)
{
    Boolean available = true;
    if (pq.StartTimeOfDay != pq.UntilTimeOfDay) // If the printer is not available 24 hours a day
    {
        DateTime utcNow = DateTime.UtcNow;
        Int32 utcNowAsMinutesAfterMidnight = (utcNow.TimeOfDay.Hours * 60) + utcNow.TimeOfDay.Minutes;

        // If now is not within the range of available times . . .
        if (!((pq.StartTimeOfDay < utcNowAsMinutesAfterMidnight)
            &&
            (utcNowAsMinutesAfterMidnight < pq.UntilTimeOfDay)))
        {
            available = false;
        }
    }
    return available;
}//end ReportAvailabilityAtThisTime

```

```

Private Shared Function ReportAvailabilityAtThisTime(ByVal pq As PrintQueue) As Boolean
    Dim available As Boolean = True
    If pq.StartTimeOfDay <> pq.UntilTimeOfDay Then ' If the printer is not available 24 hours a day
        Dim utcNow As Date = Date.UtcNow
        Dim utcNowAsMinutesAfterMidnight As Int32 = (utcNow.TimeOfDay.Hours * 60) + utcNow.TimeOfDay.Minutes

        ' If now is not within the range of available times . . .
        If Not((pq.StartTimeOfDay < utcNowAsMinutesAfterMidnight) AndAlso (utcNowAsMinutesAfterMidnight <
pq.UntilTimeOfDay)) Then
            available = False
        End If
    End If
    Return available
End Function 'end ReportAvailabilityAtThisTime

```

The **TimeConverter.ConvertToLocalHumanReadableTime** method (presented in the code example below) does not use any methods introduced with Microsoft .NET Framework, so the discussion is brief. The method has a double conversion task: it must take an integer expressing minutes-after-midnight and convert it to a human-readable time and it must convert this to the local time. It accomplishes this by first creating a [DateTime](#) object that is set to midnight UTC and then it uses the [AddMinutes](#) method to add the minutes that were passed to the method. This returns a new [DateTime](#) expressing the original time that was passed to the method. The [ToLocalTime](#) method then converts this to local time.

```

private ref class TimeConverter {

internal:
    static DateTime ConvertToLocalHumanReadableTime (Int32 timeInMinutesAfterUTCMidnight)
    {
        // Construct a UTC midnight object.
        // Must start with current date so that the local Daylight Savings system, if any, will be taken into
account.
        DateTime utcNow = DateTime::UtcNow;
        DateTime utcMidnight = DateTime(utcNow.Year, utcNow.Month, utcNow.Day, 0, 0, 0, DateTimeKind::Utc);

        // Add the minutes passed into the method in order to get the intended UTC time.
        Double minutesAfterUTCMidnight = ((Double)timeInMinutesAfterUTCMidnight);
        DateTime utcTime = utcMidnight.AddMinutes(minutesAfterUTCMidnight);

        // Convert to local time.
        DateTime localTime = utcTime.ToLocalTime();

        return localTime;
    };
}

```

```

class TimeConverter
{
    // Convert time as minutes past UTC midnight into human readable time in local time zone.
    internal static DateTime ConvertToLocalHumanReadableTime(Int32 timeInMinutesAfterUTCMidnight)
    {
        // Construct a UTC midnight object.
        // Must start with current date so that the local Daylight Savings system, if any, will be taken into
account.
        DateTime utcNow = DateTime.UtcNow;
        DateTime utcMidnight = new DateTime(utcNow.Year, utcNow.Month, utcNow.Day, 0, 0, 0, DateTimeKind.Utc);

        // Add the minutes passed into the method in order to get the intended UTC time.
        Double minutesAfterUTCMidnight = (Double)timeInMinutesAfterUTCMidnight;
        DateTime utcTime = utcMidnight.AddMinutes(minutesAfterUTCMidnight);

        // Convert to local time.
        DateTime localTime = utcTime.ToLocalTime();

        return localTime;
   }// end ConvertToLocalHumanReadableTime

}//end TimeConverter class

```

```

Friend Class TimeConverter
    ' Convert time as minutes past UTC midnight into human readable time in local time zone.
    Friend Shared Function ConvertToLocalHumanReadableTime(ByVal timeInMinutesAfterUTCMidnight As Int32) As
Date
        ' Construct a UTC midnight object.
        ' Must start with current date so that the local Daylight Savings system, if any, will be taken into
account.
        Dim utcNow As Date = Date.UtcNow
        Dim utcMidnight As New Date(utcNow.Year, utcNow.Month, utcNow.Day, 0, 0, 0, DateTimeKind.Utc)

        ' Add the minutes passed into the method in order to get the intended UTC time.
        Dim minutesAfterUTCMidnight As Double = CType(timeInMinutesAfterUTCMidnight, Double)
        Dim utcTime As Date = utcMidnight.AddMinutes(minutesAfterUTCMidnight)

        ' Convert to local time.
        Dim localTime As Date = utcTime.ToLocalTime()

        Return localTime

    End Function ' end ConvertToLocalHumanReadableTime

End Class

```

## See also

- [DateTime](#)
- [PrintSystemJobInfo](#)
- [PrintQueue](#)
- [Documents in WPF](#)
- [Printing Overview](#)

# How to: Enumerate a Subset of Print Queues

4/8/2019 • 2 minutes to read • [Edit Online](#)

A common situation faced by information technology (IT) professionals managing a company-wide set of printers is to generate a list of printers having certain characteristics. This functionality is provided by the [GetPrintQueues](#) method of a [PrintServer](#) object and the [EnumeratedPrintQueueTypes](#) enumeration.

## Example

In the example below, the code begins by creating an array of flags that specify the characteristics of the print queues we want to list. In this example, we are looking for print queues that are installed locally on the print server and are shared. The [EnumeratedPrintQueueTypes](#) enumeration provides many other possibilities.

The code then creates a [LocalPrintServer](#) object, a class derived from [PrintServer](#). The local print server is the computer on which the application is running.

The last significant step is to pass the array to the [GetPrintQueues](#) method.

Finally, the results are presented to the user.

```
// Specify that the list will contain only the print queues that are installed as local and are shared
array<System::Printing::EnumeratedPrintQueueTypes>^ enumerationFlags =
{EnumeratedPrintQueueTypes::Local, EnumeratedPrintQueueTypes::Shared};

LocalPrintServer^ printServer = gcnew LocalPrintServer();

//Use the enumerationFlags to filter out unwanted print queues
PrintQueueCollection^ printQueuesOnLocalServer = printServer->GetPrintQueues(enumerationFlags);

Console::WriteLine("These are your shared, local print queues:\n\n");

for each (PrintQueue^ printer in printQueuesOnLocalServer)
{
    Console::WriteLine("\tThe shared printer " + printer->Name + " is located at " + printer->Location + "\n");
}
Console::WriteLine("Press enter to continue.");
Console::ReadLine();
```

```
// Specify that the list will contain only the print queues that are installed as local and are shared
EnumeratedPrintQueueTypes[] enumerationFlags = {EnumeratedPrintQueueTypes.Local,
                                                EnumeratedPrintQueueTypes.Shared};

LocalPrintServer printServer = new LocalPrintServer();

//Use the enumerationFlags to filter out unwanted print queues
PrintQueueCollection printQueuesOnLocalServer = printServer.GetPrintQueues(enumerationFlags);

Console.WriteLine("These are your shared, local print queues:\n\n");

foreach (PrintQueue printer in printQueuesOnLocalServer)
{
    Console.WriteLine("\tThe shared printer " + printer.Name + " is located at " + printer.Location + "\n");
}
Console.WriteLine("Press enter to continue.");
Console.ReadLine();
```

```

' Specify that the list will contain only the print queues that are installed as local and are shared
Dim enumerationFlags() As EnumeratedPrintQueueTypes = {EnumeratedPrintQueueTypes.Local,
EnumeratedPrintQueueTypes.Shared}

Dim printServer As New LocalPrintServer()

'Use the enumerationFlags to filter out unwanted print queues
Dim printQueuesOnLocalServer As PrintQueueCollection = printServer.GetPrintQueues(enumerationFlags)

Console.WriteLine("These are your shared, local print queues:" & vbCrLf & vbCrLf)

For Each printer As PrintQueue In printQueuesOnLocalServer
    Console.WriteLine(vbTab & "The shared printer " & printer.Name & " is located at " & printer.Location &
vbLf)
Next printer
Console.WriteLine("Press enter to continue.")
Console.ReadLine()

```

You could extend this example by having the `foreach` loop that steps through each print queue do further screening. For example, you could screen out printers that do not support two-sided printing by having the loop call each print queue's [GetPrintCapabilities](#) method and test the returned value for the presence of duplexing.

## See also

- [GetPrintQueues](#)
- [PrintServer](#)
- [LocalPrintServer](#)
- [EnumeratedPrintQueueTypes](#)
- [PrintQueue](#)
- [GetPrintCapabilities](#)
- [Documents in WPF](#)
- [Printing Overview](#)
- [Microsoft XPS Document Writer](#)

# How to: Get Print System Object Properties Without Reflection

4/9/2019 • 2 minutes to read • [Edit Online](#)

Using reflection to itemize the properties (and the types of those properties) on an object can slow application performance. The [System.Printing.IndexedProperties](#) namespace provides a means to getting this information without using reflection.

## Example

The steps for doing this are as follows.

1. Create an instance of the type. In the example below, the type is the [PrintQueue](#) type that ships with Microsoft .NET Framework, but nearly identical code should work for types that you derive from [PrintSystemObject](#).
2. Create a [PrintPropertyDictionary](#) from the type's [PropertiesCollection](#). The [Value](#) property of each entry in this dictionary is an object of one of the types derived from [PrintProperty](#).
3. Enumerate the members of the dictionary. For each of them, do the following.
4. Up-cast the value of each entry to [PrintProperty](#) and use it to create a [PrintProperty](#) object.
5. Get the type of the [Value](#) of each of the [PrintProperty](#) object.

```
// Enumerate the properties, and their types, of a queue without using Reflection
LocalPrintServer localPrintServer = new LocalPrintServer();
PrintQueue defaultPrintQueue = LocalPrintServer.GetDefaultPrintQueue();

PrintPropertyDictionary printQueueProperties = defaultPrintQueue.PropertiesCollection;

Console.WriteLine("These are the properties, and their types, of {0}, a {1}", defaultPrintQueue.Name,
defaultPrintQueue.GetType().ToString() + "\n");

foreach (DictionaryEntry entry in printQueueProperties)
{
    PrintProperty property = (PrintProperty)entry.Value;

    if (property.Value != null)
    {
        Console.WriteLine(property.Name + "\t(Type: {0})", property.Value.GetType().ToString());
    }
}
Console.WriteLine("\n\nPress Return to continue...");
Console.ReadLine();
```

```
' Enumerate the properties, and their types, of a queue without using Reflection
Dim localPrintServer As New LocalPrintServer()
Dim defaultPrintQueue As PrintQueue = LocalPrintServer.GetDefaultPrintQueue()

Dim printQueueProperties As PrintPropertyDictionary = defaultPrintQueue.PropertiesCollection

Console.WriteLine("These are the properties, and their types, of {0}, a {1}", defaultPrintQueue.Name,
defaultPrintQueue.GetType().ToString() + vbCrLf)

For Each entry As DictionaryEntry In printQueueProperties
    Dim [property] As PrintProperty = CType(entry.Value, PrintProperty)

    If [property].Value IsNot Nothing Then
        Console.WriteLine([property].Name & vbTab & "(Type: {0})", [property].Value.GetType().ToString())
    End If
Next entry
Console.WriteLine(vbLf & vbCrLf & "Press Return to continue...")
Console.ReadLine()
```

## See also

- [PrintProperty](#)
- [PrintSystemObject](#)
- [System.Printing.IndexedProperties](#)
- [PrintPropertyDictionary](#)
- [LocalPrintServer](#)
- [PrintQueue](#)
- [DictionaryEntry](#)
- [Documents in WPF](#)
- [Printing Overview](#)

# How to: Programmatically Print XPS Files

11/12/2019 • 7 minutes to read • [Edit Online](#)

You can use one overload of the [AddJob](#) method to print XML Paper Specification (XPS) files without opening a [PrintDialog](#) or, in principle, any user interface (UI) at all.

You can also print XPS files using the many [XpsDocumentWriter.Write](#) and [XpsDocumentWriter.WriteAsync](#) methods. For more information, see [Printing an XPS Document](#).

Another way of printing XPS is to use the [PrintDialog.PrintDocument](#) or [PrintDialog.PrintVisual](#) methods. See [Invoke a Print Dialog](#).

## Example

The main steps to using the three-parameter [AddJob\(String, String, Boolean\)](#) method are as follows. The example below gives details.

1. Determine if the printer is an XPSDrv printer. See [Printing Overview](#) for more about XPSDrv.
2. If the printer is not an XPSDrv printer, set the thread's apartment to single thread.
3. Instantiate a print server and print queue object.
4. Call the method, specifying a job name, the file to be printed, and a [Boolean](#) flag indicating whether or not the printer is an XPSDrv printer.

The example below shows how to batch print all XPS files in a directory. Although the application prompts the user to specify the directory, the three-parameter [AddJob\(String, String, Boolean\)](#) method does not require a user interface (UI). It can be used in any code path where you have an XPS file name and path that you can pass to it.

The three-parameter [AddJob\(String, String, Boolean\)](#) overload of [AddJob](#) must run in a single thread apartment whenever the [Boolean](#) parameter is `false`, which it must be when a non-XPSDrv printer is being used. However, the default apartment state for .NET is multiple thread. This default must be reversed since the example assumes a non-XPSDrv printer.

There are two ways to change the default. One way is to simply add the [STAThreadAttribute](#) (that is, "`[System.STAThreadAttribute()]`") just above the first line of the application's `Main` method (usually "`static void Main(string[] args)`"). However, many applications require that the `Main` method have a multi-threaded apartment state, so there is a second method: put the call to [AddJob\(String, String, Boolean\)](#) in a separate thread whose apartment state is set to [STA](#) with [SetApartmentState](#). The example below uses this second technique.

Accordingly, the example begins by instantiating a [Thread](#) object and passing it a [PrintXPS](#) method as the [ThreadStart](#) parameter. (The [PrintXPS](#) method is defined later in the example.) Next the thread is set to a single thread apartment. The only remaining code of the `Main` method starts the new thread.

The meat of the example is in the `static BatchXPSPrinter.PrintXPS` method. After creating a print server and queue, the method prompts the user for a directory containing XPS files. After validating the existence of the directory and the presence of \*.xps files in it, the method adds each such file to the print queue. The example assumes that the printer is non-XPSDrv, so we are passing `false` to the last parameter of [AddJob\(String, String, Boolean\)](#) method. For this reason, the method will validate the XPS markup in the file before it attempts to convert it to the printer's page description language. If the validation fails, an exception is thrown. The example code will catch the exception, notify the user about it, and then go on to process the next XPS file.

```

class Program
{
    [System.MTAThreadAttribute()] // Added for clarity, but this line is redundant because MTA is the default.
    static void Main(string[] args)
    {
        // Create the secondary thread and pass the printing method for
        // the constructor's ThreadStart delegate parameter. The BatchXPSPrinter
        // class is defined below.
        Thread printingThread = new Thread(BatchXPSPrinter.PrintXPS);

        // Set the thread that will use PrintQueue.AddJob to single threading.
        printingThread.SetApartmentState(ApartmentState.STA);

        // Start the printing thread. The method passed to the Thread
        // constructor will execute.
        printingThread.Start();

    } // end Main

} // end Program class

public class BatchXPSPrinter
{
    public static void PrintXPS()
    {
        // Create print server and print queue.
        LocalPrintServer localPrintServer = new LocalPrintServer();
        PrintQueue defaultPrintQueue = LocalPrintServer.GetDefaultPrintQueue();

        // Prompt user to identify the directory, and then create the directory object.
        Console.WriteLine("Enter the directory containing the XPS files: ");
        String directoryPath = Console.ReadLine();
        DirectoryInfo dir = new DirectoryInfo(directoryPath);

        // If the user mistyped, end the thread and return to the Main thread.
        if (!dir.Exists)
        {
            Console.WriteLine("There is no such directory.");
        }
        else
        {
            // If there are no XPS files in the directory, end the thread
            // and return to the Main thread.
            if (dir.GetFiles("*.*").Length == 0)
            {
                Console.WriteLine("There are no XPS files in the directory.");
            }
            else
            {
                Console.WriteLine("\nJobs will now be added to the print queue.");
                Console.WriteLine("If the queue is not paused and the printer is working, jobs will begin
printing.");

                // Batch process all XPS files in the directory.
                foreach (FileInfo f in dir.GetFiles("*.*"))
                {
                    String nextFile = directoryPath + "\\\" + f.Name;
                    Console.WriteLine("Adding {0} to queue.", nextFile);

                    try
                    {
                        // Print the Xps file while providing XPS validation and progress notifications.
                        PrintSystemJobInfo xpsPrintJob = defaultPrintQueue.AddJob(f.Name, nextFile, false);
                    }
                    catch (PrintJobException e)
                    {
                        Console.WriteLine("\n\t{0} could not be added to the print queue.", f.Name);
                        if (e.InnerException.Message == "File contains corrupted data.")

```

```

        {
            Console.WriteLine("\tIt is not a valid XPS file. Use the isXPS Conformance Tool to
debug it.");
        }
        Console.WriteLine("\tContinuing with next XPS file.\n");
    }

}// end for each XPS file

}//end if there are no XPS files in the directory

}//end if the directory does not exist

Console.WriteLine("Press Enter to end program.");
Console.ReadLine();

}// end PrintXPS method

}// end BatchXPSPrinter class

```

```

Friend Class Program
<System.MTAThreadAttribute()>
Shared Sub Main(ByVal args() As String) ' Added for clarity, but this line is redundant because MTA is the
default.
    ' Create the secondary thread and pass the printing method for
    ' the constructor's ThreadStart delegate parameter. The BatchXPSPrinter
    ' class is defined below.
    Dim printingThread As New Thread(AddressOf BatchXPSPrinter.PrintXPS)

    ' Set the thread that will use PrintQueue.AddJob to single threading.
    printingThread.SetApartmentState(ApartmentState.STA)

    ' Start the printing thread. The method passed to the Thread
    ' constructor will execute.
    printingThread.Start()

End Sub

End Class

Public Class BatchXPSPrinter
    Public Shared Sub PrintXPS()
        ' Create print server and print queue.
        Dim localPrintServer As New LocalPrintServer()
        Dim defaultPrintQueue As PrintQueue = LocalPrintServer.GetDefaultPrintQueue()

        ' Prompt user to identify the directory, and then create the directory object.
        Console.Write("Enter the directory containing the XPS files: ")
        Dim directoryPath As String = Console.ReadLine()
        Dim dir As New DirectoryInfo(directoryPath)

        ' If the user mistyped, end the thread and return to the Main thread.
        If Not dir.Exists Then
            Console.WriteLine("There is no such directory.")
        Else
            ' If there are no XPS files in the directory, end the thread
            ' and return to the Main thread.
            If dir.GetFiles("*.xps").Length = 0 Then
                Console.WriteLine("There are no XPS files in the directory.")
            Else
                Console.WriteLine(vbLf & "Jobs will now be added to the print queue.")
                Console.WriteLine("If the queue is not paused and the printer is working, jobs will begin
printing.")

                ' Batch process all XPS files in the directory.
                For Each f As FileInfo In dir.GetFiles("*.xps")
                    Dim nextFile As String = directoryPath & "\" & f.Name

```

```

        Console.WriteLine("Adding {0} to queue.", nextFile)

    Try
        ' Print the Xps file while providing XPS validation and progress notifications.
        Dim xpsPrintJob As PrintSystemJobInfo = defaultPrintQueue.AddJob(f.Name, nextFile,
False)
    Catch e As PrintJobException
        Console.WriteLine(vbLf & vbTab & "{0} could not be added to the print queue.", f.Name)
        If e.InnerException.Message = "File contains corrupted data." Then
            Console.WriteLine(vbTab & "It is not a valid XPS file. Use the isXPS Conformance
Tool to debug it.")
        End If
        Console.WriteLine(vbTab & "Continuing with next XPS file." & vbCrLf)
    End Try

    Next f ' end for each XPS file

    End If 'end if there are no XPS files in the directory

    End If 'end if the directory does not exist

    Console.WriteLine("Press Enter to end program.")
    Console.ReadLine()

End Sub

End Class

```

If you are using an XPSDrv printer, then you can set the final parameter to `true`. In that case, since XPS is the printer's page description language, the method will send the file to the printer without validating it or converting it to another page description language. If you are uncertain at design time whether the application will be using an XPSDrv printer, you can modify the application to have it read the `IsXpsDevice` property and branch according to what it finds.

Since there will initially be few XPSDrv printers available immediately after the release of Windows Vista and Microsoft .NET Framework, you may need to disguise a non-XPSDrv printer as an XPSDrv printer. To do so, add `Pipelineconfig.xml` to the list of files in the following registry key of the computer running your application:

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Environments\Windows NT
x86\Drivers\Version-3\<Printer>\DependentFiles`

where `<Printer>` is any print queue. The machine must then be rebooted.

This disguise will enable you to pass `true` as the final parameter of `AddJob(String, String, Boolean)` without causing an exception, but since `<Printer>` is not really an XPSDrv printer, only garbage will print.

#### NOTE

For simplicity, the example above uses the presence of an \*.xps extension as its test that a file is XPS. However, XPS files do not have to have this extension. The [isXPS.exe \(isXPS Conformance Tool\)](#) is one way of testing a file for XPS validity.

## See also

- [PrintQueue](#)
- [AddJob](#)
- [ApartmentState](#)
- [STAThreadAttribute](#)
- [XPS Documents](#)
- [Printing an XPS Document](#)

- Managed and Unmanaged Threading
- isXPS.exe (isXPS Conformance Tool)
- Documents in WPF
- Printing Overview

# How to: Remotely Survey the Status of Printers

8/22/2019 • 13 minutes to read • [Edit Online](#)

At any given time at medium and large companies there may be multiple printers that are not working due to a paper jam or being out of paper or some other problematic situation. The rich set of printer properties exposed in the APIs of Microsoft .NET Framework provide a means for performing a rapid survey of the states of printers.

## Example

The major steps for creating this kind of utility are as follows.

1. Obtain a list of all print servers.
2. Loop through the servers to query their print queues.
3. Within each pass of the server loop, loop through all the server's queues and read each property that might indicate that the queue is not currently working.

The code below is a series of snippets. For simplicity, this example assumes that there is a CRLF-delimited list of print servers. The variable `fileOfPrintServers` is a `StreamReader` object for this file. Since each server name is on its own line, any call of `ReadLine` gets the name of the next server and moves the `StreamReader`'s cursor to the beginning of the next line.

Within the outer loop, the code creates a `PrintServer` object for the latest print server and specifies that the application is to have administrative rights to the server.

### NOTE

If there are a lot of servers, you can improve performance by using the `PrintServer(String, String[], PrintSystemDesiredAccess)` constructors that only initialize the properties you are going to need.

The example then uses `GetPrintQueues` to create a collection of all of the server's queues and begins to loop through them. This inner loop contains a branching structure corresponding to the two ways of checking a printer's status:

- You can read the flags of the `QueueStatus` property which is of type `PrintQueueStatus`.
- You can read each relevant property such as `IsOutOfPaper`, and `IsPaperJammed`.

This example demonstrates both methods, so the user was previously prompted as to which method to use and responded with "y" if he or she wanted to use the flags of the `QueueStatus` property. See below for the details of the two methods.

Finally, the results are presented to the user.

```

// Survey queue status for every queue on every print server
System::String^ line;
System::String^ statusReport = "\n\nAny problem states are indicated below:\n\n";
while ((line = fileOfPrintServers->ReadLine()) != nullptr)
{
    System::Printing::PrintServer^ myPS = gcnew System::Printing::PrintServer(line,
PrintSystemDesiredAccess::AdministrateServer);
    System::Printing::PrintQueueCollection^ myPrintQueues = myPS->GetPrintQueues();
    statusReport = statusReport + "\n" + line;
    for each (System::Printing::PrintQueue^ pq in myPrintQueues)
    {
        pq->Refresh();
        statusReport = statusReport + "\n\t" + pq->Name + ":";

        if (useAttributesResponse == "y")
        {
            TroubleSpotter::SpotTroubleUsingQueueAttributes(statusReport, pq);
            // TroubleSpotter class is defined in the complete example.
        } else
        {
            TroubleSpotter::SpotTroubleUsingProperties(statusReport, pq);
        }
    }
}
fileOfPrintServers->Close();
Console::WriteLine(statusReport);
Console::WriteLine("\nPress Return to continue.");
Console::ReadLine();

```

```

// Survey queue status for every queue on every print server
String line;
String statusReport = "\n\nAny problem states are indicated below:\n\n";
while ((line = fileOfPrintServers.ReadLine()) != null)
{
    PrintServer myPS = new PrintServer(line, PrintSystemDesiredAccess.AdministrateServer);
    PrintQueueCollection myPrintQueues = myPS.GetPrintQueues();
    statusReport = statusReport + "\n" + line;
    foreach (PrintQueue pq in myPrintQueues)
    {
        pq.Refresh();
        statusReport = statusReport + "\n\t" + pq.Name + ":";

        if (useAttributesResponse == "y")
        {
            TroubleSpotter.SpotTroubleUsingQueueAttributes(ref statusReport, pq);
            // TroubleSpotter class is defined in the complete example.
        }
        else
        {
            TroubleSpotter.SpotTroubleUsingProperties(ref statusReport, pq);
        }
    }
    // end for each print queue
}

// end while list of print servers is not yet exhausted

fileOfPrintServers.Close();
Console.WriteLine(statusReport);
Console.WriteLine("\nPress Return to continue.");
Console.ReadLine();

```

```

' Survey queue status for every queue on every print server
Dim line As String
Dim statusReport As String = vbLf & vbLf & "Any problem states are indicated below:" & vbLf & vbLf
line = fileOfPrintServers.ReadLine()
Do While line IsNot Nothing
    Dim myPS As New PrintServer(line, PrintSystemDesiredAccess.AdministrateServer)
    Dim myPrintQueues As PrintQueueCollection = myPS.GetPrintQueues()
    statusReport = statusReport & vbLf & line
    For Each pq As PrintQueue In myPrintQueues
        pq.Refresh()
        statusReport = statusReport & vbLf & vbTab & pq.Name & ":" 
        If useAttributesResponse = "y" Then
            TroubleSpotter.SpotTroubleUsingQueueAttributes(statusReport, pq)
            ' TroubleSpotter class is defined in the complete example.
        Else
            TroubleSpotter.SpotTroubleUsingProperties(statusReport, pq)
        End If
    Next pq ' end for each print queue
    line = fileOfPrintServers.ReadLine()
Loop ' end while list of print servers is not yet exhausted

fileOfPrintServers.Close()
Console.WriteLine(statusReport)
Console.WriteLine(vbLf & "Press Return to continue.")
Console.ReadLine()

```

To check printer status using the flags of the [QueueStatus](#) property, you check each relevant flag to see if it is set. The standard way to see if one bit is set in a set of bit flags is to perform a logical AND operation with the set of flags as one operand and the flag itself as the other. Since the flag itself has only one bit set, the result of the logical AND is that, at most, that same bit is set. To find out whether it is or not, just compare the result of the logical AND with the flag itself. For more information, see [PrintQueueStatus](#), the [& Operator \(C# Reference\)](#), and [FlagsAttribute](#).

For each attribute whose bit is set, the code adds a notice to the final report that will be presented to the user. (The [ReportAvailabilityAtThisTime](#) method that is called at the end of the code is discussed below.)

```

internal:
    // Check for possible trouble states of a printer using the flags of the QueueStatus property
    static void SpotTroubleUsingQueueAttributes (System::String^% statusReport, System::Printing::PrintQueue^
pq)
    {
        if ((pq->QueueStatus & PrintQueueStatus::PaperProblem) == PrintQueueStatus::PaperProblem)
        {
            statusReport = statusReport + "Has a paper problem. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::NoToner) == PrintQueueStatus::NoToner)
        {
            statusReport = statusReport + "Is out of toner. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::DoorOpen) == PrintQueueStatus::DoorOpen)
        {
            statusReport = statusReport + "Has an open door. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::Error) == PrintQueueStatus::Error)
        {
            statusReport = statusReport + "Is in an error state. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::NotAvailable) == PrintQueueStatus::NotAvailable)
        {
            statusReport = statusReport + "Is not available. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::Offline) == PrintQueueStatus::Offline)
        {
            statusReport = statusReport + "Is off line. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::OutOfMemory) == PrintQueueStatus::OutOfMemory)
        {
            statusReport = statusReport + "Is out of memory. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::PaperOut) == PrintQueueStatus::PaperOut)
        {
            statusReport = statusReport + "Is out of paper. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::OutputBinFull) == PrintQueueStatus::OutputBinFull)
        {
            statusReport = statusReport + "Has a full output bin. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::PaperJam) == PrintQueueStatus::PaperJam)
        {
            statusReport = statusReport + "Has a paper jam. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::Paused) == PrintQueueStatus::Paused)
        {
            statusReport = statusReport + "Is paused. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::TonerLow) == PrintQueueStatus::TonerLow)
        {
            statusReport = statusReport + "Is low on toner. ";
        }
        if ((pq->QueueStatus & PrintQueueStatus::UserIntervention) == PrintQueueStatus::UserIntervention)
        {
            statusReport = statusReport + "Needs user intervention. ";
        }

        // Check if queue is even available at this time of day
        // The method below is defined in the complete example.
        ReportAvailabilityAtThisTime(statusReport, pq);
    };

```

```

// Check for possible trouble states of a printer using the flags of the QueueStatus property
internal static void SpotTroubleUsingQueueAttributes(ref String statusReport, PrintQueue pq)
{
    if ((pq.QueueStatus & PrintQueueStatus.PaperProblem) == PrintQueueStatus.PaperProblem)
    {
        statusReport = statusReport + "Has a paper problem. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.NoToner) == PrintQueueStatus.NoToner)
    {
        statusReport = statusReport + "Is out of toner. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.DoorOpen) == PrintQueueStatus.DoorOpen)
    {
        statusReport = statusReport + "Has an open door. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.Error) == PrintQueueStatus.Error)
    {
        statusReport = statusReport + "Is in an error state. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.NotAvailable) == PrintQueueStatus.NotAvailable)
    {
        statusReport = statusReport + "Is not available. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.Offline) == PrintQueueStatus.Offline)
    {
        statusReport = statusReport + "Is off line. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.OutOfMemory) == PrintQueueStatus.OutOfMemory)
    {
        statusReport = statusReport + "Is out of memory. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.PaperOut) == PrintQueueStatus.PaperOut)
    {
        statusReport = statusReport + "Is out of paper. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.OutputBinFull) == PrintQueueStatus.OutputBinFull)
    {
        statusReport = statusReport + "Has a full output bin. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.PaperJam) == PrintQueueStatus.PaperJam)
    {
        statusReport = statusReport + "Has a paper jam. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.Paused) == PrintQueueStatus.Paused)
    {
        statusReport = statusReport + "Is paused. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.TonerLow) == PrintQueueStatus.TonerLow)
    {
        statusReport = statusReport + "Is low on toner. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.UserIntervention) == PrintQueueStatus.UserIntervention)
    {
        statusReport = statusReport + "Needs user intervention. ";
    }

    // Check if queue is even available at this time of day
    // The method below is defined in the complete example.
    ReportAvailabilityAtThisTime(ref statusReport, pq);
}

```

```

' Check for possible trouble states of a printer using the flags of the QueueStatus property
Friend Shared Sub SpotTroubleUsingQueueAttributes(ByRef statusReport As String, ByVal pq As PrintQueue)
    If (pq.QueueStatus And PrintQueueStatus.PaperProblem) = PrintQueueStatus.PaperProblem Then
        statusReport = statusReport & "Has a paper problem. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.NoToner) = PrintQueueStatus.NoToner Then
        statusReport = statusReport & "Is out of toner. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.DoorOpen) = PrintQueueStatus.DoorOpen Then
        statusReport = statusReport & "Has an open door. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.Error) = PrintQueueStatus.Error Then
        statusReport = statusReport & "Is in an error state. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.NotAvailable) = PrintQueueStatus.NotAvailable Then
        statusReport = statusReport & "Is not available. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.Offline) = PrintQueueStatus.Offline Then
        statusReport = statusReport & "Is off line. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.OutOfMemory) = PrintQueueStatus.OutOfMemory Then
        statusReport = statusReport & "Is out of memory. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.PaperOut) = PrintQueueStatus.PaperOut Then
        statusReport = statusReport & "Is out of paper. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.OutputBinFull) = PrintQueueStatus.OutputBinFull Then
        statusReport = statusReport & "Has a full output bin. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.PaperJam) = PrintQueueStatus.PaperJam Then
        statusReport = statusReport & "Has a paper jam. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.Paused) = PrintQueueStatus.Paused Then
        statusReport = statusReport & "Is paused. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.TonerLow) = PrintQueueStatus.TonerLow Then
        statusReport = statusReport & "Is low on toner. "
    End If
    If (pq.QueueStatus And PrintQueueStatus.UserIntervention) = PrintQueueStatus.UserIntervention Then
        statusReport = statusReport & "Needs user intervention. "
    End If

    ' Check if queue is even available at this time of day
    ' The method below is defined in the complete example.
    ReportAvailabilityAtThisTime(statusReport, pq)
End Sub

```

To check printer status using each property, you simply read each property and add a note to the final report that will be presented to the user if the property is `true`. (The **ReportAvailabilityAtThisTime** method that is called at the end of the code is discussed below.)

```

internal:
    // Check for possible trouble states of a printer using its properties
    static void SpotTroubleUsingProperties (System::String^% statusReport, System::Printing::PrintQueue^ pq)
    {
        if (pq->HasPaperProblem)
        {
            statusReport = statusReport + "Has a paper problem. ";
        }
        if (!(pq->HasToner))
        {
            statusReport = statusReport + "Is out of toner. ";
        }
        if (pq->IsDoorOpened)
        {
            statusReport = statusReport + "Has an open door. ";
        }
        if (pq->IsInError)
        {
            statusReport = statusReport + "Is in an error state. ";
        }
        if (pq-> IsNotAvailable)
        {
            statusReport = statusReport + "Is not available. ";
        }
        if (pq->IsOffline)
        {
            statusReport = statusReport + "Is off line. ";
        }
        if (pq->IsOutOfMemory)
        {
            statusReport = statusReport + "Is out of memory. ";
        }
        if (pq->IsOutOfPaper)
        {
            statusReport = statusReport + "Is out of paper. ";
        }
        if (pq->IsOutputBinFull)
        {
            statusReport = statusReport + "Has a full output bin. ";
        }
        if (pq->IsPaperJammed)
        {
            statusReport = statusReport + "Has a paper jam. ";
        }
        if (pq->IsPaused)
        {
            statusReport = statusReport + "Is paused. ";
        }
        if (pq->IsTonerLow)
        {
            statusReport = statusReport + "Is low on toner. ";
        }
        if (pq->NeedUserIntervention)
        {
            statusReport = statusReport + "Needs user intervention. ";
        }

        // Check if queue is even available at this time of day
        // The following method is defined in the complete example.
        ReportAvailabilityAtThisTime(statusReport, pq);
    };

```

```

// Check for possible trouble states of a printer using its properties
internal static void SpotTroubleUsingProperties(ref String statusReport, PrintQueue pq)
{
    if (pq.HasPaperProblem)
    {
        statusReport = statusReport + "Has a paper problem. ";
    }
    if (!(pq.HasToner))
    {
        statusReport = statusReport + "Is out of toner. ";
    }
    if (pq.IsDoorOpened)
    {
        statusReport = statusReport + "Has an open door. ";
    }
    if (pq.IsInError)
    {
        statusReport = statusReport + "Is in an error state. ";
    }
    if (pq.IsNotNullAvailable)
    {
        statusReport = statusReport + "Is not available. ";
    }
    if (pq.IsNotNullOffline)
    {
        statusReport = statusReport + "Is off line. ";
    }
    if (pq.IsNotNullOfMemory)
    {
        statusReport = statusReport + "Is out of memory. ";
    }
    if (pq.IsNotNullOfPaper)
    {
        statusReport = statusReport + "Is out of paper. ";
    }
    if (pq.IsNotNullBinFull)
    {
        statusReport = statusReport + "Has a full output bin. ";
    }
    if (pq.IsPaperJammed)
    {
        statusReport = statusReport + "Has a paper jam. ";
    }
    if (pq.IsPaused)
    {
        statusReport = statusReport + "Is paused. ";
    }
    if (pq.IsTonerLow)
    {
        statusReport = statusReport + "Is low on toner. ";
    }
    if (pq.NeedUserIntervention)
    {
        statusReport = statusReport + "Needs user intervention. ";
    }

    // Check if queue is even available at this time of day
    // The following method is defined in the complete example.
    ReportAvailabilityAtThisTime(ref statusReport, pq);

} //end SpotTroubleUsingProperties

```

```

' Check for possible trouble states of a printer using its properties
Friend Shared Sub SpotTroubleUsingProperties(ByRef statusReport As String, ByVal pq As PrintQueue)
    If pq.HasPaperProblem Then
        statusReport = statusReport & "Has a paper problem. "
    End If
    If Not(pq.HasToner) Then
        statusReport = statusReport & "Is out of toner. "
    End If
    If pq.IsDoorOpened Then
        statusReport = statusReport & "Has an open door. "
    End If
    If pq.IsInError Then
        statusReport = statusReport & "Is in an error state. "
    End If
    If pq.IsNotNullAvailable Then
        statusReport = statusReport & "Is not available. "
    End If
    If pq.IsOffline Then
        statusReport = statusReport & "Is off line. "
    End If
    If pq.IsOutOfMemory Then
        statusReport = statusReport & "Is out of memory. "
    End If
    If pq.IsOutOfPaper Then
        statusReport = statusReport & "Is out of paper. "
    End If
    If pq.IsOutputBinFull Then
        statusReport = statusReport & "Has a full output bin. "
    End If
    If pq.IsPaperJammed Then
        statusReport = statusReport & "Has a paper jam. "
    End If
    If pq.IsPaused Then
        statusReport = statusReport & "Is paused. "
    End If
    If pq.IsTonerLow Then
        statusReport = statusReport & "Is low on toner. "
    End If
    If pq.NeedUserIntervention Then
        statusReport = statusReport & "Needs user intervention. "
    End If

    ' Check if queue is even available at this time of day
    ' The following method is defined in the complete example.
    ReportAvailabilityAtThisTime(statusReport, pq)

End Sub

```

The **ReportAvailabilityAtThisTime** method was created in case you need to determine if the queue is available at the current time of day.

The method will do nothing if the [StartTimeOfDay](#) and [UntilTimeOfDay](#) properties are equal; because in that case the printer is available at all times. If they are different, the method gets the current time which then has to be converted into total minutes past midnight because the [StartTimeOfDay](#) and [UntilTimeOfDay](#) properties are [Int32](#)s representing minutes-after-midnight, not [DateTime](#) objects. Finally, the method checks to see if the current time is between the start and "until" times.

```

private:
    static void ReportAvailabilityAtThisTime (System::String^% statusReport, System::Printing::PrintQueue^ pq)
    {
        if (pq->StartTimeOfDay != pq->UntilTimeOfDay)
        {
            System::DateTime utcNow = DateTime::UtcNow;
            System::Int32 utcNowAsMinutesAfterMidnight = (utcNow.TimeOfDay.Hours * 60) +
            utcNow.TimeOfDay.Minutes;

            // If now is not within the range of available times . . .
            if (!((pq->StartTimeOfDay < utcNowAsMinutesAfterMidnight) && (utcNowAsMinutesAfterMidnight < pq-
            >UntilTimeOfDay)))
            {
                statusReport = statusReport + " Is not available at this time of day. ";
            }
        }
    };

```

```

private static void ReportAvailabilityAtThisTime(ref String statusReport, PrintQueue pq)
{
    if (pq.StartTimeOfDay != pq.UntilTimeOfDay) // If the printer is not available 24 hours a day
    {
        DateTime utcNow = DateTime.UtcNow;
        Int32 utcNowAsMinutesAfterMidnight = (utcNow.TimeOfDay.Hours * 60) + utcNow.TimeOfDay.Minutes;

        // If now is not within the range of available times . . .
        if (!((pq.StartTimeOfDay < utcNowAsMinutesAfterMidnight)
              &&
              (utcNowAsMinutesAfterMidnight < pq.UntilTimeOfDay)))
        {
            statusReport = statusReport + " Is not available at this time of day. ";
        }
    }
}

```

```

Private Shared Sub ReportAvailabilityAtThisTime(ByRef statusReport As String, ByVal pq As PrintQueue)
    If pq.StartTimeOfDay <> pq.UntilTimeOfDay Then ' If the printer is not available 24 hours a day
        Dim utcNow As Date = Date.UtcNow
        Dim utcNowAsMinutesAfterMidnight As Int32 = (utcNow.TimeOfDay.Hours * 60) + utcNow.TimeOfDay.Minutes

        ' If now is not within the range of available times . . .
        If Not((pq.StartTimeOfDay < utcNowAsMinutesAfterMidnight) AndAlso (utcNowAsMinutesAfterMidnight <
        pq.UntilTimeOfDay)) Then
            statusReport = statusReport & " Is not available at this time of day. "
        End If
    End If
End Sub

```

## See also

- [StartTimeOfDay](#)
- [UntilTimeOfDay](#)
- [DateTime](#)
- [PrintQueueStatus](#)
- [FlagsAttribute](#)
- [GetPrintQueues](#)
- [PrintServer](#)
- [LocalPrintServer](#)

- [EnumeratedPrintQueueTypes](#)
- [PrintQueue](#)
- [& Operator \(C# Reference\)](#)
- [Documents in WPF](#)
- [Printing Overview](#)

# How to: Validate and Merge PrintTickets

10/29/2019 • 8 minutes to read • [Edit Online](#)

The Microsoft Windows [Print Schema](#) includes the flexible and extensible [PrintCapabilities](#) and [PrintTicket](#) elements. The former itemizes the capabilities of a print device and the latter specifies how the device should use those capabilities with respect to a particular sequence of documents, individual document, or individual page.

A typical sequence of tasks for an application that supports printing would be as follows.

1. Determine a printer's capabilities.
2. Configure a [PrintTicket](#) to use those capabilities.
3. Validate the [PrintTicket](#).

This article shows how to do this.

## Example

In the simple example below, we are interested only in whether a printer can support duplexing — two-sided printing. The major steps are as follows.

1. Get a [PrintCapabilities](#) object with the [GetPrintCapabilities](#) method.
2. Test for the presence of the capability you want. In the example below, we test the [DuplexingCapability](#) property of the [PrintCapabilities](#) object for the presence of the capability of printing on both sides of a sheet of paper with the "page turning" along the long side of the sheet. Since [DuplexingCapability](#) is a collection, we use the `Contains` method of [ReadOnlyCollection<T>](#).

### NOTE

This step is not strictly necessary. The [MergeAndValidatePrintTicket](#) method used below will check each request in the [PrintTicket](#) against the capabilities of the printer. If the requested capability is not supported by printer, the printer driver will substitute an alternative request in the [PrintTicket](#) returned by the method.

3. If the printer supports duplexing, the sample code creates a [PrintTicket](#) that asks for duplexing. But the application does not specify every possible printer setting available in the [PrintTicket](#) element. That would be wasteful of both programmer and program time. Instead, the code sets only the duplexing request and then merges this [PrintTicket](#) with an existing, fully configured and validated, [PrintTicket](#), in this case, the user's default [PrintTicket](#).
4. Accordingly, the sample calls the [MergeAndValidatePrintTicket](#) method to merge the new, minimal, [PrintTicket](#) with the user's default [PrintTicket](#). This returns a [ValidationResult](#) that includes the new [PrintTicket](#) as one of its properties.
5. The sample then tests that the new [PrintTicket](#) requests duplexing. If it does, then the sample makes it the new default print ticket for the user. If step 2 above had been left out and the printer did not support duplexing along the long side, then the test would have resulted in `false`. (See the note above.)
6. The last significant step is to commit the change to the [UserPrintTicket](#) property of the [PrintQueue](#) with the [Commit](#) method.

```

/// <summary>
/// Changes the user-default PrintTicket setting of the specified print queue.
/// </summary>
/// <param name="queue">the printer whose user-default PrintTicket setting needs to be changed</param>
static private void ChangePrintTicketSetting(PrintQueue queue)
{
    //
    // Obtain the printer's PrintCapabilities so we can determine whether or not
    // duplexing printing is supported by the printer.
    //
    PrintCapabilities printcap = queue.GetPrintCapabilities();

    //
    // The printer's duplexing capability is returned as a read-only collection of duplexing options
    // that can be supported by the printer. If the collection returned contains the duplexing
    // option we want to set, it means the duplexing option we want to set is supported by the printer,
    // so we can make the user-default PrintTicket setting change.
    //
    if (printcap.DuplexingCapability.Contains(Duplexing.TwoSidedLongEdge))
    {
        //
        // To change the user-default PrintTicket, we can first create a delta PrintTicket with
        // the new duplexing setting.
        //
        PrintTicket deltaTicket = new PrintTicket();
        deltaTicket.Duplexing = Duplexing.TwoSidedLongEdge;

        //
        // Then merge the delta PrintTicket onto the printer's current user-default PrintTicket,
        // and validate the merged PrintTicket to get the new PrintTicket we want to set as the
        // printer's new user-default PrintTicket.
        //
        ValidationResult result = queue.MergeAndValidatePrintTicket(queue.UserPrintTicket, deltaTicket);

        //
        // The duplexing option we want to set could be constrained by other PrintTicket settings
        // or device settings. We can check the validated merged PrintTicket to see whether the
        // the validation process has kept the duplexing option we want to set unchanged.
        //
        if (result.ValidatedPrintTicket.Duplexing == Duplexing.TwoSidedLongEdge)
        {
            //
            // Set the printer's user-default PrintTicket and commit the set operation.
            //
            queue.UserPrintTicket = result.ValidatedPrintTicket;
            queue.Commit();
            Console.WriteLine("PrintTicket new duplexing setting is set on '{0}'.", queue.FullName);
        }
        else
        {
            //
            // The duplexing option we want to set has been changed by the validation process
            // when it was resolving setting constraints.
            //
            Console.WriteLine("PrintTicket new duplexing setting is constrained on '{0}'.", queue.FullName);
        }
    }
    else
    {
        //
        // If the printer doesn't support the duplexing option we want to set, skip it.
        //
        Console.WriteLine("PrintTicket new duplexing setting is not supported on '{0}'.", queue.FullName);
    }
}

```

```

''' <summary>
''' Changes the user-default PrintTicket setting of the specified print queue.
''' </summary>
''' <param name="queue">the printer whose user-default PrintTicket setting needs to be changed</param>
Private Shared Sub ChangePrintTicketSetting(ByVal queue As PrintQueue)

    ' Obtain the printer's PrintCapabilities so we can determine whether or not
    ' duplexing printing is supported by the printer.
    '

    Dim printcap As PrintCapabilities = queue.GetPrintCapabilities()

    '

    ' The printer's duplexing capability is returned as a read-only collection of duplexing options
    ' that can be supported by the printer. If the collection returned contains the duplexing
    ' option we want to set, it means the duplexing option we want to set is supported by the printer,
    ' so we can make the user-default PrintTicket setting change.
    '

    If printcap.DuplexingCapability.Contains(Duplexing.TwoSidedLongEdge) Then
        '

        ' To change the user-default PrintTicket, we can first create a delta PrintTicket with
        ' the new duplexing setting.
        '

        Dim deltaTicket As New PrintTicket()
        deltaTicket.Duplexing = Duplexing.TwoSidedLongEdge

        '

        ' Then merge the delta PrintTicket onto the printer's current user-default PrintTicket,
        ' and validate the merged PrintTicket to get the new PrintTicket we want to set as the
        ' printer's new user-default PrintTicket.
        '

        Dim result As ValidationResult = queue.MergeAndValidatePrintTicket(queue.UserPrintTicket, deltaTicket)

        '

        ' The duplexing option we want to set could be constrained by other PrintTicket settings
        ' or device settings. We can check the validated merged PrintTicket to see whether the
        ' validation process has kept the duplexing option we want to set unchanged.
        '

        If result.ValidatedPrintTicket.Duplexing = Duplexing.TwoSidedLongEdge Then
            '

            ' Set the printer's user-default PrintTicket and commit the set operation.
            '

            queue.UserPrintTicket = result.ValidatedPrintTicket
            queue.Commit()
            Console.WriteLine("PrintTicket new duplexing setting is set on '{0}'.", queue.FullName)
        Else
            '

            ' The duplexing option we want to set has been changed by the validation process
            ' when it was resolving setting constraints.
            '

            Console.WriteLine("PrintTicket new duplexing setting is constrained on '{0}'.", queue.FullName)
        End If
    Else
        '

        ' If the printer doesn't support the duplexing option we want to set, skip it.
        '

        Console.WriteLine("PrintTicket new duplexing setting is not supported on '{0}'.", queue.FullName)
    End If
End Sub

```

So that you can quickly test this example, the remainder of it is presented below. Create a project and a namespace and then paste both the code snippets in this article into the namespace block.

```

/// <summary>
/// Displays the correct command line syntax to run this sample program.
/// </summary>
static private void DisplayUsage()

```

```

{
    Console.WriteLine();
    Console.WriteLine("Usage #1: printticket.exe -l \"<printer_name>\"");
    Console.WriteLine("      Run program on the specified local printer");
    Console.WriteLine();
    Console.WriteLine("      Quotation marks may be omitted if there are no spaces in printer_name.");
    Console.WriteLine();
    Console.WriteLine("Usage #2: printticket.exe -r \"\\\\\\<server_name>\\\\<printer_name>\"");
    Console.WriteLine("      Run program on the specified network printer");
    Console.WriteLine();
    Console.WriteLine("      Quotation marks may be omitted if there are no spaces in server_name or
printer_name.");
    Console.WriteLine();
    Console.WriteLine("Usage #3: printticket.exe -a");
    Console.WriteLine("      Run program on all installed printers");
    Console.WriteLine();
}

[STAThread]
static public void Main(string[] args)
{
    try
    {
        if ((args.Length == 1) && (args[0] == "-a"))
        {
            //
            // Change PrintTicket setting for all local and network printer connections.
            //
            LocalPrintServer server = new LocalPrintServer();

            EnumeratedPrintQueueTypes[] queue_types = {EnumeratedPrintQueueTypes.Local,
                                                       EnumeratedPrintQueueTypes.Connections};

            //
            // Enumerate through all the printers.
            //
            foreach (PrintQueue queue in server.GetPrintQueues(queue_types))
            {
                //
                // Change the PrintTicket setting queue by queue.
                //
                ChangePrintTicketSetting(queue);
            }
        }//end if -a

        else if ((args.Length == 2) && (args[0] == "-l"))
        {
            //
            // Change PrintTicket setting only for the specified local printer.
            //
            LocalPrintServer server = new LocalPrintServer();
            PrintQueue queue = new PrintQueue(server, args[1]);
            ChangePrintTicketSetting(queue);
        }//end if -l

        else if ((args.Length == 2) && (args[0] == "-r"))
        {
            //
            // Change PrintTicket setting only for the specified remote printer.
            //
            String serverName = args[1].Remove(args[1].LastIndexOf(@"\""));
            String printerName = args[1].Remove(0, args[1].LastIndexOf(@"\") + 1);
            PrintServer ps = new PrintServer(serverName);
            PrintQueue queue = new PrintQueue(ps, printerName);
            ChangePrintTicketSetting(queue);
        }//end if -r
    }
}

```

```

        else
        {
            //
            // Unrecognized command line.
            // Show user the correct command line syntax to run this sample program.
            //
            DisplayUsage();
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);

        //
        // Show inner exception information if it's provided.
        //
        if (e.InnerException != null)
        {
            Console.WriteLine("--- Inner Exception ---");
            Console.WriteLine(e.InnerException.Message);
            Console.WriteLine(e.InnerException.StackTrace);
        }
    }
    finally
    {
        Console.WriteLine("Press Return to continue...");
        Console.ReadLine();
    }
}
//end Main

```

```

''' <summary>
''' Displays the correct command line syntax to run this sample program.
''' </summary>
Private Shared Sub DisplayUsage()
    Console.WriteLine()
    Console.WriteLine("Usage #1: printticket.exe -l ""<printer_name>""")
    Console.WriteLine("      Run program on the specified local printer")
    Console.WriteLine()
    Console.WriteLine("      Quotation marks may be omitted if there are no spaces in printer_name.")
    Console.WriteLine()
    Console.WriteLine("Usage #2: printticket.exe -r ""\<server_name>\<printer_name>""")
    Console.WriteLine("      Run program on the specified network printer")
    Console.WriteLine()
    Console.WriteLine("      Quotation marks may be omitted if there are no spaces in server_name or
printer_name.")
    Console.WriteLine()
    Console.WriteLine("Usage #3: printticket.exe -a")
    Console.WriteLine("      Run program on all installed printers")
    Console.WriteLine()
End Sub

<STAThread>
Public Shared Sub Main(ByVal args() As String)
    Try
        If (args.Length = 1) AndAlso (args(0) = "-a") Then
            '
            ' Change PrintTicket setting for all local and network printer connections.
            '
            Dim server As New LocalPrintServer()

            Dim queue_types() As EnumeratedPrintQueueTypes = {EnumeratedPrintQueueTypes.Local,
EnumeratedPrintQueueTypes.Connections}

            '
            ' Enumerate through all the printers.
            '

```

```

For Each queue As PrintQueue In server.GetPrintQueues(queue_types)
    '
    ' Change the PrintTicket setting queue by queue.
    '

    ChangePrintTicketSetting(queue)
    Next queue 'end if -a

ElseIf (args.Length = 2) AndAlso (args(0) = "-l") Then
    '
    ' Change PrintTicket setting only for the specified local printer.
    '

    Dim server As New LocalPrintServer()
    Dim queue As New PrintQueue(server, args(1))
    ChangePrintTicketSetting(queue) 'end if -l

ElseIf (args.Length = 2) AndAlso (args(0) = "-r") Then
    '
    ' Change PrintTicket setting only for the specified remote printer.
    '

    Dim serverName As String = args(1).Remove(args(1).LastIndexOf("\"))
    Dim printerName As String = args(1).Remove(0, args(1).LastIndexOf("\") + 1)
    Dim ps As New PrintServer(serverName)
    Dim queue As New PrintQueue(ps, printerName)
    ChangePrintTicketSetting(queue) 'end if -r

Else
    '
    ' Unrecognized command line.
    ' Show user the correct command line syntax to run this sample program.
    '

    DisplayUsage()
End If
Catch e As Exception
    Console.WriteLine(e.Message)
    Console.WriteLine(e.StackTrace)

    '
    ' Show inner exception information if it's provided.
    '

    If e.InnerException IsNot Nothing Then
        Console.WriteLine("--- Inner Exception ---")
        Console.WriteLine(e.InnerException.Message)
        Console.WriteLine(e.InnerException.StackTrace)
    End If
Finally
    Console.WriteLine("Press Return to continue...")
    Console.ReadLine()
End Try
End Sub

```

## See also

- [PrintCapabilities](#)
- [PrintTicket](#)
- [GetPrintQueues](#)
- [PrintServer](#)
- [EnumeratedPrintQueueTypes](#)
- [PrintQueue](#)
- [GetPrintCapabilities](#)
- [Documents in WPF](#)
- [Printing Overview](#)
- [Print Schema](#)



# Globalization and Localization

3/5/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides extensive support for the development of world-ready applications.

## In This Section

[WPF Globalization and Localization Overview](#)

[Globalization for WPF](#)

[Use Automatic Layout Overview](#)

[Localization Attributes and Comments](#)

[Bidirectional Features in WPF Overview](#)

[How-to Topics](#)

## Reference

[System.Globalization](#)

[FlowDirection](#)

[NeutralResourcesLanguageAttribute](#)

[xml:lang Handling in XAML](#)

## Related Sections

# WPF Globalization and Localization Overview

11/7/2019 • 15 minutes to read • [Edit Online](#)

When you limit your product's availability to only one language, you limit your potential customer base to a fraction of our world's 6.5 billion population. If you want your applications to reach a global audience, cost-effective localization of your product is one of the best and most economical ways to reach more customers.

This overview introduces globalization and localization in Windows Presentation Foundation (WPF). Globalization is the design and development of applications that perform in multiple locations. For example, globalization supports localized user interfaces and regional data for users in different cultures. WPF provides globalized design features, including automatic layout, satellite assemblies, and localized attributes and commenting.

Localization is the translation of application resources into localized versions for the specific cultures that the application supports. When you localize in WPF, you use the APIs in the [System.Windows.Markup.Localizer](#) namespace. These APIs power the [LocBaml Tool Sample](#) command-line tool. For information about how to build and use LocBaml, see [Localize an Application](#).

## Best Practices for Globalization and Localization in WPF

You can make the most of the globalization and localization functionality that is built into WPF by following the UI design and localization-related tips that this section provides.

### Best Practices for WPF UI Design

When you design a WPF-based UI, consider implementing these best practices:

- Write your UI in XAML; avoid creating UI in code. When you create your UI by using XAML, you expose it through built-in localization APIs.
- Avoid using absolute positions and fixed sizes to lay out content; instead, use relative or automatic sizing.
  - Use [SizeToContent](#) and keep widths and heights set to `Auto`.
  - Avoid using [Canvas](#) to lay out UIs.
  - Use [Grid](#) and its size-sharing feature.
- Provide extra space in margins because localized text often requires more space. Extra space allows for possible overhanging characters.
- Enable [TextWrapping](#) on [TextBlock](#) to avoid clipping.
- Set the `xml:lang` attribute. This attribute describes the culture of a specific element and its child elements. The value of this property changes the behavior of several features in WPF. For example, it changes the behavior of hyphenation, spell checking, number substitution, complex script shaping, and font fallback. See [Globalization for WPF](#) for more information about setting the [xml:lang Handling in XAML](#).
- Create a customized composite font to obtain better control of fonts that are used for different languages. By default, WPF uses the `GlobalUserInterface.composite` font in your `Windows\Fonts` directory.
- When you create navigation applications that may be localized in a culture that presents text in a right-to-left format, explicitly set the [FlowDirection](#) of every page to ensure the page does not inherit [FlowDirection](#) from the [NavigationWindow](#).
- When you create stand-alone navigation applications that are hosted outside a browser, set the [StartupUri](#)

for your initial application to a [NavigationWindow](#) instead of to a page (for example, `<Application StartupUri="NavigationWindow.xaml">`). This design enables you to change the [FlowDirection](#) of the Window and the navigation bar. For more information and an example, see [Globalization Homepage Sample](#).

## Best Practices for WPF Localization

When you localize WPF-based applications, consider implementing these best practices:

- Use localization comments to provide extra context for localizers.
- Use localization attributes to control localization instead of selectively omitting [Uid](#) properties on elements. See [Localization Attributes and Comments](#) for more information.
- Use `msbuild -t:updateuid` and `-t:checkuid` to add and check [Uid](#) properties in your XAML. Use [Uid](#) properties to track changes between development and localization. [Uid](#) properties help you localize new development changes. If you manually add [Uid](#) properties to a UI, the task is typically tedious and less accurate.
  - Do not edit or change [Uid](#) properties after you begin localization.
  - Do not use duplicate [Uid](#) properties (remember this tip when you use the copy-and-paste command).
  - Set the `UltimateResourceFallback` location in `AssemblyInfo.*` to specify the appropriate language for fallback (for example,  
`[assembly: NeutralResourcesLanguage("en-US", UltimateResourceFallbackLocation.Satellite)]`).

If you decide to include your source language in the main assembly by omitting the `<UICulture>` tag in your project file, set the `UltimateResourceFallback` location as the main assembly instead of the satellite (for example,  
`[assembly: NeutralResourcesLanguage("en-US", UltimateResourceFallbackLocation.MainAssembly)]`).

## Localize a WPF Application

When you localize a WPF application, you have several options. For example, you can bind the localizable resources in your application to an XML file, store localizable text in resx tables, or have your localizer use Extensible Application Markup Language (XAML) files. This section describes a localization workflow that uses the BAML form of XAML, which provides several benefits:

- You can localize after you build.
- You can update to a newer version of the BAML form of XAML with localizations from an older version of the BAML form of XAML so that you can localize at the same time that you develop.
- You can validate original source elements and semantics at compile time because the BAML form of XAML is the compiled form of XAML.

## Localization Build Process

When you develop a WPF application, the build process for localization is as follows:

- The developer creates and globalizes the WPF application. In the project file the developer sets `<UICulture>en-US</UICulture>` so that when the application is compiled, a language-neutral main assembly is generated. This assembly has a satellite .resources.dll file that contains all the localizable resources. Optionally, you can keep the source language in the main assembly because our localization APIs support extraction from the main assembly.
- When the file is compiled into the build, the XAML is converted to the BAML form of XAML. The culturally

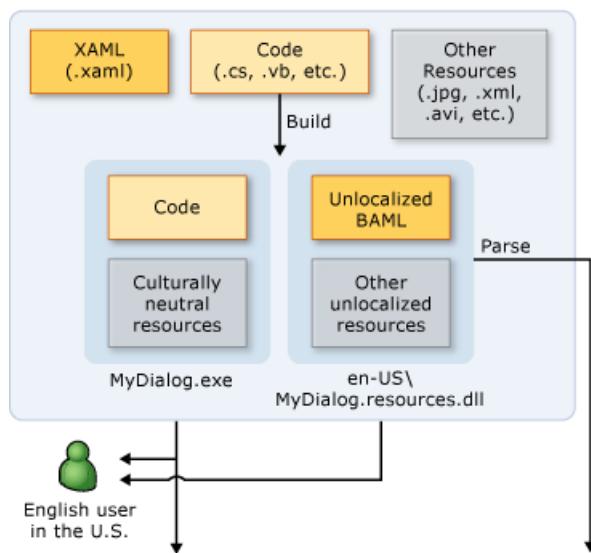
neutral `MyDialog.exe` and the culturally dependent (English) `MyDialog.resources.dll` files are released to the English-speaking customer.

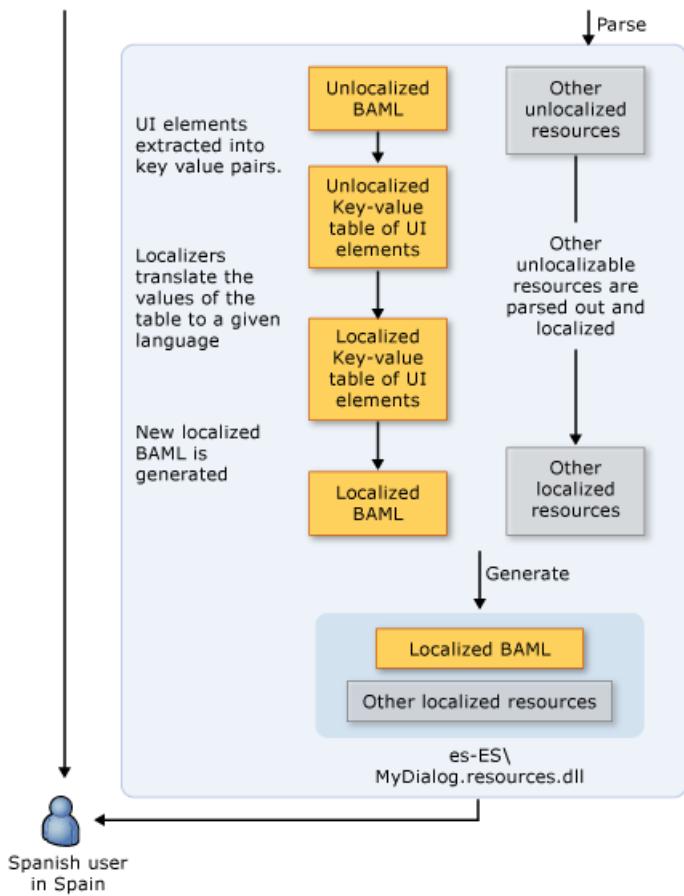
## Localization Workflow

The localization process begins after the unlocalized `MyDialog.resources.dll` file is built. The UI elements and properties in your original XAML are extracted from the BAML form of XAML into key-value pairs by using the APIs under `System.Windows.Markup.Localizer`. Localizers use the key-value pairs to localize the application. You can generate a new .resource.dll from the new values after localization is complete.

The keys of the key-value pairs are `x:Uid` values that are placed by the developer in the original XAML. These `x:Uid` values enable the API to track and merge changes that happen between the developer and the localizer during localization. For example, if the developer changes the UI after the localizer begins localizing, you can merge the development change with the already completed localization work so that minimal translation work is lost.

The following graphic shows a typical localization workflow that is based on the BAML form of XAML. This diagram assumes the developer writes the application in English. The developer creates and globalizes the WPF application. In the project file the developer sets `<UICulture>en-US</UICulture>` so that on build, a language neutral main assembly gets generated with a satellite .resources.dll containing all localizable resources. Alternately, one could keep the source language in the main assembly because WPF localization APIs support extraction from the main assembly. After the build process, the XAML get compiled into BAML. The culturally neutral `MyDialog.exe.resources.dll` get shipped to the English speaking customer.





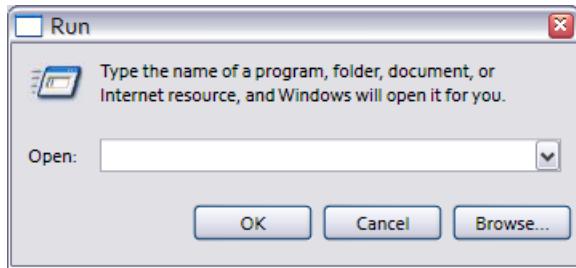
## Examples of WPF Localization

This section contains examples of localized applications to help you understand how to build and localize WPF applications.

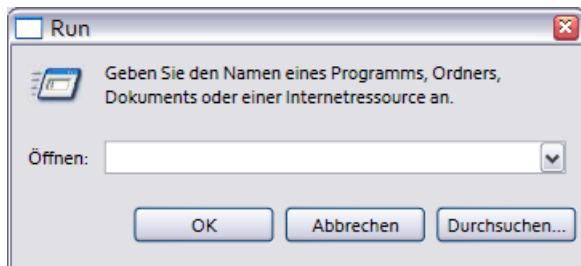
### Run Dialog Box Example

The following graphics show the output of the **Run** dialog box sample.

#### English:



#### German:



## Designing a Global Run Dialog Box

This example produces a **Run** dialog box by using WPF and XAML. This dialog box is equivalent to the **Run** dialog box that is available from the Microsoft Windows Start menu.

Some highlights for making global dialog boxes are:

## Automatic Layout

In Window1.xaml:

```
<Window SizeToContent="WidthAndHeight">
```

The previous Window property automatically resizes the window according to the size of the content. This property prevents the window from cutting off content that increases in size after localization; it also removes unneeded space when content decreases in size after localization.

```
<Grid x:Uid="Grid_1">
```

**Uid** properties are needed in order for WPF localization APIs to work correctly.

They are used by WPF localization APIs to track changes between the development and localization of the user interface (UI). **Uid** properties enable you to merge a newer version of the UI with an older localization of the UI. You add a **Uid** property by running `msbuild -t:updateuid RunDialog.csproj` in a command shell. This is the recommended method of adding **Uid** properties because manually adding them is typically time-consuming and less accurate. You can check that **Uid** properties are correctly set by running `msbuild -t:checkuid RunDialog.csproj`.

The UI is structured by using the **Grid** control, which is a useful control for taking advantage of the automatic layout in WPF. Note that the dialog box is split into three rows and five columns. Not one of the row and column definitions has a fixed size; hence, the UI elements that are positioned in each cell can adapt to increases and decreases in size during localization.

```
<Grid.ColumnDefinitions>
  <ColumnDefinition x:Uid="ColumnDefinition_1" />
  <ColumnDefinition x:Uid="ColumnDefinition_2" />
```

The first two columns where the **Open:** label and **ComboBox** are placed use 10 percent of the UI total width.

```
<ColumnDefinition x:Uid="ColumnDefinition_3" SharedSizeGroup="Buttons" />
<ColumnDefinition x:Uid="ColumnDefinition_4" SharedSizeGroup="Buttons" />
<ColumnDefinition x:Uid="ColumnDefinition_5" SharedSizeGroup="Buttons" />
</Grid.ColumnDefinitions>
```

Note that of the example uses the shared-sizing feature of **Grid**. The last three columns take advantage of this by placing themselves in the same **SharedSizeGroup**. As one would expect from the name of the property, this allows the columns to share the same size. So when the "Browse..." gets localized to the longer string "Durchsuchen...", all buttons grow in width instead of having a small "OK" button and a disproportionately large "Durchsuchen..." button.

## xml:lang

```
xml:lang="en-US"
```

Notice the **xml:lang Handling in XAML** placed at the root element of the UI. This property describes the culture of a given element and its children. This value is used by several features in WPF and should be changed appropriately during localization. This value changes what language dictionary is used to hyphenate and spell check words. It also affects the display of digits and how the font fallback system selects which font to use. Finally, the property affects the way numbers are displayed and the way texts written in complex scripts are shaped. The default value is "en-US".

## Building a Satellite Resource Assembly

In .csproj:

Edit the `.csproj` file and add the following tag to an unconditional `<PropertyGroup>`:

```
<UICulture>en-US</UICulture>
```

Notice the addition of a `uiculture` value. When this is set to a valid `CultureInfo` value such as `en-US`, building the project will generate a satellite assembly with all localizable resources in it.

```
<Resource Include="RunIcon.JPG">  
  <Localizable>False</Localizable>  
</Resource>
```

The `RunIcon.JPG` does not need to be localized because it should appear the same for all cultures. `Localizable` is set to `false` so that it remains in the language neutral main assembly instead of the satellite assembly. The default value of all noncompilable resources is `Localizable` set to `true`.

## Localizing the Run Dialog

### Parse

After building the application, the first step in localizing it is parsing the localizable resources out of the satellite assembly. For the purposes of this topic, use the sample LocBaml tool which can be found at [LocBaml Tool Sample](#). Note that LocBaml is only a sample tool meant to help you get started in building a localization tool that fits into your localization process. Using LocBaml, run the following to parse: **LocBaml /parse RunDialog.resources.dll /out:** to generate a "RunDialog.resources.dll.CSV" file.

### Localize

Use your favorite CSV editor that supports Unicode to edit this file. Filter out all entries with a localization category of "None". You should see the following entries:

RESOURCE KEY	LOCALIZATION CATEGORY	VALUE
Button_1:System.Windows.Controls.Button.\$Content	Button	OK
Button_2:System.Windows.Controls.Button.\$Content	Button	Cancel
Button_3:System.Windows.Controls.Button.\$Content	Button	Browse...
ComboBox_1:System.Windows.Controls.ComboBox.\$Content	ComboBox	
TextBlock_1:System.Windows.Controls.TextBlock.\$Content	Text	Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
TextBlock_2:System.Windows.Controls.TextBlock.\$Content	Text	Open:
Window_1:System.Windows.Window.Title	Title	Run

Localizing the application to German would require the following translations:

RESOURCE KEY	LOCALIZATION CATEGORY	VALUE
Button_1:System.Windows.Controls.Button.\$Content	Button	OK
Button_2:System.Windows.Controls.Button.\$Content	Button	Abbrechen
Button_3:System.Windows.Controls.Button.\$Content	Button	Durchsuchen...
ComboBox_1:System.Windows.Controls.ComboBox.\$Content	ComboBox	
TextBlock_1:System.Windows.Controls.TextBlock.\$Content	Text	Geben Sie den Namen eines Programms, Ordners, Dokuments oder einer Internetresource an.
TextBlock_2:System.Windows.Controls.TextBlock.\$Content	Text	Öffnen:
Window_1:System.Windows.Window.Title	Title	Run

## Generate

The last step of localization involves creating the newly localized satellite assembly. This can be accomplished with the following LocBaml command:

**LocBaml.exe /generate RunDialog.resources.dll /trans:RunDialog.resources.dll.CSV /out: ./cul:de-DE**

On German Windows, if this resources.dll is placed in a de-DE folder next to the main assembly, this resource will automatically load instead of the one in the en-US folder. If you do not have a German version of Windows to test this, set the culture to whatever culture of Windows you are using (for example, `en-us`), and replace the original resources DLL.

## Satellite Resource Loading

MYDIALOG.EXE	EN-US\MYDIALOG.ROCESSES.DLL	DE-DE\MYDIALOG.ROCESSES.DLL
Code	Original English BAML	Localized BAML
Culturally neutral resources	Other resources in English	Other resources localized to German

The .NET framework automatically chooses which satellite resources assembly to load based on the application's `Thread.CurrentThread.CurrentCulture`. This defaults to the culture of your Windows OS. So if you are using German Windows, the de-DE\MyDialog.resources.dll loads, if you are using English Windows, the en-US\MyDialog.resources.dll loads. You can set the ultimate fallback resource for your application by specifying the `NeutralResourcesLanguage` in your project's `AssemblyInfo.*`. For example if you specify:

```
[assembly: NeutralResourcesLanguage("en-US", UltimateResourceFallbackLocation.Satellite)]
```

then the en-US\MyDialog.resources.dll will be used with German Windows if a de-DE\MyDialog.resources.dll or de\MyDialog.resources.dll are both unavailable.

## Microsoft Saudi Arabia Homepage

The following graphics show an English and Arabic Homepage. For the complete sample that produces these

graphics see [Globalization Homepage Sample](#).

**English:**

The screenshot shows the Microsoft homepage for Saudi Arabia, viewed through a web browser window. The page is primarily in English, with some Arabic text visible in the top navigation bar and sidebar.

**Top Navigation:** Microsoft United States | Microsoft Worldwide | Home | Search | Microsoft.com | msn Web Search

**Sidebar (Left):**

- Select your Location | عربي
- Product Families:**
  - Windows
  - Office
  - Windows Server System
  - Developer Tools
  - Business Solutions
  - Windows Mobile
  - Games
  - Hardware
  - MSN Arabia Services
  - Arabic Development & Support
- Resources:**
  - Events
  - Security
  - Support
  - Downloads
  - Windows Update
  - Communities
  - Learning Tools
  - Licensing
  - Careers
  - Find a Partner

**Main Content Area:** Windows Vista

**Bottom Navigation (Footer):**

- Popular Destinations for your home:**
  - [Windows Vista](#)
  - [Windows Mobile](#)
  - [Clip Art](#)
- Business Solutions:**
  - [CRM](#)
  - [Analytics & Reporting](#)
  - [Financial management](#)
- Security:**
  - [What You Should Know About Zotob](#)
  - [Protect Your PC](#)
  - [Windows XP Service Pack 2](#)
  - [Security Home](#)

**Arabic:**



## Designing a Global Microsoft home page

This mock up of the Microsoft Saudi Arabia web site illustrates the globalization features provided for RightToLeft languages. Languages such as Hebrew and Arabic have a right-to-left reading order so the layout of UI must often be laid out quite differently than it would be in left-to-right languages such as English. Localizing from a left-to-right language to a right-to-left language or vice versa can be quite challenging. WPF has been designed to make such localizations much easier.

### FlowDirection

*Homepage.xaml:*

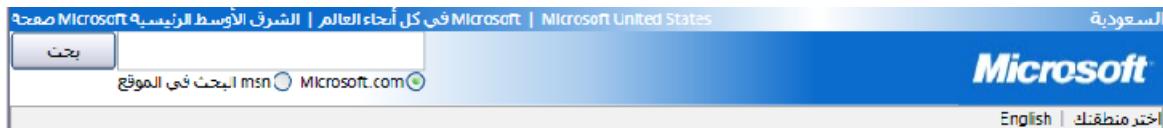
```
<Page x:Uid="Page_1" x:Class="MicrosoftSaudiArabiaHomepage.Homepage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      FlowDirection="LeftToRight"
      Localization.Comments="FlowDirection(This FlowDirection controls the actual content of the homepage)"
      xml:lang="en-US">
```

Notice the **FlowDirection** property on **Page**. Changing this property to **RightToLeft** will change the **FlowDirection** of the **Page** and its children elements so that the layout of this UI is flipped to become right-to-left as an Arabic user would expect. One can override the inheritance behavior by specifying an explicit **FlowDirection** on any element. The **FlowDirection** property is available on any **FrameworkElement** or document related element and has an implicit value of **LeftToRight**.

Observe that even the background gradient brushes are flipped correctly when the root **FlowDirection** is changed:

**FlowDirection="LeftToRight"**

**FlowDirection="RightToLeft"**



### Avoid Using Fixed Dimensions for Panels and Controls

Take a look through Homepage.xaml, notice that aside from the fixed width and height specified for the entire UI on the top **DockPanel**, there are no other fixed dimensions. Avoid using fixed dimensions to prevent clipping localized text that may be longer than the source text. WPF panels and controls will automatically resize based on the content that they contain. Most controls also have minimum and maximum dimensions that you can set for more control (for example, `MinWidth="20"`). With **Grid**, you can also set relative widths and heights by using '\*' (for example, `Width="0.25*"`) or use its cell size sharing feature.

### Localization Comments

There are many cases where content may be ambiguous and difficult to translate. The developer or designer has the ability to provide extra context and comments to localizers through localization comments. For example the Localization.Comments below clarifies the usage of the character '|'.

```
<TextBlock
    x:Uid="TextBlock_2"
    DockPanel.Dock="Right"
    Foreground="White"
    Margin="5,0,5,0"
    Localization.Comments="$Content(This character is used as a decorative rule.)">
    |
</TextBlock>
```

This comment becomes associated with TextBlock\_1's content and in the case of the LocBaml Tool, ( see [Localize an Application](#)), it can be seen in the 6th column of the TextBlock\_1 row in the output.csv file:

RESOURCE KEY	CATEGORY	READABLE	MODIFIABLE	COMMENT	VALUE
TextBlock_1:System.Windows.Controls.TextBlock.\$Content	Text	TRUE	TRUE	This character is used as a decorative rule.	

Comments can be placed on the content or property of any element using the following syntax:

```
<TextBlock
    x:Uid="TextBlock_1"
    DockPanel.Dock="Right"
    Foreground="White"
    Margin="5,0,5,0"
    Localization.Comments="$Content(This is a comment on the TextBlock's content.)"
    Margin(This is a comment on the TextBlock's Margin property.)">
    |
</TextBlock>
```

### Localization Attributes

Often the developer or localization manager needs control of what localizers can read and modify. For example, you might not want the localizer to translate the name of your company or legal wording. WPF provides attributes that enable you to set the readability, modifiability, and category of an element's content or property which your localization tool can use to lock, hide, or sort elements. For more information, see [Attributes](#). For the purposes of this sample, the LocBaml Tool just outputs the values of these attributes. WPF controls all have default values for these attributes, but you can override them. For example, the following example overrides the default localization attributes for `TextBlock_1` and sets the content to be readable but unmodifiable for localizers.

```
<TextBlock  
x:Uid="TextBlock_1"  
Localization.Attributes=  
"$Content(Readable Unmodifiable)">  
    Microsoft Corporation  
</TextBlock>
```

In addition to the readability and modifiability attributes, WPF provides an enumeration of common UI categories ([LocalizationCategory](#)) that can be used to give localizers more context. The WPF default categories for platform controls can be overridden in XAML as well:

```
<TextBlock x:Uid="TextBlock_2">  
<TextBlock.ToolTip>  
<TextBlock  
x:Uid="TextBlock_3"  
Localization.Attributes=  
"$Content(ToolTip Readable Unmodifiable)">  
    Microsoft Corporation  
</TextBlock>  
</TextBlock.ToolTip>  
Windows Vista  
</TextBlock>
```

The default localization attributes that WPF provides can also be overridden through code, so you can correctly set the right default values for custom controls. For example:

```
[Localizability(Readability = Readability.Readable, Modifiability=Modifiability.Unmodifiable,  
LocalizationCategory.None)]  
public class CorporateLogo : TextBlock  
{  
    // ...  
}
```

The per instance attributes set in XAML will take precedence over the values set in code on custom controls. For more information on attributes and comments, see [Localization Attributes and Comments](#).

## Font Fallback and Composite Fonts

If you specify a font that does not support a given codepoint range, WPF will automatically fallback to one that does by using the Global User Interface.compositefont that is located in your Windows\Fonts directory. Composite fonts work just as any other font and can be used explicitly by setting an element's `FontFamily` (for instance, `FontFamily="Global User Interface"`). You can specify your own font fallback preference by creating your own composite font and specifying what font to use for specific codepoint ranges and languages.

For more information on composite fonts see [FontFamily](#).

## Localizing the Microsoft Homepage

You can follow the same steps as the Run Dialog example to localize this application. The localized .csv file for Arabic is available for you in the [Globalization Homepage Sample](#).



# Globalization for WPF

11/7/2019 • 6 minutes to read • [Edit Online](#)

This topic introduces issues that you should be aware of when writing Windows Presentation Foundation (WPF) applications for the global market. The globalization programming elements are defined in .NET in the [System.Globalization](#) namespace.

## XAML Globalization

Extensible Application Markup Language (XAML) is based on XML and takes advantage of the globalization support defined in the XML specification. The following sections describe some XAML features that you should be aware of.

### Character References

A character reference gives the UTF16 code unit of the particular Unicode character it represents, in either decimal or hexadecimal. The following example shows a decimal character reference for the COPTIC CAPITAL LETTER HORI, or 'ؒ':

```
&#1000;
```

The following example shows a hexadecimal character reference. Notice that it has an **x** in front of the hexadecimal number.

```
&x3E8;
```

### Encoding

The encoding supported by XAML are ASCII, Unicode UTF-16, and UTF-8. The encoding statement is at the beginning of XAML document. If no encoding attribute exists and there is no byte-order, the parser defaults to UTF-8. UTF-8 and UTF-16 are the preferred encodings. UTF-7 is not supported. The following example demonstrates how to specify a UTF-8 encoding in a XAML file.

```
?xml encoding="UTF-8"?
```

### Language Attribute

XAML uses [xml:lang](#) to represent the language attribute of an element. To take advantage of the [CultureInfo](#) class, the language attribute value needs to be one of the culture names predefined by [CultureInfo.xml:lang](#) is inheritable in the element tree (by XML rules, not necessarily because of dependency property inheritance) and its default value is an empty string if it is not assigned explicitly.

The language attribute is very useful for specifying dialects. For example, French has different spelling, vocabulary, and pronunciation in France, Quebec, Belgium, and Switzerland. Also Chinese, Japanese, and Korean share code points in Unicode, but the ideographic shapes are different and they use totally different fonts.

The following Extensible Application Markup Language (XAML) example uses the `fr-CA` language attribute to specify Canadian French.

```
<TextBlock xml:lang="fr-CA">Découvrir la France</TextBlock>
```

## Unicode

XAML supports all Unicode features including surrogates. As long as the character set can be mapped to Unicode, it is supported. For example, GB18030 introduces some characters that are mapped to the Chinese, Japanese, and Korean (CFK) extension A and B and surrogate pairs, therefore it is fully supported. A WPF application can use [StringInfo](#) to manipulate strings without understanding whether they have surrogate pairs or combining characters.

# Designing an International User Interface with XAML

This section describes user interface (UI) features that you should consider when writing an application.

## International Text

WPF includes built-in processing for all Microsoft .NET Framework supported writing systems.

The following scripts are currently supported:

- Arabic
- Bengali
- Devanagari
- Cyrillic
- Greek
- Gujarati
- Gurmukhi
- Hebrew
- Ideographic scripts
- Kannada
- Lao
- Latin
- Malayalam
- Mongolian
- Odia
- Syriac
- Tamil
- Telugu
- Thaana
- Thai\*
- Tibetan

\*In this release the display and editing of Thai text is supported; word breaking is not.

The following scripts are not currently supported:

- Khmer

- Korean Old Hangul
- Myanmar
- Sinhala

All the writing system engines support OpenType fonts. OpenType fonts can include the OpenType layout tables that enable font creators to design better international and high-end typographic fonts. The OpenType font layout tables contain information about glyph substitutions, glyph positioning, justification, and baseline positioning, enabling text-processing applications to improve text layout.

OpenType fonts allow the handling of large glyph sets using Unicode encoding. Such encoding enables broad international support as well as for typographic glyph variants.

WPF text rendering is powered by Microsoft ClearType sub-pixel technology that supports resolution independence. This significantly improves legibility and provides the ability to support high quality magazine style documents for all scripts.

### **International Layout**

WPF provides a very convenient way to support horizontal, bidirectional, and vertical layouts. In presentation framework the [FlowDirection](#) property can be used to define layout. The flow direction patterns are:

- *LeftToRight* - horizontal layout for Latin, East Asian and so forth.
- *RightToLeft* - bidirectional for Arabic, Hebrew and so forth.

## **Developing Localizable Applications**

When you write an application for global consumption you should keep in mind that the application must be localizable. The following topics point out things to consider.

### **Multilingual User Interface**

Multilingual User Interfaces (MUI) is a Microsoft support for switching UIs from one language to another. A WPF application uses the assembly model to support MUI. One application contains language-neutral assemblies as well as language-dependent satellite resource assemblies. The entry point is a managed .EXE in the main assembly. WPF resource loader takes advantage of the Framework's resource manager to support resource lookup and fallback. Multiple language satellite assemblies work with the same main assembly. The resource assembly that is loaded depends on the [CurrentUICulture](#) of the current thread.

### **Localizable User Interface**

WPF applications use XAML to define their UI. XAML allows developers to specify a hierarchy of objects with a set of properties and logic. The primary use of XAML is to develop WPF applications but it can be used to specify a hierarchy of any common language runtime (CLR) objects. Most developers use XAML to specify their application's UI and use a programming language such as C# to react to user interaction.

From a resource point of view, a XAML file designed to describe a language-dependent UI is a resource element and therefore its final distribution format must be localizable to support international languages. Because XAML cannot handle events many XAML applications contain blocks of code to do this. For more information, see [XAML Overview \(WPF\)](#). Code is stripped out and compiled into different binaries when a XAML file is tokenized into the BAML form of XAML. The BAML form of XAML files, images, and other types of managed resource objects are embedded in the satellite resource assembly, which can be localized into other languages, or the main assembly when localization is not required.

#### **NOTE**

WPF applications support all the FrameworkCLR resources including string tables, images, and so forth.

## Building Localizable Applications

Localization means to adapt a UI to different cultures. To make a WPF application localizable, developers need to build all the localizable resources into a resource assembly. The resource assembly is localized into different languages, and the code-behind uses resource management API to load. One of the files required for a WPF application is a project file (.proj). All resources that you use in your application should be included in the project file. The following example from a .csproj file shows how to do this.

```
<Resource Include="data\picture1.jpg"/>
<EmbeddedResource Include="data\stringtable.en-US.restext"/>
```

To use a resource in your application instantiate a [ResourceManager](#) and load the resource you want to use. The following example demonstrates how to do this.

```
void OnClick(object sender, RoutedEventArgs e)
{
    ResourceManager rm = new ResourceManager ("MySampleApp.data.stringtable",
        Assembly.GetExecutingAssembly());
    Text1.Text = rm.GetString("Message");
}
```

## Using ClickOnce with Localized Applications

ClickOnce is a new Windows Forms deployment technology that will ship with Visual Studio 2005. It enables application installation and upgrading of Web applications. When an application that was deployed with ClickOnce is localized it can only be viewed on the localized culture. For example, if a deployed application is localized to Japanese it can only be viewed on Japanese Microsoft Windows not on English Windows. This presents a problem because it is a common scenario for Japanese users to run an English version of Windows.

The solution to this problem is setting the neutral language fallback attribute. An application developer can optionally remove resources from the main assembly and specify that the resources can be found in a satellite assembly corresponding to a specific culture. To control this process use the [NeutralResourcesLanguageAttribute](#). The constructor of the [NeutralResourcesLanguageAttribute](#) class has two signatures, one that takes an [UltimateResourceFallbackLocation](#) parameter to specify the location where the [ResourceManager](#) should extract the fallback resources: main assembly or satellite assembly. The following example shows how to use the attribute. For the ultimate fallback location, the code causes the [ResourceManager](#) to look for the resources in the "de" subdirectory of the directory of the currently executing assembly.

```
[assembly: NeutralResourcesLanguageAttribute(
    "de" , UltimateResourceFallbackLocation.Satellite)]
```

## See also

- [WPF Globalization and Localization Overview](#)

# Use Automatic Layout Overview

10/12/2019 • 4 minutes to read • [Edit Online](#)

This topic introduces guidelines for developers on how to write Windows Presentation Foundation (WPF) applications with localizable user interfaces (UIs). In the past, localization of a UI was a time consuming process. Each language that the UI was adapted for required a pixel by pixel adjustment. Today with the right design and right coding standards, UIs can be constructed so that localizers have less resizing and repositioning to do. The approach to writing applications that can be more easily resized and repositioned is called automatic layout, and can be achieved by using WPF application design.

## Advantages of Using Automatic Layout

Because the WPF presentation system is powerful and flexible, it provides the ability to layout elements in an application that can be adjusted to fit the requirements of different languages. The following list points out some of the advantages of automatic layout.

- UI displays well in any language.
- Reduces the need to readjust position and size of controls after text is translated.
- Reduces the need to readjust window size.
- UI layout renders properly in any language.
- Localization can be reduced to the point that it is little more than string translation.

## Automatic Layout and Controls

Automatic layout enables an application to adjust the size of a control automatically. For example, a control can change to accommodate the length of a string. This capability enables localizers to translate the string; they no longer need to resize the control to fit the translated text. The following example creates a button with English content.

```
<Window  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    x:Class="ButtonLoc.Pane1"  
    Name="myWindow"  
    SizeToContent="WidthAndHeight"  
    >  
  
<DockPanel>  
    <Button FontSize="28" Height="50">My name is Hope.</Button>  
</DockPanel>  
</Window>
```

In the example, all you have to do to make a Spanish button is change the text. For example,

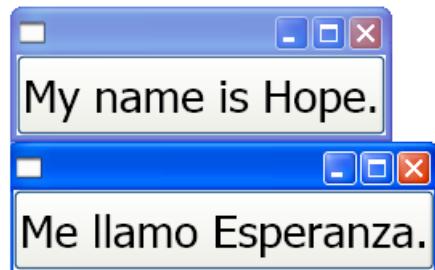
```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ButtonLoc.Pane1"
    Name="myWindow"
    SizeToContent="WidthAndHeight"
    >

    <DockPanel>
        <Button FontSize="28" Height="50">Me llamo Esperanza.</Button>
    </DockPanel>
</Window>

```

The following graphic shows the output of the code samples:



## Automatic Layout and Coding Standards

Using the automatic layout approach requires a set of coding and design standards and rules to produce a fully localizable UI. The following guidelines will aid your automatic layout coding.

### **Do not use absolute positions**

- Do not use [Canvas](#) because it positions elements absolutely.
- Use [DockPanel](#), [StackPanel](#), and [Grid](#) to position controls.

For a discussion about various types of panels, see [Panels Overview](#).

### **Do not set a fixed size for a window**

- Use [Window.SizeToContent](#). For example:

```

<StackPanel
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="GridLoc.Pane1"
    >

```

### [Add a FlowDirection](#)

- Add a [FlowDirection](#) to the root element of your application.

WPF provides a convenient way to support horizontal, bidirectional, and vertical layouts. In presentation framework, the [FlowDirection](#) property can be used to define layout. The flow-direction patterns are:

- [FlowDirection.LeftToRight](#) (LrTb) — horizontal layout for Latin, East Asian, and so forth.
- [FlowDirection.RightToLeft](#) (RITb) — bidirectional for Arabic, Hebrew, and so forth.

### [Use composite fonts instead of physical fonts](#)

- With composite fonts, the [FontFamily](#) property does not need to be localized.
- Developers can use one of the following fonts or create their own.
  - Global User Interface
  - Global San Serif
  - Global Serif

### Add `xml:lang`

- Add the `xml:lang` attribute in the root element of your UI, such as `xml:lang="en-US"` for an English application.
- Because composite fonts use `xml:lang` to determine what font to use, set this property to support multilingual scenarios.

## Automatic Layout and Grids

The [Grid](#) element, is useful for automatic layout because it enables a developer to position elements. A [Grid](#) control is capable of distributing the available space among its child elements, using a column and row arrangement. The UI elements can span multiple cells, and it is possible to have grids within grids. Grids are useful because they enable you to create and position complex UI. The following example demonstrates using a grid to position some buttons and text. Notice that the height and width of the cells are set to [Auto](#); therefore, the cell that contains the button with an image adjusts to fit the image.

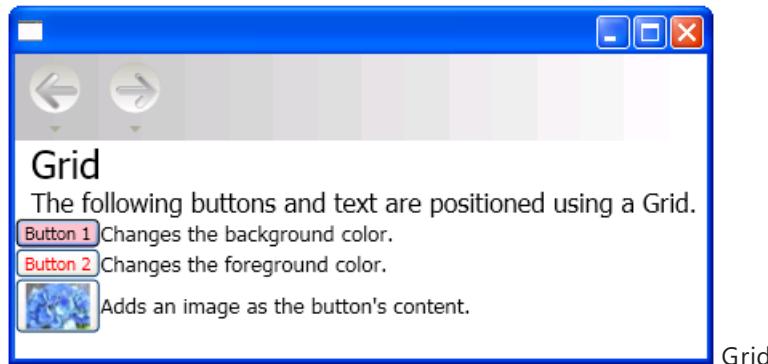
```

<Grid Name="grid" ShowGridLines ="false">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>

<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="0" FontSize="24">Grid
</TextBlock>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="1" FontSize="12"
    Grid.ColumnSpan="2">The following buttons and text are positioned using a Grid.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="2" Background="Pink"
    BorderBrush="Black" BorderThickness="10">Button 1
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="2" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the background
    color.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="3" Foreground="Red">
    Button 2
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="3" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the foreground
    color.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="4">
    <Image Source="data\flower.jpg"/>
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="4" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Adds an image as
    the button's content.
</TextBlock>
</Grid>

```

The following graphic shows the grid produced by the previous code.



## Automatic Layout and Grids Using the IsSharedSizeScope Property

A [Grid](#) element is useful in localizable applications to create controls that adjust to fit content. However, at times you want controls to maintain a particular size regardless of content. For example, if you have "OK", "Cancel" and "Browse" buttons you probably do not want the buttons sized to fit the content. In this case the [Grid.IsSharedSizeScope](#) attached property is useful for sharing the same sizing among multiple grid elements. The following example demonstrates how to share column and row sizing data between multiple [Grid](#) elements.

```

<StackPanel Orientation="Horizontal" DockPanel.Dock="Top">
    <Button Click="setTrue" Margin="0,0,10,10">Set IsSharedSizeScope="True"</Button>
    <Button Click="setFalse" Margin="0,0,10,10">Set IsSharedSizeScope="False"</Button>
</StackPanel>

<StackPanel Orientation="Horizontal" DockPanel.Dock="Top">

    <Grid ShowGridLines="True" Margin="0,0,10,0">
        <Grid.ColumnDefinitions>
            <ColumnDefinition SharedSizeGroup="FirstColumn"/>
            <ColumnDefinition SharedSizeGroup="SecondColumn"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" SharedSizeGroup="FirstRow"/>
        </Grid.RowDefinitions>

        <Rectangle Fill="Silver" Grid.Column="0" Grid.Row="0" Width="200" Height="100"/>
        <Rectangle Fill="Blue" Grid.Column="1" Grid.Row="0" Width="150" Height="100"/>

        <TextBlock Grid.Column="0" Grid.Row="0" FontWeight="Bold">First Column</TextBlock>
        <TextBlock Grid.Column="1" Grid.Row="0" FontWeight="Bold">Second Column</TextBlock>
    </Grid>

    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition SharedSizeGroup="FirstColumn"/>
            <ColumnDefinition SharedSizeGroup="SecondColumn"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" SharedSizeGroup="FirstRow"/>
        </Grid.RowDefinitions>

        <Rectangle Fill="Silver" Grid.Column="0" Grid.Row="0"/>
        <Rectangle Fill="Blue" Grid.Column="1" Grid.Row="0"/>

        <TextBlock Grid.Column="0" Grid.Row="0" FontWeight="Bold">First Column</TextBlock>
        <TextBlock Grid.Column="1" Grid.Row="0" FontWeight="Bold">Second Column</TextBlock>
    </Grid>
</StackPanel>

<TextBlock Margin="10" DockPanel.Dock="Top" Name="txt1"/>

```

#### NOTE

For the complete code sample, see [Share Sizing Properties Between Grids](#).

## See also

- [Globalization for WPF](#)
- [Use Automatic Layout to Create a Button](#)
- [Use a Grid for Automatic Layout](#)

# Localization Attributes and Comments

10/7/2019 • 3 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) localization comments are properties, inside XAML source code, supplied by developers to provide rules and hints for localization. Windows Presentation Foundation (WPF) localization comments contain two sets of information: localizability attributes and free-form localization comments. Localizability attributes are used by the WPF Localization API to indicate which resources are to be localized. Free-form comments are any information that the application author wants to include.

## Localization Comments

If markup application authors have requirements for specific elements in XAML, such as constraints on text length, font family, or font size, they can convey this information to localizers with comments in the XAML code. The process for adding comments to source code is as follows:

1. Application developer adds localization comments to XAML source code.
2. During the build process, you can specify in the .proj file whether to leave the free-form localization comments in the assembly, strip out part of the comments, or strip out all the comments. The stripped-out comments are placed in a separate file. You specify your option using a `<LocalizationDirectivesToLocFile>` tag, eg:

```
<LocalizationDirectivesToLocFile> value </LocalizationDirectivesToLocFile>
```
3. The values that can be assigned are:
  - **None** - Both comments and attributes stay inside the assembly and no separate file is generated.
  - **CommentsOnly** - Strips only the comments from the assembly and places them in the separate LocFile.
  - **All** - Strips both the comments and the attributes from the assembly and places them both in a separate LocFile.
4. When localizable resources are extracted from BAML, the localizability attributes are respected by the BAML Localization API.
5. Localization comment files, containing only free-form comments, are incorporated into the localization process at a later time.

The following example shows how to add localization comments to a XAML file.

```
<TextBlock x:Id = "text01"

FontFamily = "Microsoft Sans Serif"

FontSize = "12"

Localization.Attributes = "$Content (Unmodifiable Readable Text)

FontFamily (Unmodifiable Readable)"

Localization.Comments = "$Content (Trademark)

FontSize (Trademark font size)" >
```

```
Microsoft
```

```
</TextBlock>
```

In the previous sample the Localization.Attributes section contains the localization attributes and the Localization.Comments section the free-form comments. The following tables show the attributes and comments and their meaning to the localizer.

LOCALIZATION ATTRIBUTES	MEANING
\$Content (Unmodifiable Readable Text)	Contents of the TextBlock element cannot be modified. Localizers cannot change the word "Microsoft". The content is visible (Readable) to the localizer. The category of the content is text.
LOCALIZATION FREE-FORM COMMENTS	MEANING
FontFamily (Unmodifiable Readable)	The font family property of the TextBlock element cannot be changed but it is visible to the localizer.
FontSize (Trademark font size)	The application author indicates that the font size property should follow the standard trademark size.

## Localizability Attributes

The information in Localization.Attributes contains a list of pairs: the targeted value name and the associated localizability values. The target name can be a property name or the special \$Content name. If it is a property name, the targeted value is the value of the property. If it is \$Content, the target value is the content of the element.

There are three types of attributes:

- **Category**. This specifies whether a value should be modifiable from a localizer tool. See [Category](#).
- **Readability**. This specifies whether a localizer tool should read (and display) a value. See [Readability](#).
- **Modifiability**. This specifies whether a localizer tool allows a value to be modified. See [Modifiability](#).

These attributes can be specified in any order delimited by a space. In case duplicate attributes are specified, the last attribute will override former ones. For example, Localization.Attributes = "Unmodifiable Modifiable" sets Modifiability to Modifiable because it is the last value.

Modifiability and Readability are self-explanatory. The Category attribute provides predefined categories that help the localizer when translating text. Categories, such as, Text, Label, and Title give the localizer information about how to translate the text. There are also special categories: None, Inherit, Ignore, and NeverLocalize.

The following table shows the meaning of the special categories.

CATEGORY	MEANING
None	Targeted value has no defined category.
Inherit	Targeted value inherits its category from its parent.

CATEGORY	MEANING
Ignore	Targeted value is ignored in the localization process. Ignore affects only the current value. It will not affect child nodes.
NeverLocalize	Current value cannot be localized. This category is inherited by the children of an element.

## Localization Comments

Localization.Comments contains free-form strings concerning the targeted value. Application developers can add information to give localizers hints about how the applications text should be translated. The format of the comments can be any string surrounded by "()". Use '\' to escape characters.

## See also

- [Globalization for WPF](#)
- [Use Automatic Layout to Create a Button](#)
- [Use a Grid for Automatic Layout](#)
- [Localize an Application](#)

# Bidirectional Features in WPF Overview

11/7/2019 • 13 minutes to read • [Edit Online](#)

Unlike any other development platform, WPF has many features that support rapid development of bidirectional content, for example, mixed left to right and right to left data in the same document. At the same time, WPF creates an excellent experience for users who require bidirectional features such as Arabic and Hebrew speaking users.

The following sections explain many bidirectional features together with examples illustrating how to achieve the best display of bidirectional content. Most of the samples use XAML, though you can easily apply the concepts to C# or Microsoft Visual Basic code.

## FlowDirection

The basic property that defines the content flow direction in a WPF application is [FlowDirection](#). This property can be set to one of two enumeration values, [LeftToRight](#) or [RightToLeft](#). The property is available to all WPF elements that inherit from [FrameworkElement](#).

The following examples set the flow direction of a [TextBox](#) element.

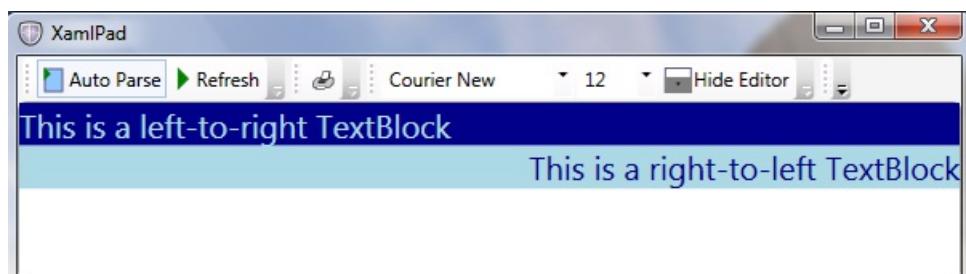
### Left-to-right flow direction

```
<TextBlock Background="DarkBlue" Foreground="LightBlue"
FontSize="20" FlowDirection="LeftToRight">
    This is a left-to-right TextBlock
</TextBlock>
```

### Right-to-left flow direction

```
<TextBlock Background="LightBlue" Foreground="DarkBlue"
FontSize="20" FlowDirection="RightToLeft">
    This is a right-to-left TextBlock
</TextBlock>
```

The following graphic shows how the previous code renders.



An element within a user interface (UI) tree will inherit the [FlowDirection](#) from its container. In the following example, the [TextBlock](#) is inside a [Grid](#), which resides in a [Window](#). Setting the [FlowDirection](#) for the [Window](#) implies setting it for the [Grid](#) and [TextBlock](#) as well.

The following example demonstrates setting [FlowDirection](#).

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="FlowDirectionApp.Window1"
    Title="BidiFeatures" Height="200" Width="700"
    FlowDirection="RightToLeft">

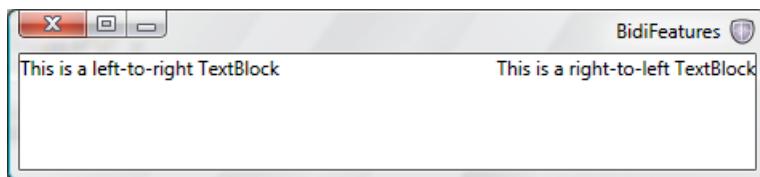
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Column="0" >
            This is a right-to-left TextBlock
        </TextBlock>

        <TextBlock Grid.Column="1" FlowDirection="LeftToRight">
            This is a left-to-right TextBlock
        </TextBlock>
    </Grid>
</Window>

```

The top level [Window](#) has a [RightToLeftFlowDirection](#), so all elements contained within it also inherit the same [FlowDirection](#). For an element to override a specified [FlowDirection](#) it must add an explicit direction change such as the second [TextBlock](#) in the previous example which changes to [LeftToRight](#). When no [FlowDirection](#) is defined, the default [LeftToRight](#) applies.

The following graphic shows the output of the previous example:



## FlowDocument

Many development platforms such as HTML, Win32 and Java provide special support for bidirectional content development. Markup languages such as HTML give content writers the necessary markup to display text in any required direction, for example the HTML 4.0 tag, "dir" that takes "rtl" or "ltr" as values. This tag is similar to the [FlowDirection](#) property, but the [FlowDirection](#) property works in a more advanced way to layout textual content and can be used for content other than text.

In WPF, a [FlowDocument](#) is a versatile UI element that can host a combination of text, tables, images and other elements. The samples in the following sections use this element.

Adding text to a [FlowDocument](#) can be done in more than one way. A simple way to do so is through a [Paragraph](#) which is a block-level element used to group content such as text. To add text to inline-level elements the samples use [Span](#) and [Run](#). [Span](#) is an inline-level flow content element used for grouping other inline elements, while a [Run](#) is an inline-level flow content element intended to contain a run of unformatted text. A [Span](#) can contain multiple [Run](#) elements.

The first document example contains a document that has a number of network share names; for example `\server1\folder\file.ext`. Whether you have this network link in an Arabic or English document, you always want it to appear in the same way. The following graphic illustrates using the [Span](#) element and shows the link in an Arabic [RightToLeft](#) document:

The screenshot shows the XamlPad application window. The status bar at the top displays "XamlPad". Below it are standard toolbar icons for "Auto Parse", "Refresh", "Save", "Font", "FontSize", "FontColor", "Courier New", "FontSize 12", "Hide Editor", and "100%". The main content area shows a single page of XAML code. The text in the XAML is displayed correctly in a right-to-left flow direction. A status message at the bottom of the editor window says "Done. Markup saved to 'C:\Program Files\XamlPad\XamlPad\_Saved.xaml'".

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">
    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" >
                ستجد الملف هنا:\\server1\filename\filename1.txt
                ثم باقى النص!
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>
```

Because the text is **RightToLeft**, all special characters, such as the "\", separate the text in a right to left order. That results in the link not being shown in the correct order, therefore to solve the problem, the text must be embedded to preserve a separate **Run** flowing **LeftToRight**. Instead of having a separate **Run** for each language, a better way to solve the problem is to embed the less frequently used English text into a larger Arabic **Span**.

The following graphic illustrates this by using the **Run** element embedded in a **Span** element:

The screenshot shows the XamlPad application window. The status bar at the top displays "XamlPad". Below it are standard toolbar icons for "Auto Parse", "Refresh", "Save", "Font", "FontSize", "FontColor", "Courier New", "FontSize 12", "Hide Editor", and "100%". The main content area shows a single page of XAML code. The text in the XAML is displayed correctly in a right-to-left flow direction, with the English path "\\server1\filename\filename1.txt" embedded within a **Run** element inside a **Span**. A status message at the bottom of the editor window says "Done. Markup saved to 'C:\Program Files\XamlPad\XamlPad\_Saved.xaml'".

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">
    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" >
                ستجد الملف هنا:\\server1\filename\filename1.txt
                <Run FlowDirection="LeftToRight">\\server1\filename\filename1.txt</Run>
                ثم باقى النص!
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>
```

The following example demonstrates using **Run** and **Span** elements in documents.

```

<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">

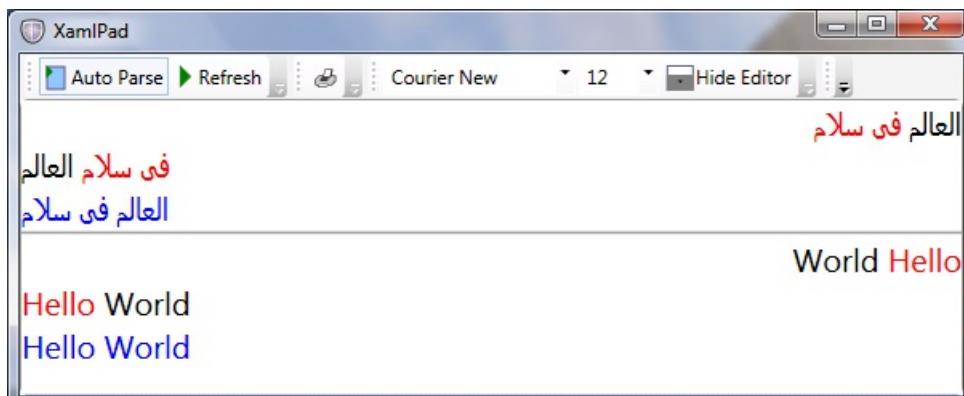
    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" >
                ستجد الملف هنا:
                <Run FlowDirection="LeftToRight">
                    \\server1\filename\filename1.txt</Run>
                ثم باقى النص!
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>

```

## Span Elements

The [Span](#) element works as a boundary separator between texts with different flow directions. Even [Span](#) elements with the same flow direction are considered to have different bidirectional scopes which means that the [Span](#) elements are ordered in the container's [FlowDirection](#), only the content within the [Span](#) element follows the [FlowDirection](#) of the [Span](#).

The following graphic shows the flow direction of several [TextBlock](#) elements.



The following example shows how to use the [Span](#) and [Run](#) elements to produce the results shown in the previous graphic.

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<StackPanel >

    <TextBlock FontSize="20" FlowDirection="RightToLeft">
        <Run FlowDirection="LeftToRight">العالم</Run>
        <Run FlowDirection="LeftToRight" Foreground="Red" >فى سلام</Run>
    </TextBlock>

    <TextBlock FontSize="20" FlowDirection="LeftToRight">
        <Run FlowDirection="RightToLeft">العالم</Run>
        <Run FlowDirection="RightToLeft" Foreground="Red" >فى سلام</Run>
    </TextBlock>

    <TextBlock FontSize="20" Foreground="Blue">العالم فى سلام</TextBlock>

    <Separator/>

    <TextBlock FontSize="20" FlowDirection="RightToLeft">
        <Span Foreground="Red" FlowDirection="LeftToRight">Hello</Span>
        <Span FlowDirection="LeftToRight">World</Span>
    </TextBlock>

    <TextBlock FontSize="20" FlowDirection="LeftToRight">
        <Span Foreground="Red" FlowDirection="RightToLeft">Hello</Span>
        <Span FlowDirection="RightToLeft">World</Span>
    </TextBlock>

    <TextBlock FontSize="20" Foreground="Blue">Hello World</TextBlock>

</StackPanel>

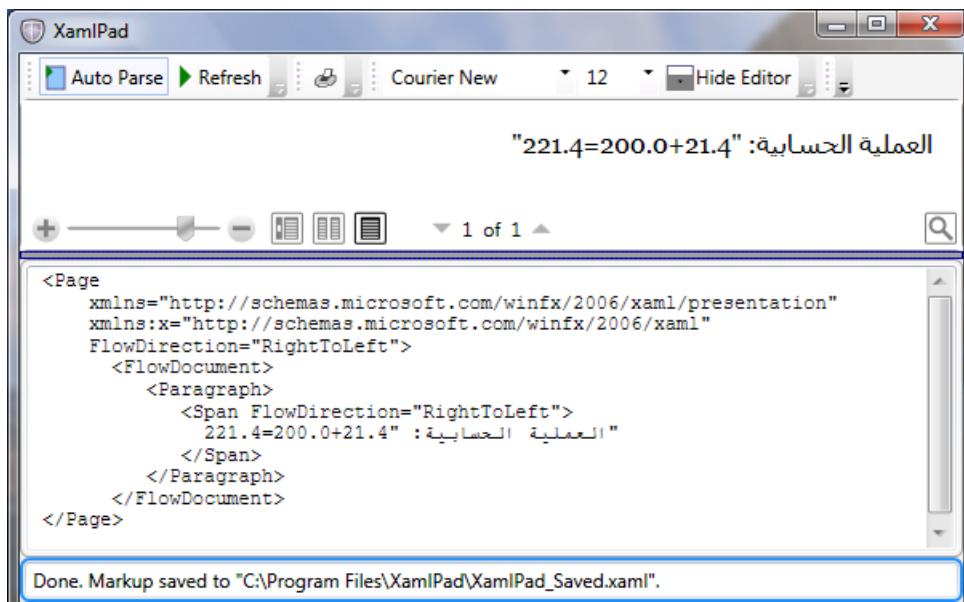
</Page>

```

In the **TextBlock** elements in the sample, the **Span** elements are laid out according to the **FlowDirection** of their parents, but the text within each **Span** element flows according to its own **FlowDirection**. This is applicable to Latin and Arabic – or any other language.

### Adding **xml:lang**

The following graphic shows another example that uses numbers and arithmetic expressions, such as "200.0+21.4=221.4". Notice that only the **FlowDirection** is set.

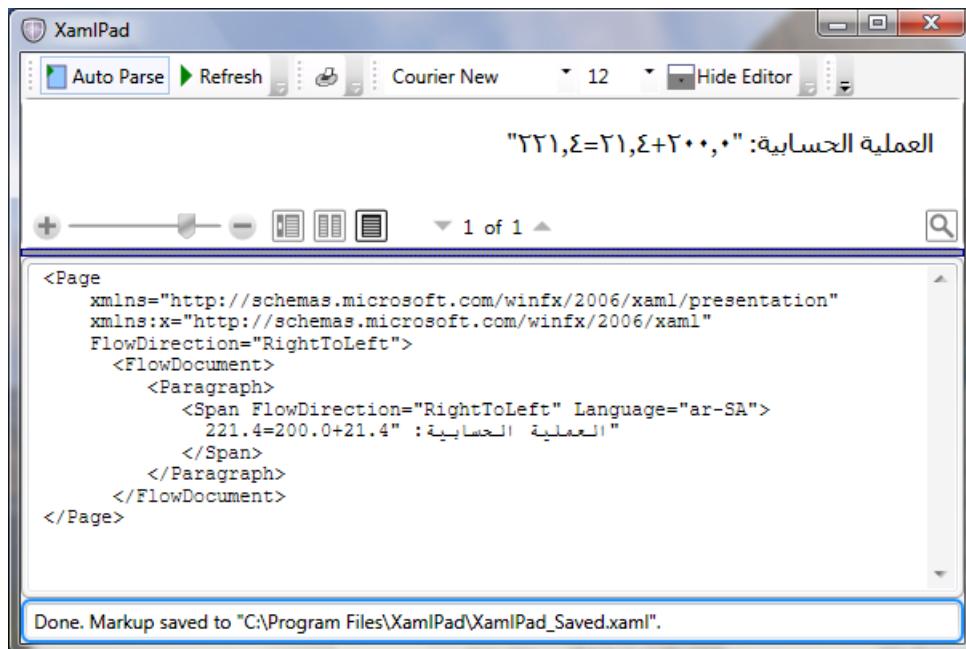


Users of this application will be disappointed by the output, even though the **FlowDirection** is correct the numbers

are not shaped as Arabic numbers should be shaped.

XAML elements can include an XML attribute (`xml:lang`) that defines the language of each element. XAML also supports a XML language principle whereby `xml:lang` values applied to parent elements in the tree are used by child elements. In the previous example, because a language was not defined for the `Run` element or any of its top level elements, the default `xml:lang` was used, which is `en-US` for XAML. The internal number shaping algorithm of Windows Presentation Foundation (WPF) selects numbers in the corresponding language – in this case English. To make the Arabic numbers render correctly `xml:lang` needs to be set.

The following graphic shows the example with `xml:lang` added.



The following example adds `xml:lang` to the application.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">
    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" Language="ar-SA">
                221.4=21.4+200.0" العربية الحسابية :"
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>
```

Be aware that many languages have different `xml:lang` values depending on the targeted region, for example, `"ar-SA"` and `"ar-EG"` represent two variations of Arabic. The previous examples illustrate that you need to define both the `xml:lang` and `FlowDirection` values.

## FlowDirection with Non-text Elements

`FlowDirection` defines not only how text flows in a textual element but also the flow direction of almost every other UI element. The following graphic shows a `ToolBar` that uses a horizontal `LinearGradientBrush` to draw its background with a left to right gradient.

The screenshot shows the XamlPad application interface. At the top, there's a toolbar with buttons for Auto Parse, Refresh, Hide Editor, and font size selection. Below the toolbar is a toolbar containing four buttons labeled Button1, Button2, Button3, and Button4. The main area displays XAML code for a page with a tool bar. The XAML includes a LinearGradientBrush for the background of the tool bar, which transitions from dark red to white. The buttons are arranged from right to left. A status bar at the bottom indicates "Done. Markup saved to 'C:\Program Files\XamlPad\XamlPad\_Saved.xaml'".

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >

<ToolBar>
    <ToolBar.Background>
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,1">
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="DarkRed" Offset="0" />
                <GradientStop Color="DarkBlue" Offset="0.3" />
                <GradientStop Color="LightBlue" Offset="0.6" />
                <GradientStop Color="White" Offset="1" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </ToolBar.Background>

    <Button FontSize="12" Foreground="White">Button1</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button2</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button3</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button4</Button>
    <Rectangle Width="20"/>
</ToolBar>

</Page>
```

Done. Markup saved to "C:\Program Files\XamlPad\XamlPad\_Saved.xaml".

After setting the [FlowDirection](#) to [RightToLeft](#), not only the [ToolBar](#) buttons are arranged from right to left, but even the [LinearGradientBrush](#) realigns its offsets to flow from right to left.

The following graphic shows the realignment of the [LinearGradientBrush](#).

The screenshot shows the XamlPad application interface. At the top, there's a toolbar with buttons for Auto Parse, Refresh, Hide Editor, and font size selection. Below the toolbar is a toolbar containing four buttons labeled Button4, Button3, Button2, and Button1, arranged from right to left. The main area displays XAML code for a page with a tool bar. The XAML includes a LinearGradientBrush for the background of the tool bar, which transitions from dark red to white. The buttons are arranged from right to left. The status bar at the bottom is not visible in this specific screenshot.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >

<ToolBar FlowDirection="RightToLeft">
    <ToolBar.Background>
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,1">
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="DarkRed" Offset="0" />
                <GradientStop Color="DarkBlue" Offset="0.3" />
                <GradientStop Color="LightBlue" Offset="0.6" />
                <GradientStop Color="White" Offset="1" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </ToolBar.Background>

    <Button FontSize="12" Foreground="White">Button4</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button3</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button2</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button1</Button>
    <Rectangle Width="20"/>
</ToolBar>

</Page>
```

The following example draws a [RightToLeftToolBar](#). (To draw it left to right, remove the [FlowDirection](#) attribute on the [ToolBar](#).)

```

<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <ToolBar FlowDirection="RightToLeft" Height="50" DockPanel.Dock="Top">
        <ToolBar.Background>
            <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="DarkRed" Offset="0" />
                    <GradientStop Color="DarkBlue" Offset="0.3" />
                    <GradientStop Color="LightBlue" Offset="0.6" />
                    <GradientStop Color="White" Offset="1" />
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </ToolBar.Background>

        <Button FontSize="12" Foreground="White">Button1</Button>
        <Rectangle Width="20"/>
        <Button FontSize="12" Foreground="White">Button2</Button>
        <Rectangle Width="20"/>
        <Button FontSize="12" Foreground="White">Button3</Button>
        <Rectangle Width="20"/>
        <Button FontSize="12" Foreground="White">Button4</Button>
        <Rectangle Width="20"/>
    </ToolBar>
</Page>

```

## FlowDirection Exceptions

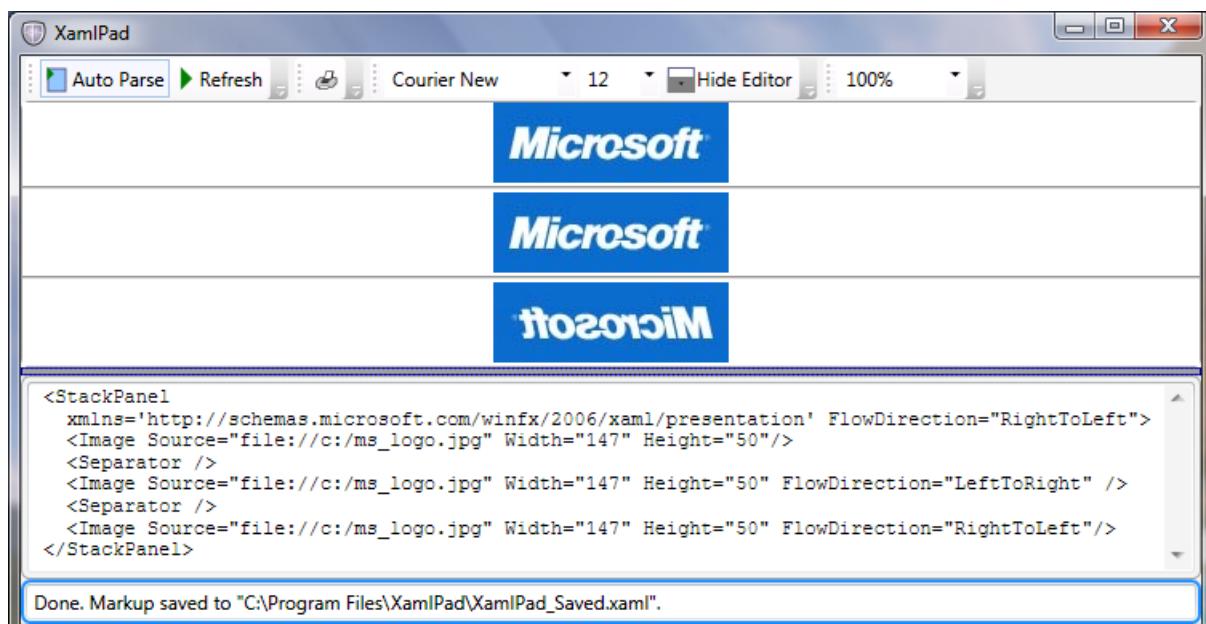
There are a few cases where [FlowDirection](#) does not behave as expected. This section covers two of these exceptions.

### Image

An [Image](#) represents a control that displays an image. In XAML it can be used with a [Source](#) property that defines the uniform resource identifier (URI) of the [Image](#) to display.

Unlike other UI elements, an [Image](#) does not inherit the [FlowDirection](#) from the container. However, if the [FlowDirection](#) is set explicitly to [RightToLeft](#), an [Image](#) is displayed flipped horizontally. This is implemented as a convenient feature for developers of bidirectional content; because in some cases, horizontally flipping the image produces the desired effect.

The following graphic shows a flipped [Image](#).



The following example demonstrates that the [Image](#) fails to inherit the [FlowDirection](#) from the [StackPanel](#) that contains it.

#### NOTE

You must have a file named **ms\_logo.jpg** on your C:\ drive to run this example.

```
<StackPanel  
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'  
    FlowDirection="RightToLeft">  
  
    <Image Source="file:///c:/ms_logo.jpg"  
        Width="147" Height="50"/>  
    <Separator Height="10"/>  
    <Image Source="file:///c:/ms_logo.jpg"  
        Width="147" Height="50" FlowDirection="LeftToRight" />  
    <Separator Height="10"/>  
    <Image Source="file:///c:/ms_logo.jpg"  
        Width="147" Height="50" FlowDirection="RightToLeft"/>  
</StackPanel>
```

#### NOTE

Included in the download files is an **ms\_logo.jpg** file. The code assumes that the jpg file is not inside your project but somewhere on the C:\ drive. You must copy the jpg from the project files to your C:\ drive or change the code to look for the file inside the project. To do this change `Source="file:///c:/ms_logo.jpg"` to `Source="ms_logo.jpg"`.

## Paths

In addition to an [Image](#), another interesting element is [Path](#). A Path is an object that can draw a series of connected lines and curves. It behaves in a manner similar to an [Image](#) regarding its [FlowDirection](#); for example its [RightToLeftFlowDirection](#) is a horizontal mirror of its [LeftToRight](#) one. However, unlike an [Image](#), [Path](#) inherits its [FlowDirection](#) from the container and one does not need to specify it explicitly.

The following example draws a simple arrow using 3 lines. The first arrow inherits the [RightToLeft](#) flow direction from the [StackPanel](#) so that its start and end points are measured from a root on the right side. The second arrow which has an explicit [RightToLeftFlowDirection](#) also starts on the right side. However, the third arrow has its starting root on the left side. For more information on drawing see [LineGeometry](#) and [GeometryGroup](#).

```

<StackPanel
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
    FlowDirection="RightToLeft">

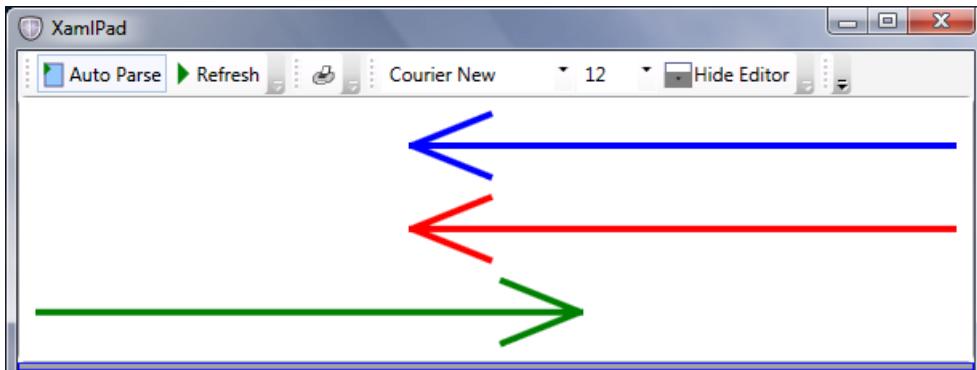
    <Path Stroke="Blue" StrokeThickness="4">
        <Path.Data>
            <GeometryGroup >
                <LineGeometry StartPoint="300,10" EndPoint="350,30" />
                <LineGeometry StartPoint="10,30" EndPoint="352,30" />
                <LineGeometry StartPoint="300,50" EndPoint="350,30" />
            </GeometryGroup>
        </Path.Data>
    </Path>

    <Path Stroke="Red" StrokeThickness="4" FlowDirection="RightToLeft">
        <Path.Data>
            <GeometryGroup >
                <LineGeometry StartPoint="300,10" EndPoint="350,30" />
                <LineGeometry StartPoint="10,30" EndPoint="352,30" />
                <LineGeometry StartPoint="300,50" EndPoint="350,30" />
            </GeometryGroup>
        </Path.Data>
    </Path>

    <Path Stroke="Green" StrokeThickness="4" FlowDirection="LeftToRight">
        <Path.Data>
            <GeometryGroup >
                <LineGeometry StartPoint="300,10" EndPoint="350,30" />
                <LineGeometry StartPoint="10,30" EndPoint="352,30" />
                <LineGeometry StartPoint="300,50" EndPoint="350,30" />
            </GeometryGroup>
        </Path.Data>
    </Path>
</StackPanel>

```

The following graphic shows the output of the previous example with arrows drawn using the `Path` element:



The [Image](#) and [Path](#) are two examples of how Windows Presentation Foundation (WPF) uses [FlowDirection](#). Beside laying out UI elements in a specific direction within a container, [FlowDirection](#) can be used with elements such as [InkPresenter](#) which renders ink on a surface, [LinearGradientBrush](#), [RadialGradientBrush](#). Whenever you need a right to left behavior for your content that mimics a left to right behavior, or vice versa, Windows Presentation Foundation (WPF) provides that capability.

## Number Substitution

Historically, Windows has supported number substitution by allowing the representation of different cultural shapes for the same digits while keeping the internal storage of these digits unified among different locales, for example numbers are stored in their well known hexadecimal values, 0x40, 0x41, but displayed according to the selected language.

This has allowed applications to process numerical values without the need to convert them from one language to another, for example a user can open an Microsoft Excel spreadsheet in a localized Arabic Windows and see the numbers shaped in Arabic, but open it in a European version of Windows and see European representation of the same numbers. This is also necessary for other symbols such as comma separators and percentage symbol because they usually accompany numbers in the same document.

Windows Presentation Foundation (WPF) continues the same tradition, and adds further support for this feature that allows more user control over when and how substitution is used. While this feature is designed for any language, it is particularly useful in bidirectional content where shaping digits for a specific language is usually a challenge for application developers because of the various cultures an application might run on.

The core property controlling how number substitution works in Windows Presentation Foundation (WPF) is the [Substitution](#) dependency property. The [NumberSubstitution](#) class specifies how numbers in text are to be displayed. It has three public properties that define its behavior. The following is a summary of each of the properties:

#### **CultureSource:**

This property specifies how the culture for numbers is determined. It takes one of three [NumberCultureSource](#) enumeration values.

- **Override:** Number culture is that of [CultureOverride](#) property.
- **Text:** Number culture is the culture of the text run. In markup, this would be `xml:lang`, or its alias `Language` property ([Language](#) or [Language](#)). Also, it is the default for classes deriving from [FrameworkContentElement](#). Such classes include [System.Windows.Documents.Paragraph](#), [System.Windows.Documents.Table](#), [System.Windows.Documents.TableCell](#) and so forth.
- **User:** Number culture is the culture of the current thread. This property is the default for all subclasses of [FrameworkElement](#) such as [Page](#), [Window](#) and [TextBlock](#).

#### **CultureOverride:**

The [CultureOverride](#) property is used only if the [CultureSource](#) property is set to [Override](#) and is ignored otherwise. It specifies the number culture. A value of `null`, the default value, is interpreted as en-US.

#### **Substitution:**

This property specifies the type of number substitution to perform. It takes one of the following [NumberSubstitutionMethod](#) enumeration values:

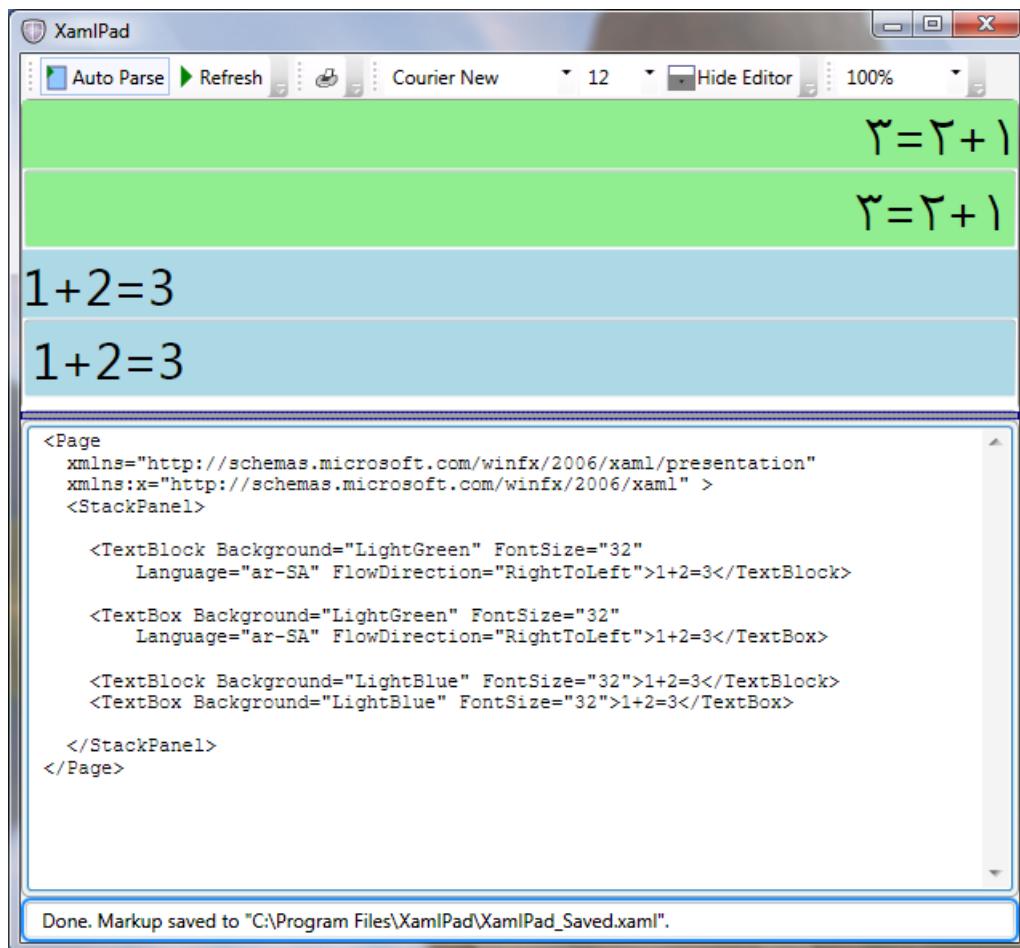
- **AsCulture:** The substitution method is determined based on the number culture's [NumberFormatInfo.DigitSubstitution](#) property. This is the default.
- **Context:** If the number culture is an Arabic or Persian culture, it specifies that the digits depend on the context.
- **European:** Numbers are always rendered as European digits.
- **NativeNational:** Numbers are rendered using the national digits for the number culture, as specified by the culture's [NumberFormat](#).
- **Traditional:** Numbers are rendered using the traditional digits for the number culture. For most cultures, this is the same as [NativeNational](#). However, [NativeNational](#) results in Latin digits for some Arabic cultures, whereas this value results in Arabic digits for all Arabic cultures.

What do those values mean for a bidirectional content developer? In most cases, the developer might need only to define [FlowDirection](#) and the language of each textual UI element, for example `Language="ar-SA"` and the [NumberSubstitution](#) logic takes care of displaying the numbers according to the correct UI. The following example

demonstrates using Arabic and English numbers in a Windows Presentation Foundation (WPF) application running in an Arabic version of Windows.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
<StackPanel>
  <TextBlock Background="LightGreen" FontSize="32"
    Language="ar-SA" FlowDirection="RightToLeft">۱+۲=۳</TextBlock>
  <TextBox Background="LightGreen" FontSize="32"
    Language="ar-SA" FlowDirection="RightToLeft">۱+۲=۳</TextBox>
  <TextBlock Background="LightBlue" FontSize="32">۱+۲=۳</TextBlock>
  <TextBlock Background="LightBlue" FontSize="32">۱+۲=۳</TextBlock>
</StackPanel>
</Page>
```

The following graphic shows the output of the previous sample if you're running in an Arabic version of Windows with Arabic and English numbers displayed:



The **FlowDirection** was important in this case because setting the **FlowDirection** to **LeftToRight** instead would have yielded European digits. The following sections discuss how to have a unified display of digits throughout your document. If this example is not running on Arabic Windows, all the digits display as European digits.

## Defining Substitution Rules

In a real application you might need to set the **Language** programmatically. For example, you want to set the **xml:lang** attribute to be the same as the one used by the system's UI, or maybe change the language depending on the application state.

If you want to make changes based on the application's state, make use of other features provided by Windows Presentation Foundation (WPF).

First, set the application component's `NumberSubstitution.CultureSource="Text"`. Using this setting makes sure that the settings do not come from the UI for text elements that have "User" as the default, such as [TextBlock](#).

For example:

```
<TextBlock  
    Name="text1" NumberSubstitution.CultureSource="Text">  
    1234+5679=6913  
</TextBlock>
```

In the corresponding C# code, set the `Language` property, for example, to `"ar-SA"`.

```
text1.Language = System.Windows.Markup.XmlLanguage.GetLanguage("ar-SA");
```

If you need to set the `Language` property to the current user's UI language use the following code.

```
text1.Language =  
System.Windows.Markup.XmlLanguage.GetLanguage(System.Globalization.CultureInfo.CurrentCulture.IetfLanguageTa  
g);
```

`CultureInfo.CurrentCulture` represents the current culture used by the current thread at run time.

Your final XAML example should be similar to the following example.

```
<Page x:Class="WindowsApplication.Window1"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Title="Code Sample" Height="300" Width="300"  
>  
    <StackPanel>  
        <TextBlock Language="ar-SA"  
            FlowDirection="RightToLeft">3=2+1 :عربى  
        </TextBlock>  
        <TextBlock Language="ar-SA"  
            FlowDirection="RightToLeft"  
            NumberSubstitution.Substitution="European">3=2+1 :عربى :  
        </TextBlock>  
    </StackPanel>  
</Page>
```

Your final C# example should be similar to the following.

```

namespace BidiTest
{
    public partial class Window1 : Window
    {

        public Window1()
        {
            InitializeComponent();

            string currentLanguage =
                System.Globalization.CultureInfo.CurrentCulture.IetfLanguageTag;

            text1.Language = System.Windows.Markup.XmlLanguage.GetLanguage(currentLanguage);

            if (currentLanguage.ToLower().StartsWith("ar"))
            {
                text1.FlowDirection = FlowDirection.RightToLeft;
            }
            else
            {
                text1.FlowDirection = FlowDirection.LeftToRight;
            }
        }
    }
}

```

The following graphic shows what the window looks like for either programming language, displaying Arabic numbers:



## Using the Substitution Property

The way number substitution works in Windows Presentation Foundation (WPF) depends on both the Language of the text element and its [FlowDirection](#). If the [FlowDirection](#) is left to right, then European digits are rendered. However if it is preceded by Arabic text, or has the language set to "ar" and the [FlowDirection](#) is [RightToLeft](#), Arabic digits are rendered instead.

In some cases, however, you might want to create a unified application, for example European digits for all users. Or Arabic digits in [Table](#) cells with a specific [Style](#). One easy way to do that is using the [Substitution](#) property.

In the following example, the first [TextBlock](#) does not have the [Substitution](#) property set, so the algorithm displays Arabic digits as expected. However in the second [TextBlock](#), the substitution is set to European overriding the default substitution for Arabic numbers, and European digits are displayed.

```

<Page x:Class="WindowsApplication.Window1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Code Sample" Height="300" Width="300"
>
    <StackPanel>
        <TextBlock Language="ar-SA"
                  FlowDirection="RightToLeft">3=2+1 :عربي
        </TextBlock>
        <TextBlock Language="ar-SA"
                  FlowDirection="RightToLeft"
                  NumberSubstitution.Substitution="European">3=2+1 :عربى
        </TextBlock>
    </StackPanel>
</Page>

```



# Globalization and Localization How-to Topics

3/5/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section describe how to develop world-ready applications.

## In This Section

- [Localize an Application](#)
- [Use Automatic Layout to Create a Button](#)
- [Use a Grid for Automatic Layout](#)
- [Use a ResourceDictionary to Manage Localizable String Resources](#)
- [Use Resources in Localizable Applications](#)

## Reference

- [System.Globalization](#)
- [FlowDirection](#)
- [NeutralResourcesLanguageAttribute](#)
- [xml:lang Handling in XAML](#)

## Related Sections

# How to: Localize an Application

11/3/2019 • 7 minutes to read • [Edit Online](#)

This tutorial explains how to create a localized application by using the LocBaml tool.

## NOTE

The LocBaml tool is not a production-ready application. It is presented as a sample that uses some of the localization APIs and illustrates how you might write a localization tool.

## Overview

This discussion gives you a step-by-step approach to localizing an application. First, you will prepare your application so that the text that will be translated can be extracted. After the text is translated, you will merge the translated text into a new copy of the original application.

## Requirements

Over the course of this discussion, you will use Microsoft build engine (MSBuild), which is a compiler that runs from the command line.

Also, you will be instructed to use a project file. For instructions on how to use MSBuild and project files, see [Build and Deploy](#).

All the examples in this discussion use en-US (English-US) as the culture. This enables you to work through the steps of the examples without installing a different language.

## Create a Sample Application

In this step, you will prepare your application for localization. In the Windows Presentation Foundation (WPF) samples, a HelloApp sample is supplied that will be used for the code examples in this discussion. If you would like to use this sample, download the Extensible Application Markup Language (XAML) files from the [LocBaml Tool Sample](#).

1. Develop your application to the point where you want to start localization.
2. Specify the development language in the project file so that MSBuild generates a main assembly and a satellite assembly (a file with the .resources.dll extension) to contain the neutral language resources. The project file in the HelloApp sample is HelloApp.csproj. In that file, you will find the development language identified as follows:

```
<UICulture>en-US</UICulture>
```

3. Add Uids to your XAML files. Uids are used to keep track of changes to files and to identify items that must be translated. To add Uids to your files, run **updateuid** on your project file:

```
msbuild -t:updateuid helloapp.csproj
```

To verify that you have no missing or duplicate Uids, run **checkuid**:

```
msbuild -t:checkuid helloapp.csproj
```

After running **updateuid**, your files should contain Uids. For example, in the Pane1.xaml file of HelloApp,

you should find the following:

```
<StackPanel x:Uid="StackPanel_1">  
  
<TextBlock x:Uid="TextBlock_1">Hello World</TextBlock>  
  
<TextBlock x:Uid="TextBlock_2">Goodbye World</TextBlock>  
  
</StackPanel>
```

## Create the Neutral Language Resources Satellite Assembly

After the application is configured to generate a neutral language resources satellite assembly, you build the application. This generates the main application assembly, as well as the neutral language resources satellite assembly that is required by LocBaml for localization. To build the application:

1. Compile HelloApp to create a dynamic-link library (DLL):

```
msbuild helloapp.csproj
```

2. The newly created main application assembly, HelloApp.exe, is created in the following folder:

```
C:\HelloApp\Bin\Debug\
```

3. The newly created neutral language resources satellite assembly, HelloApp.resources.dll, is created in the following folder:

```
C:\HelloApp\Bin\Debug\en-US\
```

## Build the LocBaml Tool

1. All the files necessary to build LocBaml are located in the WPF samples. Download the C# files from the [LocBaml Tool Sample](#).

2. From the command line, run the project file (locbaml.csproj) to build the tool:

```
msbuild locbaml.csproj
```

3. Go to the Bin\Release directory to find the newly created executable file (locbaml.exe).

Example:C:\LocBaml\Bin\Release\locbaml.exe.

4. The options that you can specify when you run LocBaml are as follows:

- **parse** or **-p**: Parses Baml, resources, or DLL files to generate a .csv or .txt file.
- **generate** or **-g**: Generates a localized binary file by using a translated file.
- **out** or **-o {filedirectory}** : Output file name.
- **culture** or **-cul {culture}** : Locale of output assemblies.
- **translation** or **-trans {translation.csv}** : Translated or localized file.
- **asmpath** or **-asmpath: {filedirectory}** : If your XAML code contains custom controls, you must supply the **asmpath** to the custom control assembly.
- **nologo**: Displays no logo or copyright information.
- **verbose**: Displays verbose mode information.

**NOTE**

If you need a list of the options when you are running the tool, type **LocBaml.exe** and press ENTER.

## Use LocBaml to Parse a File

Now that you have created the LocBaml tool, you are ready to use it to parse HelloApp.resources.dll to extract the text content that will be localized.

1. Copy LocBaml.exe to your application's bin\debug folder, where the main application assembly was created.
2. To parse the satellite assembly file and store the output as a .csv file, use the following command:

**LocBaml.exe /parse HelloApp.resources.dll /out:Hello.csv**

**NOTE**

If the input file, HelloApp.resources.dll, is not in the same directory as LocBaml.exe move one of the files so that both files are in the same directory.

3. When you run LocBaml to parse files, the output consists of seven fields delimited by commas (.csv files) or tabs (.txt files). The following shows the parsed .csv file for the HelloApp.resources.dll:

HelloApp.g.en-US.resources:window1.baml,Stack1:System.Windows.Controls.StackPanel.\$Content,Ignore,TRUE, FALSE,,#Text1;#Text2;
---

HelloApp.g.en-US.resources:window1.baml,Text1:System.Windows.Controls.TextBlock.\$Content,None,TRUE, TRUE,,Hello World
---

HelloApp.g.en-US.resources:window1.baml,Text2:System.Windows.Controls.TextBlock.\$Content,None,TRUE, TRUE,,Goodbye World
---

The seven fields are:

- a. **BAML Name**. The name of the BAML resource with respect to the source language satellite assembly.
- b. **Resource Key**. The localized resource identifier.
- c. **Category**. The value type. See [Localization Attributes and Comments](#).
- d. **Readability**. Whether the value can be read by a localizer. See [Localization Attributes and Comments](#).
- e. **Modifiability**. Whether the value can be modified by a localizer. See [Localization Attributes and Comments](#).
- f. **Comments**. Additional description of the value to help determine how a value is localized. See [Localization Attributes and Comments](#).
- g. **Value**. The text value to translate to the desired culture.

The following table shows how these fields map to the delimited values of the .csv file:

BAML NAME	RESOURCE KEY	CATEGORY	READABILITY	MODIFIABILITY	COMMENTS	VALUE
HelloApp.g.en-US.resources:window1.xaml	Stack1:System.Windows.Controls.StackPanel.Content	Ignore	FALSE	FALSE		#Text1;#Text2
HelloApp.g.en-US.resources:window1.xaml	Text1:System.Windows.Controls.TextBlock.Content	None	TRUE	TRUE		Hello World
HelloApp.g.en-US.resources:window1.xaml	Text2:System.Windows.Controls.TextBlock.Content	None	TRUE	TRUE		Goodbye World

Notice that all the values for the **Comments** field contain no values; if a field doesn't have a value, it is empty. Also notice that the item in the first row is neither readable nor modifiable, and has "Ignore" as its **Category** value, all of which indicates that the value is not localizable.

- To facilitate discovery of localizable items in parsed files, particularly in large files, you can sort or filter the items by **Category**, **Readability**, and **Modifiability**. For example, you can filter out unreadable and unmodifiable values.

## Translate the Localizable Content

Use any tool that you have available to translate the extracted content. A good way to do this is to write the resources to a .csv file and view them in Microsoft Excel, making translation changes to the last column (value).

## Use LocBaml to Generate a New .resources.dll File

The content that was identified by parsing HelloApp.resources.dll with LocBaml has been translated and must be merged back into the original application. Use the **generate** or **-g** option to generate a new .resources.dll file.

- Use the following syntax to generate a new HelloApp.resources.dll file. Mark the culture as en-US (/cul:en-US).

```
LocBaml.exe /generate HelloApp.resources.dll /trans:Hello.csv /out:c:\ /cul:en-US
```

### NOTE

If the input file, Hello.csv, is not in the same directory as the executable, LocBaml.exe, move one of the files so that both files are in the same directory.

- Replace the old HelloApp.resources.dll file in the C:\HelloApp\Bin\Debug\en-US\HelloApp.resources.dll directory with your newly created HelloApp.resources.dll file.
- "Hello World" and "Goodbye World" should now be translated in your application.
- To translate to a different culture, use the culture of the language that you are translating to. The following example shows how to translate to French-Canadian:

```
LocBaml.exe /generate HelloApp.resources.dll /trans:Hellofr-CA.csv /out:c:\ /cul:fr-CA
```

5. In the same assembly as the main application assembly, create a new culture-specific folder to house the new satellite assembly. For French-Canadian, the folder would be fr-CA.
6. Copy the generated satellite assembly to the new folder.
7. To test the new satellite assembly, you need to change the culture under which your application will run. You can do this in one of two ways:
  - Change your operating system's regional settings (**Start | Control Panel | Regional and Language Options**).
  - In your application, add the following code to App.xaml.cs:

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.App"
    x:Uid="Application_1"
    StartupUri="Window1.xaml">
</Application>
```

```
using System.Windows;
using System.Globalization;
using System.Threading;

namespace SDKSample
{
    public partial class App : Application
    {
        public App()
        {
            // Change culture under which this application runs
            CultureInfo ci = new CultureInfo("fr-CA");
            Thread.CurrentThread.CurrentCulture = ci;
            Thread.CurrentThread.CurrentUICulture = ci;
        }
    }
}
```

```
Imports System.Windows
Imports System.Globalization
Imports System.Threading

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Public Sub New()
            ' Change culture under which this application runs
            Dim ci As New CultureInfo("fr-CA")
            Thread.CurrentThread.CurrentCulture = ci
            Thread.CurrentThread.CurrentUICulture = ci
        End Sub
    End Class
End Namespace
```

## Some Tips for Using LocBaml

- All dependent assemblies that define custom controls must be copied into the local directory of LocBaml or installed into the GAC. This is necessary because the localization API must have access to the dependent

assemblies when it reads the binary XAML (BAML).

- If the main assembly is signed, the generated resource DLL must also be signed in order for it to be loaded.
- The version of the localized resource DLL needs to be synchronized with the main assembly.

## What's Next

You should now have a basic understanding of how to use the LocBaml tool. You should be able to make a file that contains Uids. By using the LocBaml tool, you should be able to parse a file to extract the localizable content, and after the content is translated, you should be able to generate a .resources.dll file that merges the translated content. This topic does not include every possible detail, but you now have the knowledge necessary to use LocBaml for localizing your applications.

## See also

- [Globalization for WPF](#)
- [Use Automatic Layout Overview](#)

# How to: Use Automatic Layout to Create a Button

3/25/2019 • 2 minutes to read • [Edit Online](#)

This example describes how to use the automatic layout approach to create a button in a localizable application.

Localization of a user interface (UI) can be a time consuming process. Often localizers need to resize and reposition elements in addition to translating text. In the past each language that a UI was adapted for required adjustment. Now with the capabilities of Windows Presentation Foundation (WPF) you can design elements that reduce the need for adjustment. The approach to writing applications that can be more easily resized and repositioned is called `automatic layout`.

## Example

The following two Extensible Application Markup Language (XAML) examples create applications that instantiate a button; one with English text and one with Spanish text. Notice that the code is the same except for the text; the button adjusts to fit the text.

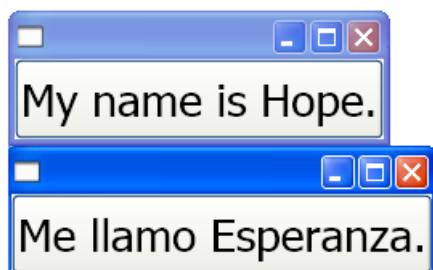
```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ButtonLoc.Panel1"
    Name="myWindow"
    SizeToContent="WidthAndHeight"
    >

    <DockPanel>
        <Button FontSize="28" Height="50">My name is Hope.</Button>
    </DockPanel>
</Window>
```

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ButtonLoc.Panel1"
    Name="myWindow"
    SizeToContent="WidthAndHeight"
    >

    <DockPanel>
        <Button FontSize="28" Height="50">Me llamo Esperanza.</Button>
    </DockPanel>
</Window>
```

The following graphic shows the output of the code samples with auto-resizable buttons:



## See also

- [Use Automatic Layout Overview](#)
- [Use a Grid for Automatic Layout](#)

# How to: Use a Grid for Automatic Layout

4/8/2019 • 2 minutes to read • [Edit Online](#)

This example describes how to use a grid in the automatic layout approach to creating a localizable application.

Localization of a user interface (UI) can be a time consuming process. Often localizers need to re-size and reposition elements in addition to translating text. In the past each language that a UI was adapted for required adjustment. Now with the capabilities of Windows Presentation Foundation (WPF) you can design elements that reduce the need for adjustment. The approach to writing applications that can be more easily re-sized and repositioned is called `auto layout`.

The following Extensible Application Markup Language (XAML) example demonstrates using a grid to position some buttons and text. Notice that the height and width of the cells are set to `Auto`; therefore the cell that contains the button with an image adjusts to fit the image. Because the `Grid` element can adjust to its content it can be useful when taking the automatic layout approach to designing applications that can be localized.

## Example

The following example shows how to use a grid.

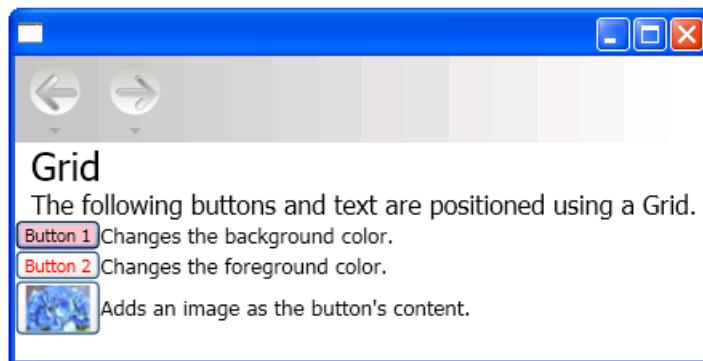
```

<Grid Name="grid" ShowGridLines ="false">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>

<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="0" FontSize="24">Grid
</TextBlock>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="1" FontSize="12"
    Grid.ColumnSpan="2">The following buttons and text are positioned using a Grid.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="2" Background="Pink"
    BorderBrush="Black" BorderThickness="10">Button 1
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="2" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the background
    color.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="3" Foreground="Red">
    Button 2
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="3" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the foreground
    color.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="4">
    <Image Source="data\flower.jpg"/>
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="4" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Adds an image as
    the button's content.
</TextBlock>
</Grid>

```

The following graphic shows the output of the code sample.



Grid

## See also

- [Use Automatic Layout Overview](#)
- [Use Automatic Layout to Create a Button](#)

# How to: Use a ResourceDictionary to Manage Localizable String Resources

4/9/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use a [ResourceDictionary](#) to package localizable string resources for Windows Presentation Foundation (WPF) applications.

## To use a ResourceDictionary to manage localizable string resources

1. Create a [ResourceDictionary](#) that contains the strings you would like to localize. The following code shows an example.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib">

    <!-- String resource that can be localized -->
    <system:String x:Key="localizedMessage">en-US Message</system:String>

</ResourceDictionary>
```

This code defines a string resource, `localizedMessage`, of type [String](#), from the [System](#) namespace in `mscorlib.dll`.

2. Add the [ResourceDictionary](#) to your application, using the following code.

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="StringResources.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

3. Use the string resource from markup, using Extensible Application Markup Language (XAML) like the following.

```
<!-- Declarative use of string resource from StringResources.xaml resource dictionary -->
<TextBox DockPanel.Dock="Top" Text="{StaticResource localizedMessage}" />
```

4. Use the string resource from code-behind, using code like the following.

```
// Programmatic use of string resource from StringResources.xaml resource dictionary
string localizedMessage = (string)Application.Current.FindResource("localizedMessage");
MessageBox.Show(localizedMessage);
```

```
' Programmatic use of string resource from StringResources.xaml resource dictionary
Dim localizedMessage As String = CStr(Application.Current.FindResource("localizedMessage"))
MessageBox.Show(localizedMessage)
```

5. Localize the application. For more information, see [Localize an Application](#).

# How to: Use Resources in Localizable Applications

7/16/2019 • 2 minutes to read • [Edit Online](#)

Localization means to adapt a UI to different cultures. To do this, text such as titles, captions, list box items and so forth have to be translated. To make translation easier the items to be translated are collected into resource files. See [Localize an Application](#) for information on how to create a resource file for localization. To make a WPF application localizable, developers need to build all the localizable resources into a resource assembly. The resource assembly is localized into different languages, and the code-behind uses resource management API to load. One of the files required for a WPF application is a project file (.proj). All resources that you use in your application should be included in the project file. The following code example shows this.

## Example

XAML

```
<Resource Include="data\picture1.jpg"/>  
  
<EmbeddedResource Include="data\stringtable.en-US.restext"/>
```

To use a resource in your application, you instantiate [ResourceManager](#) and load the resource you want to use. The following demonstrates how to do this.

```
void OnClick(object sender, RoutedEventArgs e)  
{  
    ResourceManager rm = new ResourceManager ("MySampleApp.data.stringtable",  
        Assembly.GetExecutingAssembly());  
    Text1.Text = rm.GetString("Message");  
}
```

# Layout

8/27/2019 • 9 minutes to read • [Edit Online](#)

This topic describes the Windows Presentation Foundation (WPF) layout system. Understanding how and when layout calculations occur is essential for creating user interfaces in WPF.

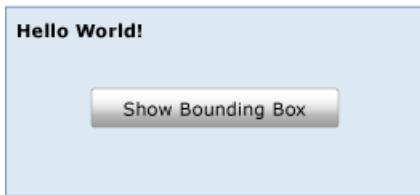
This topic contains the following sections:

- [Element Bounding Boxes](#)
- [The Layout System](#)
- [Measuring and Arranging Children](#)
- [Panel Elements and Custom Layout Behaviors](#)
- [Layout Performance Considerations](#)
- [Sub-pixel Rendering and Layout Rounding](#)
- [What's Next](#)

## Element Bounding Boxes

When thinking about layout in WPF, it is important to understand the bounding box that surrounds all elements. Each [FrameworkElement](#) consumed by the layout system can be thought of as a rectangle that is slotted into the layout. The [LayoutInformation](#) class returns the boundaries of an element's layout allocation, or slot. The size of the rectangle is determined by calculating the available screen space, the size of any constraints, layout-specific properties (such as margin and padding), and the individual behavior of the parent [Panel](#) element. Processing this data, the layout system is able to calculate the position of all the children of a particular [Panel](#). It is important to remember that sizing characteristics defined on the parent element, such as a [Border](#), affect its children.

The following illustration shows a simple layout.



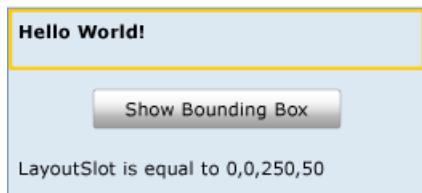
This layout can be achieved by using the following XAML.

```

<Grid Name="myGrid" Background="LightSteelBlue" Height="150">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <TextBlock Name="txt1" Margin="5" FontSize="16" FontFamily="Verdana" Grid.Column="0" Grid.Row="0">Hello
    World!</TextBlock>
    <Button Click="getLayoutSlot1" Width="125" Height="25" Grid.Column="0" Grid.Row="1">Show Bounding
    Box</Button>
    <TextBlock Name="txt2" Grid.Column="1" Grid.Row="2"/>
</Grid>

```

A single [TextBlock](#) element is hosted within a [Grid](#). While the text fills only the upper-left corner of the first column, the allocated space for the [TextBlock](#) is actually much larger. The bounding box of any [FrameworkElement](#) can be retrieved by using the [GetLayoutSlot](#) method. The following illustration shows the bounding box for the [TextBlock](#) element.



As shown by the yellow rectangle, the allocated space for the [TextBlock](#) element is actually much larger than it appears. As additional elements are added to the [Grid](#), this allocation could shrink or expand, depending on the type and size of elements that are added.

The layout slot of the [TextBlock](#) is translated into a [Path](#) by using the [GetLayoutSlot](#) method. This technique can be useful for displaying the bounding box of an element.

```

private void getLayoutSlot1(object sender, System.Windows.RoutedEventArgs e)
{
    RectangleGeometry myRectangleGeometry = new RectangleGeometry();
    myRectangleGeometry.Rect = LayoutInformation.GetLayoutSlot(txt1);
    Path myPath = new Path();
    myPath.Data = myRectangleGeometry;
    myPath.Stroke = Brushes.LightGoldenrodYellow;
    myPath.StrokeThickness = 5;
    Grid.SetColumn(myPath, 0);
    Grid.SetRow(myPath, 0);
    myGrid.Children.Add(myPath);
    txt2.Text = "LayoutSlot is equal to " + LayoutInformation.GetLayoutSlot(txt1).ToString();
}

```

```

Private Sub getLayoutSlot1(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim myRectangleGeometry As New RectangleGeometry
    myRectangleGeometry.Rect = LayoutInformation.GetLayoutSlot(txt1)
    Dim myPath As New Path
    myPath.Data = myRectangleGeometry
    myPath.Stroke = Brushes.LightGoldenrodYellow
    myPath.StrokeThickness = 5
    Grid.SetColumn(myPath, 0)
    Grid.SetRow(myPath, 0)
    myGrid.Children.Add(myPath)
    txt2.Text = "LayoutSlot is equal to " + LayoutInformation.GetLayoutSlot(txt1).ToString()
End Sub

```

## The Layout System

At its simplest, layout is a recursive system that leads to an element being sized, positioned, and drawn. More specifically, layout describes the process of measuring and arranging the members of a [Panel](#) element's [Children](#) collection. Layout is an intensive process. The larger the [Children](#) collection, the greater the number of calculations that must be made. Complexity can also be introduced based on the layout behavior defined by the [Panel](#) element that owns the collection. A relatively simple [Panel](#), such as [Canvas](#), can have significantly better performance than a more complex [Panel](#), such as [Grid](#).

Each time that a child [UIElement](#) changes its position, it has the potential to trigger a new pass by the layout system. Therefore, it is important to understand the events that can invoke the layout system, as unnecessary invocation can lead to poor application performance. The following describes the process that occurs when the layout system is invoked.

1. A child [UIElement](#) begins the layout process by first having its core properties measured.
2. Sizing properties defined on [FrameworkElement](#) are evaluated, such as [Width](#), [Height](#), and [Margin](#).
3. [Panel](#)-specific logic is applied, such as [Dock](#) direction or stacking [Orientation](#).
4. Content is arranged after all children have been measured.
5. The [Children](#) collection is drawn on the screen.
6. The process is invoked again if additional [Children](#) are added to the collection, a [LayoutTransform](#) is applied, or the [UpdateLayout](#) method is called.

This process and how it is invoked are defined in more detail in the following sections.

## Measuring and Arranging Children

The layout system completes two passes for each member of the [Children](#) collection, a measure pass and an arrange pass. Each child [Panel](#) provides its own [MeasureOverride](#) and [ArrangeOverride](#) methods to achieve its own specific layout behavior.

During the measure pass, each member of the [Children](#) collection is evaluated. The process begins with a call to the [Measure](#) method. This method is called within the implementation of the parent [Panel](#) element, and does not have to be called explicitly for layout to occur.

First, native size properties of the [UIElement](#) are evaluated, such as [Clip](#) and [Visibility](#). This generates a value named `constraintSize` that is passed to [MeasureCore](#).

Secondly, framework properties defined on [FrameworkElement](#) are processed, which affects the value of `constraintSize`. These properties generally describe the sizing characteristics of the underlying [UIElement](#), such as its [Height](#), [Width](#), [Margin](#), and [Style](#). Each of these properties can change the space that is necessary to display

the element. [MeasureOverride](#) is then called with `constraintSize` as a parameter.

#### NOTE

There is a difference between the properties of [Height](#) and [Width](#) and [ActualHeight](#) and [ActualWidth](#). For example, the [ActualHeight](#) property is a calculated value based on other height inputs and the layout system. The value is set by the layout system itself, based on an actual rendering pass, and may therefore lag slightly behind the set value of properties, such as [Height](#), that are the basis of the input change.

Because [ActualHeight](#) is a calculated value, you should be aware that there could be multiple or incremental reported changes to it as a result of various operations by the layout system. The layout system may be calculating required measure space for child elements, constraints by the parent element, and so on.

The ultimate goal of the measure pass is for the child to determine its [DesiredSize](#), which occurs during the [MeasureCore](#) call. The [DesiredSize](#) value is stored by [Measure](#) for use during the content arrange pass.

The arrange pass begins with a call to the [Arrange](#) method. During the arrange pass, the parent [Panel](#) element generates a rectangle that represents the bounds of the child. This value is passed to the [ArrangeCore](#) method for processing.

The [ArrangeCore](#) method evaluates the [DesiredSize](#) of the child and evaluates any additional margins that may affect the rendered size of the element. [ArrangeCore](#) generates an `arrangeSize`, which is passed to the [ArrangeOverride](#) method of the [Panel](#) as a parameter. [ArrangeOverride](#) generates the `finalSize` of the child. Finally, the [ArrangeCore](#) method does a final evaluation of offset properties, such as margin and alignment, and puts the child within its layout slot. The child does not have to (and frequently does not) fill the entire allocated space. Control is then returned to the parent [Panel](#) and the layout process is complete.

## Panel Elements and Custom Layout Behaviors

WPF includes a group of elements that derive from [Panel](#). These [Panel](#) elements enable many complex layouts. For example, stacking elements can easily be achieved by using the [StackPanel](#) element, while more complex and free flowing layouts are possible by using a [Canvas](#).

The following table summarizes the available layout [Panel](#) elements.

PANEL NAME	DESCRIPTION
<a href="#">Canvas</a>	Defines an area within which you can explicitly position child elements by coordinates relative to the <a href="#">Canvas</a> area.
<a href="#">DockPanel</a>	Defines an area within which you can arrange child elements either horizontally or vertically, relative to each other.
<a href="#">Grid</a>	Defines a flexible grid area that consists of columns and rows.
<a href="#">StackPanel</a>	Arranges child elements into a single line that can be oriented horizontally or vertically.
<a href="#">VirtualizingPanel</a>	Provides a framework for <a href="#">Panel</a> elements that virtualize their child data collection. This is an abstract class.
<a href="#">WrapPanel</a>	Positions child elements in sequential position from left to right, breaking content to the next line at the edge of the containing box. Subsequent ordering occurs sequentially from top to bottom or right to left, depending on the value of the <a href="#">Orientation</a> property.

For applications that require a layout that is not possible by using any of the predefined [Panel](#) elements, custom layout behaviors can be achieved by inheriting from [Panel](#) and overriding the [MeasureOverride](#) and [ArrangeOverride](#) methods.

## Layout Performance Considerations

Layout is a recursive process. Each child element in a [Children](#) collection gets processed during each invocation of the layout system. As a result, triggering the layout system should be avoided when it is not necessary. The following considerations can help you achieve better performance.

- Be aware of which property value changes will force a recursive update by the layout system.

Dependency properties whose values can cause the layout system to be initialized are marked with public flags. [AffectsMeasure](#) and [AffectsArrange](#) provide useful clues as to which property value changes will force a recursive update by the layout system. In general, any property that can affect the size of an element's bounding box should have a [AffectsMeasure](#) flag set to true. For more information, see [Dependency Properties Overview](#).

- When possible, use a [RenderTransform](#) instead of a [LayoutTransform](#).

A [LayoutTransform](#) can be a very useful way to affect the content of a user interface (UI). However, if the effect of the transform does not have to impact the position of other elements, it is best to use a [RenderTransform](#) instead, because [RenderTransform](#) does not invoke the layout system. [LayoutTransform](#) applies its transformation and forces a recursive layout update to account for the new position of the affected element.

- Avoid unnecessary calls to [UpdateLayout](#).

The [UpdateLayout](#) method forces a recursive layout update, and is frequently not necessary. Unless you are sure that a full update is required, rely on the layout system to call this method for you.

- When working with a large [Children](#) collection, consider using a [VirtualizingStackPanel](#) instead of a regular [StackPanel](#).

By virtualizing the child collection, the [VirtualizingStackPanel](#) only keeps objects in memory that are currently within the parent's ViewPort. As a result, performance is substantially improved in most scenarios.

## Sub-pixel Rendering and Layout Rounding

The WPF graphics system uses device-independent units to enable resolution and device independence. Each device independent pixel automatically scales with the system's dots per inch (dpi) setting. This provides WPF applications proper scaling for different dpi settings and makes the application automatically dpi-aware.

However, this dpi independence can create irregular edge rendering because of anti-aliasing. These artifacts, typically seen as blurry or semi-transparent edges, can occur when the location of an edge falls in the middle of a device pixel instead of between device pixels. The layout system provides a way to adjust for this with layout rounding. Layout rounding is where the layout system rounds any non-integral pixel values during the layout pass.

Layout rounding is disabled by default. To enable layout rounding, set the [UseLayoutRounding](#) property to `true` on any [FrameworkElement](#). Because it is a dependency property, the value will propagate to all the children in the visual tree. To enable layout rounding for the entire UI, set [UseLayoutRounding](#) to `true` on the root container. For an example, see [UseLayoutRounding](#).

## What's Next

Understanding how elements are measured and arranged is the first step in understanding layout. For more information about the available [Panel](#) elements, see [Panels Overview](#). To better understand the various positioning properties that can affect layout, see [Alignment, Margins, and Padding Overview](#). When you are ready to put it all together in a lightweight application, see [Walkthrough: My first WPF desktop application](#).

## See also

- [FrameworkElement](#)
- [UIElement](#)
- [Panels Overview](#)
- [Alignment, Margins, and Padding Overview](#)
- [Layout and Design](#)

# Migration and Interoperability

10/29/2019 • 2 minutes to read • [Edit Online](#)

This page contains links to documents that discuss how to implement interoperability between Windows Presentation Foundation (WPF) applications and other types of Microsoft Windows applications.

## In This Section

[WPF and Windows Forms Interoperation](#)

[WPF and Win32 Interoperation](#)

[WPF and Direct3D9 Interoperation](#)

## Reference

TERM	DEFINITION
<a href="#">WindowsFormsHost</a>	An element that you can use to host a Windows Forms control as an element of a WPF page.
<a href="#">ElementHost</a>	A Windows Forms control that you can use to host a Windows Presentation Foundation (WPF) control.
<a href="#">HwndSource</a>	Hosts a WPF region within a Win32 application.
<a href="#">HwndHost</a>	Base class for <a href="#">WindowsFormsHost</a> , defines some basic functionality that all HWND-based technologies use when hosted by a WPF application. Subclass this to host a Win32 window within a WPF application.
<a href="#">BrowserInteropHelper</a>	A helper class for reporting conditions of the browser environment for a WPF application that is hosted by a browser.

## Related Sections

# WPF and Windows Forms Interoperation

8/22/2019 • 8 minutes to read • [Edit Online](#)

WPF and Windows Forms present two different architectures for creating application interfaces. The [System.Windows.Forms.Integration](#) namespace provides classes that enable common interoperation scenarios. The two key classes that implement interoperation capabilities are [WindowsFormsHost](#) and [ElementHost](#). This topic describes which interoperation scenarios are supported and which scenarios are not supported.

## NOTE

Special consideration is given to the *hybrid control* scenario. A hybrid control has a control from one technology nested in a control from the other technology. This is also called a *nested interoperation*. A *multilevel hybrid control* has more than one level of hybrid control nesting. An example of a multilevel nested interoperation is a Windows Forms control that contains a WPF control, which contains another Windows Forms control. Multilevel hybrid controls are not supported.

## Hosting Windows Forms Controls in WPF

The following interoperation scenarios are supported when a WPF control hosts a Windows Forms control:

- The WPF control may host one or more Windows Forms controls using XAML.
- It may host one or more Windows Forms controls using code.
- It may host Windows Forms container controls that contain other Windows Forms controls.
- It may host a master/detail form with a WPF master and Windows Forms details.
- It may host a master/detail form with a Windows Forms master and WPF details.
- It may host one or more ActiveX controls.
- It may host one or more composite controls.
- It may host hybrid controls using Extensible Application Markup Language (XAML).
- It may host hybrid controls using code.

## Layout Support

The following list describes the known limitations when the [WindowsFormsHost](#) element attempts to integrate its hosted Windows Forms control into the WPF layout system.

- In some cases, Windows Forms controls cannot be resized, or can be sized only to specific dimensions. For example, a Windows Forms [ComboBox](#) control supports only a single height, which is defined by the control's font size. In a WPF dynamic layout, which assumes that elements can stretch vertically, a hosted [ComboBox](#) control will not stretch as expected.
- Windows Forms controls cannot be rotated or skewed. For example, when you rotate your user interface by 90 degrees, hosted Windows Forms controls will maintain their upright position.
- In most cases, Windows Forms controls do not support proportional scaling. Although the overall dimensions of the control will scale, child controls and component elements of the control may not resize as expected. This limitation depends on how well each Windows Forms control supports scaling.
- In a WPF user interface, you can change the z-order of elements to control overlapping behavior. A hosted

Windows Forms control is drawn in a separate HWND, so it is always drawn on top of WPF elements.

- Windows Forms controls support autoscaling based on the font size. In a WPF user interface, changing the font size does not resize the entire layout, although individual elements may dynamically resize.

## Ambient Properties

Some of the ambient properties of WPF controls have Windows Forms equivalents. These ambient properties are propagated to the hosted Windows Forms controls and exposed as public properties on the [WindowsFormsHost](#) control. The [WindowsFormsHost](#) control translates each WPF ambient property into its Windows Forms equivalent.

For more information, see [Windows Forms and WPF Property Mapping](#).

## Behavior

The following table describes interoperation behavior.

BEHAVIOR	SUPPORTED	NOT SUPPORTED
Transparency	Windows Forms control rendering supports transparency. The background of the parent WPF control can become the background of hosted Windows Forms controls.	Some Windows Forms controls do not support transparency. For example, the <a href="#">TextBox</a> and <a href="#">ComboBox</a> controls will not be transparent when hosted by WPF.

BEHAVIOR	SUPPORTED	NOT SUPPORTED
Tabbing	<p>Tab order for hosted Windows Forms controls is the same as when those controls are hosted in a Windows Forms-based application.</p> <p>Tabbing from a WPF control to a Windows Forms control with the TAB key and SHIFT+TAB keys works as usual.</p> <p>Windows Forms controls that have a <a href="#">TabStop</a> property value of <code>false</code> do not receive focus when the user tabs through controls.</p> <ul style="list-style-type: none"> <li>- Each <a href="#">WindowsFormsHost</a> control has a <a href="#">TabIndex</a> value, which determines when that <a href="#">WindowsFormsHost</a> control will receive focus.</li> <li>- Windows Forms controls that are contained inside a <a href="#">WindowsFormsHost</a> container follow the order specified by the <a href="#">TabIndex</a> property. Tabbing from the last tab index puts focus on the next WPF control, if one exists. If no other focusable WPF control exists, tabbing returns to the first Windows Forms control in the tab order.</li> <li>- <a href="#">TabIndex</a> values for controls inside the <a href="#">WindowsFormsHost</a> are relative to sibling Windows Forms controls that are contained in the <a href="#">WindowsFormsHost</a> control.</li> <li>- Tabbing honors control-specific behavior. For example, pressing the TAB key in a <a href="#">TextBox</a> control that has a <a href="#">AcceptsTab</a> property value of <code>true</code> enters a tab in the text box instead of moving the focus.</li> </ul>	Not applicable.

Behavior	Supported	Not Supported
Navigation with arrow keys	<ul style="list-style-type: none"> <li>- Navigation with arrow keys in the <a href="#">WindowsFormsHost</a> control is the same as in an ordinary Windows Forms container control: The UP ARROW and LEFT ARROW keys select the previous control, and the DOWN ARROW and RIGHT ARROW keys select the next control.</li> <li>- The UP ARROW and LEFT ARROW keys from the first control that is contained in the <a href="#">WindowsFormsHost</a> control perform the same action as the SHIFT+TAB keyboard shortcut. If there is a focusable WPF control, focus moves outside the <a href="#">WindowsFormsHost</a> control. This behavior differs from the standard <a href="#">ContainerControl</a> behavior in that no wrapping to the last control occurs. If no other focusable WPF control exists, focus returns to the last Windows Forms control in the tab order.</li> <li>- The DOWN ARROW and RIGHT ARROW keys from the last control that is contained in the <a href="#">WindowsFormsHost</a> control perform the same action as the TAB key. If there is a focusable WPF control, focus moves outside the <a href="#">WindowsFormsHost</a> control. This behavior differs from the standard <a href="#">ContainerControl</a> behavior in that no wrapping to the first control occurs. If no other focusable WPF control exists, focus returns to the first Windows Forms control in the tab order.</li> </ul>	Not applicable.
Accelerators	Accelerators work as usual, except where noted in the "Not supported" column.	Duplicate accelerators across technologies do not work like ordinary duplicate accelerators. When an accelerator is duplicated across technologies, with at least one on a Windows Forms control and the other on a WPF control, the Windows Forms control always receives the accelerator. Focus does not toggle between the controls when the duplicate accelerator is pressed.

Behavior	Supported	Not Supported
Shortcut keys	Shortcut keys work as usual, except where noted in the "Not supported" column.	<ul style="list-style-type: none"> <li>- Windows Forms shortcut keys that are handled at the preprocessing stage always take precedence over WPF shortcut keys. For example, if you have a <a href="#">ToolStrip</a> control with CTRL+S shortcut keys defined, and there is a WPF command bound to CTRL+S, the <a href="#">ToolStrip</a> control handler is always invoked first, regardless of focus.</li> <li>- Windows Forms shortcut keys that are handled by the <a href="#">KeyDown</a> event are processed last in WPF. You can prevent this behavior by overriding the Windows Forms control's <a href="#">IsInputKey</a> method or handling the <a href="#">PreviewKeyDown</a> event. Return <code>true</code> from the <a href="#">IsInputKey</a> method, or set the value of the <a href="#">PreviewKeyDownEventArgs.IsInputKey</a> property to <code>true</code> in your <a href="#">PreviewKeyDown</a> event handler.</li> </ul>
AcceptsReturn, AcceptsTab, and other control-specific behavior	Properties that change the default keyboard behavior work as usual, assuming that the Windows Forms control overrides the <a href="#">IsInputKey</a> method to return <code>true</code> .	Windows Forms controls that change default keyboard behavior by handling the <a href="#">KeyDown</a> event are processed last in the host WPF control. Because these controls are processed last, they can produce unexpected behavior.
Enter and Leave Events	When focus is not going to the containing <a href="#">ElementHost</a> control, the Enter and Leave events are raised as usual when focus changes in a single <a href="#">WindowsFormsHost</a> control.	<p>Enter and Leave events are not raised when the following focus changes occur:</p> <ul style="list-style-type: none"> <li>- From inside to outside a <a href="#">WindowsFormsHost</a> control.</li> <li>- From outside to inside a <a href="#">WindowsFormsHost</a> control.</li> <li>- Outside a <a href="#">WindowsFormsHost</a> control.</li> <li>- From a Windows Forms control hosted in a <a href="#">WindowsFormsHost</a> control to an <a href="#">ElementHost</a> control hosted inside the same <a href="#">WindowsFormsHost</a>.</li> </ul>
Multithreading	All varieties of multithreading are supported.	Both the Windows Forms and WPF technologies assume a single-threaded concurrency model. During debugging, calls to framework objects from other threads will raise an exception to enforce this requirement.
Security	All interoperation scenarios require full trust.	No interoperation scenarios are allowed in partial trust.
Accessibility	All accessibility scenarios are supported. Assistive technology products function correctly when they are used for hybrid applications that contain both Windows Forms and WPF controls.	Not applicable.

BEHAVIOR	SUPPORTED	NOT SUPPORTED
Clipboard	All Clipboard operations work as usual. This includes cutting and pasting between Windows Forms and WPF controls.	Not applicable.
Drag-and-drop feature	All drag-and-drop operations work as usual. This includes operations between Windows Forms and WPF controls.	Not applicable.

## Hosting WPF Controls in Windows Forms

The following interoperation scenarios are supported when a Windows Forms control hosts a WPF control:

- Hosting one or more WPF controls using code.
- Associating a property sheet with one or more hosted WPF controls.
- Hosting one or more WPF pages in a form.
- Starting a WPF window.
- Hosting a master/detail form with a Windows Forms master and WPF details.
- Hosting a master/detail form with a WPF master and Windows Forms details.
- Hosting custom WPF controls.
- Hosting hybrid controls.

### Ambient Properties

Some of the ambient properties of Windows Forms controls have WPF equivalents. These ambient properties are propagated to the hosted WPF controls and exposed as public properties on the [ElementHost](#) control. The [ElementHost](#) control translates each Windows Forms ambient property to its WPF equivalent.

For more information, see [Windows Forms and WPF Property Mapping](#).

### Behavior

The following table describes interoperation behavior.

BEHAVIOR	SUPPORTED	NOT SUPPORTED
Transparency	WPF control rendering supports transparency. The background of the parent Windows Forms control can become the background of hosted WPF controls.	Not applicable.
Multithreading	All varieties of multithreading are supported.	Both the Windows Forms and WPF technologies assume a single-threaded concurrency model. During debugging, calls to framework objects from other threads will raise an exception to enforce this requirement.
Security	All interoperation scenarios require full trust.	No interoperation scenarios are allowed in partial trust.

BEHAVIOR	SUPPORTED	NOT SUPPORTED
Accessibility	All accessibility scenarios are supported. Assistive technology products function correctly when they are used for hybrid applications that contain both Windows Forms and WPF controls.	Not applicable.
Clipboard	All Clipboard operations work as usual. This includes cutting and pasting between Windows Forms and WPF controls.	Not applicable.
Drag-and-drop feature	All drag-and-drop operations work as usual. This includes operations between Windows Forms and WPF controls.	Not applicable.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Walkthrough: Hosting a Windows Forms Control in WPF](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)
- [Windows Forms and WPF Property Mapping](#)

# Windows Forms and WPF Interoperability Input Architecture

4/28/2019 • 4 minutes to read • [Edit Online](#)

Interoperation between the WPF and Windows Forms requires that both technologies have the appropriate keyboard input processing. This topic describes how these technologies implement keyboard and message processing to enable smooth interoperation in hybrid applications.

This topic contains the following subsections:

- Modeless Forms and Dialog Boxes
- WindowsFormsHost Keyboard and Message Processing
- ElementHost Keyboard and Message Processing

## Modeless Forms and Dialog Boxes

Call the [EnableWindowsFormsInterop](#) method on the [WindowsFormsHost](#) element to open a modeless form or dialog box from a WPF-based application.

Call the [EnableModelessKeyboardInterop](#) method on the [ElementHost](#) control to open a modeless WPF page in a Windows Forms-based application.

## WindowsFormsHost Keyboard and Message Processing

When hosted by a WPF-based application, Windows Forms keyboard and message processing consists of the following:

- The [WindowsFormsHost](#) class acquires messages from the WPF message loop, which is implemented by the [ComponentDispatcher](#) class.
- The [WindowsFormsHost](#) class creates a surrogate Windows Forms message loop to ensure that ordinary Windows Forms keyboard processing occurs.
- The [WindowsFormsHost](#) class implements the [IKeyboardInputSink](#) interface to coordinate focus management with WPF.
- The [WindowsFormsHost](#) controls register themselves and start their message loops.

The following sections describe these parts of the process in more detail.

### Acquiring Messages from the WPF Message Loop

The [ComponentDispatcher](#) class implements the message loop manager for WPF. The [ComponentDispatcher](#) class provides hooks to enable external clients to filter messages before WPF processes them.

The interoperation implementation handles the [ComponentDispatcher.ThreadFilterMessage](#) event, which enables Windows Forms controls to process messages before WPF controls.

### Surrogate Windows Forms Message Loop

By default, the [System.Windows.Forms.Application](#) class contains the primary message loop for Windows Forms applications. During interoperation, the Windows Forms message loop does not process messages. Therefore, this logic must be reproduced. The handler for the [ComponentDispatcher.ThreadFilterMessage](#) event performs the

following steps:

1. Filters the message using the [IMessageFilter](#) interface.
2. Calls the [Control.PreProcessMessage](#) method.
3. Translates and dispatches the message, if it is required.
4. Passes the message to the hosting control, if no other controls process the message.

### IKeyboardInputSink Implementation

The surrogate message loop handles keyboard management. Therefore, the [IKeyboardInputSink.TabInto](#) method is the only [IKeyboardInputSink](#) member that requires an implementation in the [WindowsFormsHost](#) class.

By default, the [HwndHost](#) class returns `false` for its [IKeyboardInputSink.TabInto](#) implementation. This prevents tabbing from a WPF control to a Windows Forms control.

The [WindowsFormsHost](#) implementation of the [IKeyboardInputSink.TabInto](#) method performs the following steps:

1. Finds the first or last Windows Forms control that is contained by the [WindowsFormsHost](#) control and that can receive focus. The control choice depends on traversal information.
2. Sets focus to the control and returns `true`.
3. If no control can receive focus, returns `false`.

### WindowsFormsHost Registration

When the window handle to a [WindowsFormsHost](#) control is created, the [WindowsFormsHost](#) control calls an internal static method that registers its presence for the message loop.

During registration, the [WindowsFormsHost](#) control examines the message loop. If the message loop has not been started, the [ComponentDispatcher.ThreadFilterMessage](#) event handler is created. The message loop is considered to be running when the [ComponentDispatcher.ThreadFilterMessage](#) event handler is attached.

When the window handle is destroyed, the [WindowsFormsHost](#) control removes itself from registration.

## ElementHost Keyboard and Message Processing

When hosted by a Windows Forms application, WPF keyboard and message processing consists of the following:

- [HwndSource](#), [IKeyboardInputSink](#), and [IKeyboardInputSite](#) interface implementations.
- Tabbing and arrow keys.
- Command keys and dialog box keys.
- Windows Forms accelerator processing.

The following sections describe these parts in more detail.

### Interface Implementations

In Windows Forms, keyboard messages are routed to the window handle of the control that has focus. In the [ElementHost](#) control, these messages are routed to the hosted element. To accomplish this, the [ElementHost](#) control provides an [HwndSource](#) instance. If the [ElementHost](#) control has focus, the [HwndSource](#) instance routes most keyboard input so that it can be processed by the WPF [InputManager](#) class.

The [HwndSource](#) class implements the [IKeyboardInputSink](#) and [IKeyboardInputSite](#) interfaces.

Keyboard interoperation relies on implementing the [OnNoMoreTabStops](#) method to handle TAB key and arrow key input that moves focus out of hosted elements.

## Tabbing and Arrow Keys

The Windows Forms selection logic is mapped to the [IKeyboardInputSink.TabInto](#) and [OnNoMoreTabStops](#) methods to implement TAB and arrow key navigation. Overriding the [Select](#) method accomplishes this mapping.

## Command Keys and Dialog Box Keys

To give WPF the first opportunity to process command keys and dialog keys, Windows Forms command preprocessing is connected to the [TranslateAccelerator](#) method. Overriding the [Control.ProcessCmdKey](#) method connects the two technologies.

With the [TranslateAccelerator](#) method, the hosted elements can handle any key message, such as WM\_KEYDOWN, WM\_KEYUP, WM\_SYSKEYDOWN, or WM\_SYSKEYUP, including command keys, such as TAB, ENTER, ESC, and arrow keys. If a key message is not handled, it is sent up the Windows Forms ancestor hierarchy for handling.

## Accelerator Processing

To process accelerators correctly, Windows Forms accelerator processing must be connected to the WPF [AccessKeyManager](#) class. Additionally, all WM\_CHAR messages must be correctly routed to hosted elements.

Because the default [HwndSource](#) implementation of the [TranslateChar](#) method returns `false`, WM\_CHAR messages are processed using the following logic:

- The [Control.IsInputChar](#) method is overridden to ensure that all WM\_CHAR messages are forwarded to hosted elements.
- If the ALT key is pressed, the message is WM\_SYSCHAR. Windows Forms does not preprocess this message through the [IsInputChar](#) method. Therefore, the [ProcessMnemonic](#) method is overridden to query the WPF [AccessKeyManager](#) for a registered accelerator. If a registered accelerator is found, [AccessKeyManager](#) processes it.
- If the ALT key is not pressed, the WPF [InputManager](#) class processes the unhandled input. If the input is an accelerator, the [AccessKeyManager](#) processes it. The [PostProcessInput](#) event is handled for WM\_CHAR messages that were not processed.

When the user presses the ALT key, accelerator visual cues are shown on the whole form. To support this behavior, all [ElementHost](#) controls on the active form receive WM\_SYSKEYDOWN messages, regardless of which control has focus.

Messages are sent only to [ElementHost](#) controls in the active form.

## See also

- [EnableWindowsFormsInterop](#)
- [EnableModelessKeyboardInterop](#)
- [ElementHost](#)
- [WindowsFormsHost](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)
- [WPF and Win32 Interoperation](#)

# Layout Considerations for the WindowsFormsHost Element

4/28/2019 • 5 minutes to read • [Edit Online](#)

This topic describes how the [WindowsFormsHost](#) element interacts with the WPF layout system.

WPF and Windows Forms support different, but similar, logic for sizing and positioning elements on a form or page. When you create a hybrid user interface (UI) that hosts Windows Forms controls in WPF, the [WindowsFormsHost](#) element integrates the two layout schemes.

## Differences in Layout Between WPF and Windows Forms

WPF uses resolution-independent layout. All WPF layout dimensions are specified using *device-independent pixels*. A device-independent pixel is one ninety-sixth of an inch in size and resolution-independent, so you get similar results regardless of whether you are rendering to a 72-dpi monitor or a 19,200-dpi printer.

WPF is also based on *dynamic layout*. This means that a UI element arranges itself on a form or page according to its content, its parent layout container, and the available screen size. Dynamic layout facilitates localization by automatically adjusting the size and position of UI elements when the strings they contain change length.

Layout in Windows Forms is device-dependent and more likely to be static. Typically, Windows Forms controls are positioned absolutely on a form using dimensions specified in hardware pixels. However, Windows Forms does support some dynamic layout features, as summarized in the following table.

LAYOUT FEATURE	DESCRIPTION
Autosizing	Some Windows Forms controls resize themselves to display their contents properly. For more information, see <a href="#">AutoSize Property Overview</a> .
Anchoring and docking	Windows Forms controls support positioning and sizing based on the parent container. For more information, see <a href="#">Control.Anchor</a> and <a href="#">Control.Dock</a> .
Autoscaling	Container controls resize themselves and their children based on the resolution of the output device or the size, in pixels, of the default container font. For more information, see <a href="#">Automatic Scaling in Windows Forms</a> .
Layout containers	The <a href="#">FlowLayoutPanel</a> and <a href="#">TableLayoutPanel</a> controls arrange their child controls and size themselves according to their contents.

## Layout Limitations

In general, Windows Forms controls cannot be scaled and transformed to the extent possible in WPF. The following list describes the known limitations when the [WindowsFormsHost](#) element attempts to integrate its hosted Windows Forms control into the WPF layout system.

- In some cases, Windows Forms controls cannot be resized, or can be sized only to specific dimensions. For example, a Windows Forms [ComboBox](#) control supports only a single height, which is defined by the

control's font size. In a WPF dynamic layout where elements can stretch vertically, a hosted [ComboBox](#) control will not stretch as expected.

- Windows Forms controls cannot be rotated or skewed. The [WindowsFormsHost](#) element raises the [LayoutError](#) event if you apply a skew or rotation transformation. If you do not handle the [LayoutError](#) event, an [InvalidOperationException](#) is raised.
- In most cases, Windows Forms controls do not support proportional scaling. Although the overall dimensions of the control will scale, child controls and component elements of the control may not resize as expected. This limitation depends on how well each Windows Forms control supports scaling. In addition, you cannot scale Windows Forms controls down to a size of 0 pixels.
- Windows Forms controls support autoscaling, in which the form will automatically resize itself and its controls based on the font size. In a WPF user interface, changing the font size does not resize the entire layout, although individual elements may dynamically resize.

## Z-order

In a WPF user interface, you can change the z-order of elements to control overlapping behavior. A hosted Windows Forms control is drawn in a separate HWND, so it is always drawn on top of WPF elements.

A hosted Windows Forms control is also drawn on top of any [Adorner](#) elements.

# Layout Behavior

The following sections describe specific aspects of layout behavior when hosting Windows Forms controls in WPF.

## Scaling, Unit Conversion, and Device Independence

Whenever the [WindowsFormsHost](#) element performs operations involving WPF and Windows Forms dimensions, two coordinate systems are involved: device-independent pixels for WPF and hardware pixels for Windows Forms. Therefore, you must apply proper unit and scaling conversions to achieve a consistent layout.

Conversion between the coordinate systems depends on the current device resolution and any layout or rendering transforms applied to the [WindowsFormsHost](#) element or to its ancestors.

If the output device is 96 dpi and no scaling has been applied to the [WindowsFormsHost](#) element, one device-independent pixel is equal to one hardware pixel.

All other cases require coordinate system scaling. The hosted control is not resized. Instead, the [WindowsFormsHost](#) element attempts to scale the hosted control and all of its child controls. Because Windows Forms does not fully support scaling, the [WindowsFormsHost](#) element scales to the degree supported by particular controls.

Override the [ScaleChild](#) method to provide custom scaling behavior for the hosted Windows Forms control.

In addition to scaling, the [WindowsFormsHost](#) element handles rounding and overflow cases as described in the following table.

CONVERSION ISSUE	DESCRIPTION
Rounding	WPF device-independent pixel dimensions are specified as <code>double</code> , and Windows Forms hardware pixel dimensions are specified as <code>int</code> . In cases where <code>double</code> -based dimensions are converted to <code>int</code> -based dimensions, the <a href="#">WindowsFormsHost</a> element uses standard rounding, so that fractional values less than 0.5 are rounded down to 0.

CONVERSION ISSUE	DESCRIPTION
Overflow	When the <a href="#">WindowsFormsHost</a> element converts from <code>double</code> values to <code>int</code> values, overflow is possible. Values that are larger than <code>.MaxValue</code> are set to <code>.MaxValue</code> .

## Layout-related Properties

Properties that control layout behavior in Windows Forms controls and WPF elements are mapped appropriately by the [WindowsFormsHost](#) element. For more information, see [Windows Forms and WPF Property Mapping](#).

### Layout Changes in the Hosted Control

Layout changes in the hosted Windows Forms control are propagated to WPF to trigger layout updates. The [InvalidateMeasure](#) method on [WindowsFormsHost](#) ensures that layout changes in the hosted control cause the WPF layout engine to run.

### Continuously Sized Windows Forms Controls

Windows Forms controls that support continuous scaling fully interact with the WPF layout system. The [WindowsFormsHost](#) element uses the [MeasureOverride](#) and [ArrangeOverride](#) methods as usual to size and arrange the hosted Windows Forms control.

### Sizing Algorithm

The [WindowsFormsHost](#) element uses the following procedure to size the hosted control:

1. The [WindowsFormsHost](#) element overrides the [MeasureOverride](#) and [ArrangeOverride](#) methods.
2. To determine the size of the hosted control, the [MeasureOverride](#) method calls the hosted control's [GetPreferredSize](#) method with a constraint translated from the constraint passed to the [MeasureOverride](#) method.
3. The [ArrangeOverride](#) method attempts to set the hosted control to the given size constraint.
4. If the hosted control's [Size](#) property matches the specified constraint, the hosted control is sized to the constraint.

If the [Size](#) property does not match the specified constraint, the hosted control does not support continuous sizing. For example, the [MonthCalendar](#) control allows only discrete sizes. The permitted sizes for this control consist of integers (representing the number of months) for both height and width. In cases such as this, the [WindowsFormsHost](#) element behaves as follows:

- If the [Size](#) property returns a larger size than the specified constraint, the [WindowsFormsHost](#) element clips the hosted control. Height and width are handled separately, so the hosted control may be clipped in either direction.
- If the [Size](#) property returns a smaller size than the specified constraint, [WindowsFormsHost](#) accepts this size value and returns the value to the WPF layout system.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Walkthrough: Arranging Windows Forms Controls in WPF](#)
- [Arranging Windows Forms Controls in WPF Sample](#)
- [Windows Forms and WPF Property Mapping](#)
- [Migration and Interoperability](#)

# Windows Forms Controls and Equivalent WPF Controls

6/11/2019 • 2 minutes to read • [Edit Online](#)

Many Windows Forms controls have equivalent WPF controls, but some Windows Forms controls have no equivalents in WPF. This topic compares control types provided by the two technologies.

You can always use interoperation to host Windows Forms controls that do not have equivalents in your WPF-based applications.

The following table shows which Windows Forms controls and components have equivalent WPF control functionality.

WINDOWS FORMS CONTROL	WPF EQUIVALENT CONTROL	REMARKS
BindingNavigator	No equivalent control.	
BindingSource	CollectionViewSource	
Button	Button	
CheckBox	CheckBox	
CheckedListBox	ListBox with composition.	
ColorDialog	No equivalent control.	
ComboBox	ComboBox	ComboBox does not support auto-complete.
ContextMenuStrip	ContextMenu	
DataGridView	DataGrid	
DateTimePicker	DatePicker	
DomainUpDown	TextBox and two RepeatButton controls.	
ErrorProvider	No equivalent control.	
FlowLayoutPanel	WrapPanel or StackPanel	
FolderBrowserDialog	No equivalent control.	
FontDialog	No equivalent control.	
Form	Window	Window does not support child windows.

WINDOWS FORMS CONTROL	WPF EQUIVALENT CONTROL	REMARKS
GroupBox	GroupBox	
HelpProvider	No equivalent control.	No F1 Help. "What's This" Help is replaced by ToolTips.
HScrollBar	ScrollBar	Scrolling is built into container controls.
ImageList	No equivalent control.	
Label	Label	
LinkLabel	No equivalent control.	You can use the <a href="#">Hyperlink</a> class to host hyperlinks within flow content.
ListBox	ListBox	
ListView	ListView	The <a href="#">ListView</a> control provides a read-only details view.
MaskedTextBox	No equivalent control.	
MenuStrip	Menu	Menu control styling can approximate the behavior and appearance of the <a href="#">System.Windows.Forms.ToolStripProfessionalRenderer</a> class.
MonthCalendar	Calendar	
NotifyIcon	No equivalent control.	
NumericUpDown	TextBox and two <a href="#">RepeatButton</a> controls.	
OpenFileDialog	OpenFileDialog	The <a href="#">OpenFileDialog</a> class is a WPF wrapper around the Win32 control.
PageSetupDialog	No equivalent control.	
Panel	Canvas	
PictureBox	Image	
PrintDialog	PrintDialog	
PrintDocument	No equivalent control.	
PrintPreviewControl	DocumentViewer	
PrintPreviewDialog	No equivalent control.	
ProgressBar	ProgressBar	

WINDOWS FORMS CONTROL	WPF EQUIVALENT CONTROL	REMARKS
PropertyGrid	No equivalent control.	
RadioButton	RadioButton	
RichTextBox	RichTextBox	
SaveFileDialog	SaveFileDialog	The <a href="#">SaveFileDialog</a> class is a WPF wrapper around the Win32 control.
ScrollableControl	ScrollViewer	
SoundPlayer	MediaPlayer	
SplitContainer	GridSplitter	
StatusStrip	StatusBar	
TabControl	TabControl	
TableLayoutPanel	Grid	
TextBox	TextBox	
Timer	DispatcherTimer	
ToolStrip	ToolBar	
ToolStripContainer	ToolBar with composition.	
ToolStripDropDown	ToolBar with composition.	
ToolStripDropDownMenu	ToolBar with composition.	
ToolStripPanel	ToolBar with composition.	
ToolTip	ToolTip	
TrackBar	Slider	
TreeView	TreeView	
UserControl	UserControl	
VScrollBar	ScrollBar	Scrolling is built into container controls.

WINDOWS FORMS CONTROL	WPF EQUIVALENT CONTROL	REMARKS
WebBrowser	Frame, <code>System.Windows.Controls.WebBrowser</code>	<p>The <a href="#">Frame</a> control can host HTML pages.</p> <p>Starting in the .NET Framework 3.5 SP1, the <a href="#">System.Windows.Controls.WebBrowser</a> control can host HTML pages and also backs the <a href="#">Frame</a> control.</p>

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [WPF Designer for Windows Forms Developers](#)
- [Walkthrough: Hosting a Windows Forms Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)
- [Migration and Interoperability](#)

# Windows Forms and WPF Property Mapping

8/22/2019 • 5 minutes to read • [Edit Online](#)

The Windows Forms and WPF technologies have two similar but different property models. *Property mapping* supports interoperation between the two architectures and provides the following capabilities:

- Makes it easy to map relevant property changes in the host environment to the hosted control or element.
- Provides default handling for mapping the most commonly used properties.
- Allows easy removal, overriding, or extending of default properties.
- Ensures that property value changes on the host are automatically detected and translated to the hosted control or element.

## NOTE

Property-change events are not propagated up the hosting control or element hierarchy. Property translation is not performed if the local value of a property does not change because of direct setting, styles, inheritance, data binding, or other mechanisms that change the value of the property.

Use the [PropertyMap](#) property on the [WindowsFormsHost](#) element and the [PropertyMap](#) property on [ElementHost](#) control to access property mapping.

## Property Mapping with the WindowsFormsHost Element

The [WindowsFormsHost](#) element translates default WPF properties to their Windows Forms equivalents using the following translation table.

WINDOWS PRESENTATION FOUNDATION HOSTING	WINDOWS FORMS	INTEROPERATION BEHAVIOR

WINDOWS PRESENTATION FOUNDATION HOSTING	WINDOWS FORMS	INTEROPERATION BEHAVIOR
Background  (System.Windows.Media.Brush)	BackColor  (System.Drawing.Color)	<p>The <a href="#">WindowsFormsHost</a> element sets the <a href="#">BackColor</a> property of the hosted control and the <a href="#">BackgroundImage</a> property of the hosted control. Mapping is performed by using the following rules:</p> <ul style="list-style-type: none"> <li>- If <a href="#">Background</a> is a solid color, it is converted and used to set the <a href="#">BackColor</a> property of the hosted control. The <a href="#">BackColor</a> property is not set on the hosted control, because the hosted control can inherit the value of the <a href="#">BackColor</a> property. <b>Note:</b> The hosted control does not support transparency. Any color assigned to <a href="#">BackColor</a> must be fully opaque, with an alpha value of 0xFF.</li> <li>- If <a href="#">Background</a> is not a solid color, the <a href="#">WindowsFormsHost</a> control creates a bitmap from the <a href="#">Background</a> property. The <a href="#">WindowsFormsHost</a> control assigns this bitmap to the <a href="#">BackgroundImage</a> property of the hosted control. This provides an effect which is similar to transparency. <b>Note:</b> You can override this behavior or you can remove the <a href="#">Background</a> property mapping.</li> </ul>
Cursor	Cursor	<p>If the default mapping has not been reassigned, <a href="#">WindowsFormsHost</a> control traverses its ancestor hierarchy until it finds an ancestor with its <a href="#">Cursor</a> property set. This value is translated to the closest corresponding Windows Forms cursor.</p> <p>If the default mapping for the <a href="#">ForceCursor</a> property has not been reassigned, the traversal stops on the first ancestor with <a href="#">ForceCursor</a> set to <code>true</code>.</p>
FlowDirection  (System.Windows.FlowDirection)	RightToLeft  (System.Windows.Forms.RightToLeft)	<p><a href="#">LeftToRight</a> maps to <a href="#">No</a>.</p> <p><a href="#">RightToLeft</a> maps to <a href="#">Yes</a>.</p> <p><a href="#">Inherit</a> is not mapped.</p> <p><a href="#">FlowDirection.RightToLeft</a> maps to <a href="#">RightToLeft.Yes</a>.</p>

WINDOWS PRESENTATION FOUNDATION HOSTING	WINDOWS FORMS	INTEROPERATION BEHAVIOR
FontStyle	Style on the hosted control's System.Drawing.Font	The set of WPF properties is translated into a corresponding Font. When one of these properties changes, a new Font is created. For Normal: Italic is disabled. For Italic or Oblique: Italic is enabled.
FontWeight	Style on the hosted control's System.Drawing.Font	The set of WPF properties is translated into a corresponding Font. When one of these properties changes, a new Font is created. For Black, Bold, DemiBold, ExtraBold, Heavy, Medium, SemiBold, or UltraBold: Bold is enabled. For ExtraLight, Light, Normal, Regular, Thin, or UltraLight: Bold is disabled.
FontFamily	Font	The set of WPF properties is translated into a corresponding Font. When one of these properties changes, a new Font is created. The hosted Windows Forms control resizes based on the font size.
FontSize	(System.Drawing.Font)	Font size in WPF is expressed as one ninety-sixth of an inch, and in Windows Forms as one seventy-second of an inch. The corresponding conversion is:  Windows Forms font size = WPF font size * 72.0 / 96.0.
FontStretch		
FontStyle		
FontWeight		
Foreground (System.Windows.Media.Brush)	ForeColor (System.Drawing.Color)	The Foreground property mapping is performed by using the following rules:  - If Foreground is a SolidColorBrush, use Color for ForeColor. - If Foreground is a GradientBrush, use the color of the GradientStop with the lowest offset value for ForeColor. - For any other Brush type, leave ForeColor unchanged. This means the default is used.
IsEnabled	Enabled	When IsEnabled is set, WindowsFormsHost element sets the Enabled property on the hosted control.
Padding	Padding	All four values of the Padding property on the hosted Windows Forms control are set to the same Thickness value.  - Values greater than MaxValue are set to MaxValue. - Values less than MinValue are set to MinValue.

WINDOWS PRESENTATION FOUNDATION HOSTING	WINDOWS FORMS	INTEROPERATION BEHAVIOR
Visibility	Visible	<ul style="list-style-type: none"> <li>- <b>Visible</b> maps to <code>Visible = true</code>. The hosted Windows Forms control is visible. Explicitly setting the <b>Visible</b> property on the hosted control to <code>false</code> is not recommended.</li> <li>- <b>Collapsed</b> maps to <code>Visible = true</code> or <code>false</code>. The hosted Windows Forms control is not drawn, and its area is collapsed.</li> <li>- <b>Hidden</b> : The hosted Windows Forms control occupies space in the layout, but is not visible. In this case, the <b>Visible</b> property is set to <code>true</code>. Explicitly setting the <b>Visible</b> property on the hosted control to <code>false</code> is not recommended.</li> </ul>

Attached properties on container elements are fully supported by the [WindowsFormsHost](#) element.

For more information, see [Walkthrough: Mapping Properties Using the WindowsFormsHost Element](#).

## Updates to Parent Properties

Changes to most parent properties cause notifications to the hosted child control. The following list describes properties which do not cause notifications when their values change.

- [Background](#)
- [Cursor](#)
- [ForceCursor](#)
- [Visibility](#)

For example, if you change the value of the [Background](#) property of the [WindowsFormsHost](#) element, the [BackColor](#) property of the hosted control does not change.

## Property Mapping with the ElementHost Control

The following properties provide built-in change notification. Do not call the [OnPropertyChanged](#) method when you are mapping these properties:

- [AutoSize](#)
- [BackColor](#)
- [BackgroundImage](#)
- [BackgroundImageLayout](#)
- [BindingContext](#)
- [CausesValidation](#)
- [ContextMenu](#)
- [ContextMenuStrip](#)
- [Cursor](#)

- Dock
- Enabled
- Font
- ForeColor
- Location
- Margin
- Padding
- Parent
- Region
- RightToLeft
- Size
- TabIndex
- TabStop
- Text
- Visible

The [ElementHost](#) control translates default Windows Forms properties to their WPF equivalents by using the following translation table.

For more information, see [Walkthrough: Mapping Properties Using the ElementHost Control](#).

WINDOWS FORMS HOSTING	WINDOWS PRESENTATION FOUNDATION	INTEROPERATION BEHAVIOR
<a href="#">BackColor</a>  <a href="#">(System.Drawing.Color)</a>	<a href="#">Background</a>  <a href="#">(System.Windows.Media.Brush)</a> on the hosted element	Setting this property forces a repaint with an <a href="#">ImageBrush</a> . If the <a href="#">BackColorTransparent</a> property is set to <code>false</code> (the default value), this <a href="#">ImageBrush</a> is based on the appearance of the <a href="#">ElementHost</a> control, including its <a href="#">BackColor</a> , <a href="#">BackgroundImage</a> , <a href="#">BackgroundImageLayout</a> properties, and any attached paint handlers.  If the <a href="#">BackColorTransparent</a> property is set to <code>true</code> , the <a href="#">ImageBrush</a> is based on the appearance of the <a href="#">ElementHost</a> control's parent, including the parent's <a href="#">BackColor</a> , <a href="#">BackgroundImage</a> , <a href="#">BackgroundImageLayout</a> properties, and any attached paint handlers.
<a href="#">BackgroundImage</a>  <a href="#">(System.Drawing.Image)</a>	<a href="#">Background</a>  <a href="#">(System.Windows.Media.Brush)</a> on the hosted element	Setting this property causes the same behavior described for the <a href="#">BackColor</a> mapping.

WINDOWS FORMS HOSTING	WINDOWS PRESENTATION FOUNDATION	INTEROPERATION BEHAVIOR
BackgroundImageLayout	Background  (System.Windows.Media.Brush) on the hosted element	Setting this property causes the same behavior described for the BackColor mapping.
Cursor  (System.Windows.Forms.Cursor)	Cursor  (System.Windows.Input.Cursor)	The Windows Forms standard cursor is translated to the corresponding WPF standard cursor. If the Windows Forms is not a standard cursor, the default is assigned.
Enabled	IsEnabled	When Enabled is set, the ElementHost control sets the IsEnabled property on the hosted element.
Font  (System.Drawing.Font)	FontFamily  FontSize  FontStretch  FontStyle  FontWeight	The Font value is translated into a corresponding set of WPF font properties.
Bold	FontWeight on hosted element	If Bold is true, FontWeight is set to Bold.  If Bold is false, FontWeight is set to Normal.
Italic	FontStyle on hosted element	If Italic is true, FontStyle is set to Italic.  If Italic is false, FontStyle is set to Normal.
Strikeout	TextDecorations on hosted element	Applies only when hosting a TextBlock control.
Underline	TextDecorations on hosted element	Applies only when hosting a TextBlock control.
RightToLeft  (System.Windows.Forms.RightToLeft)	FlowDirection  (FlowDirection)	No maps to LeftToRight.  Yes maps to RightToLeft.
Visible	Visibility	The ElementHost control sets the Visibility property on the hosted element by using the following rules:  - Visible = true maps to Visible. - Visible = false maps to Hidden.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [WPF and Win32 Interoperation](#)
- [WPF and Windows Forms Interoperation](#)
- [Walkthrough: Mapping Properties Using the WindowsFormsHost Element](#)
- [Walkthrough: Mapping Properties Using the ElementHost Control](#)

# Troubleshooting Hybrid Applications

12/10/2019 • 6 minutes to read • [Edit Online](#)

This topic lists some common problems that can occur when authoring hybrid applications, which use both WPF and Windows Forms technologies.

## Overlapping Controls

Controls may not overlap as you would expect. Windows Forms uses a separate HWND for each control. WPF uses one HWND for all content on a page. This implementation difference causes unexpected overlapping behaviors.

A Windows Forms control hosted in WPF always appears on top of the WPF content.

WPF content hosted in an [ElementHost](#) control appears at the z-order of the [ElementHost](#) control. It is possible to overlap [ElementHost](#) controls, but the hosted WPF content does not combine or interact.

## Child Property

The [WindowsFormsHost](#) and [ElementHost](#) classes can host only a single child control or element. To host more than one control or element, you must use a container as the child content. For example, you could add Windows Forms button and check box controls to a [System.Windows.Forms.Panel](#) control, and then assign the panel to a [WindowsFormsHost](#) control's [Child](#) property. However, you cannot add the button and check box controls separately to the same [WindowsFormsHost](#) control.

## Scaling

WPF and Windows Forms have different scaling models. Some WPF scaling transformations are meaningful to Windows Forms controls, but others are not. For example, scaling a Windows Forms control to 0 will work, but if you try to scale the same control back to a non-zero value, the control's size remains 0. For more information, see [Layout Considerations for the WindowsFormsHost Element](#).

## Adapter

There may be confusion when working the [WindowsFormsHost](#) and [ElementHost](#) classes, because they include a hidden container. Both the [WindowsFormsHost](#) and [ElementHost](#) classes have a hidden container, called an *adapter*, which they use to host content. For the [WindowsFormsHost](#) element, the adapter derives from the [System.Windows.Forms.ContainerControl](#) class. For the [ElementHost](#) control, the adapter derives from the [DockPanel](#) element. When you see references to the adapter in other interoperation topics, this container is what is being discussed.

## Nesting

Nesting a [WindowsFormsHost](#) element inside an [ElementHost](#) control is not supported. Nesting an [ElementHost](#) control inside a [WindowsFormsHost](#) element is also not supported.

## Focus

Focus works differently in WPF and Windows Forms, which means that focus issues may occur in a hybrid application. For example, if you have focus inside a [WindowsFormsHost](#) element, and you either minimize and

restore the page or show a modal dialog box, focus inside the [WindowsFormsHost](#) element may be lost. The [WindowsFormsHost](#) element still has focus, but the control inside it may not.

Data validation is also affected by focus. Validation works in a [WindowsFormsHost](#) element, but it does not work as you tab out of the [WindowsFormsHost](#) element, or between two different [WindowsFormsHost](#) elements.

## Property Mapping

Some property mappings require extensive interpretation to bridge dissimilar implementations between the WPF and Windows Forms technologies. Property mappings enable your code to react to changes in fonts, colors, and other properties. In general, property mappings work by listening for either *PropertyChanged* events or *OnPropertyChanged* calls, and setting appropriate properties on either the child control or its adapter. For more information, see [Windows Forms and WPF Property Mapping](#).

## Layout-related Properties on Hosted Content

When the [WindowsFormsHost.Child](#) or [ElementHost.Child](#) property is assigned, several layout-related properties on the hosted content are set automatically. Changing these content properties can cause unexpected layout behaviors.

Your hosted content is docked to fill the [WindowsFormsHost](#) and [ElementHost](#) parent. To enable this fill behavior, several properties are set when you set the child property. The following table lists which content properties are set by the [ElementHost](#) and [WindowsFormsHost](#) classes.

HOST CLASS	CONTENT PROPERTIES
<a href="#">ElementHost</a>	<a href="#">Height</a> <a href="#">Width</a> <a href="#">Margin</a> <a href="#">VerticalAlignment</a> <a href="#">HorizontalAlignment</a>
<a href="#">WindowsFormsHost</a>	<a href="#">Margin</a> <a href="#">Dock</a> <a href="#">AutoSize</a> <a href="#">Location</a> <a href="#">MaximumSize</a>

Do not set these properties directly on the hosted content. For more information, see [Layout Considerations for the WindowsFormsHost Element](#).

## Navigation Applications

Navigation applications may not maintain user state. The [WindowsFormsHost](#) element recreates its controls when it is used in a navigation application. Recreating child controls occurs when the user navigates away from the page hosting the [WindowsFormsHost](#) element and then returns to it. Any content that has been typed in by the user will be lost.

## Message Loop Interoperation

When working with Windows Forms message loops, messages may not be processed as expected. The [EnableWindowsFormsInterop](#) method is called by the [WindowsFormsHost](#) constructor. This method adds a message filter to the WPF message loop. This filter calls the [Control.PreProcessMessage](#) method if a [System.Windows.Forms.Control](#) was the target of the message and translates/dispatches the message.

If you show a [Window](#) in a Windows Forms message loop with [Application.Run](#), you cannot type anything unless you call the [EnableModelessKeyboardInterop](#) method. The [EnableModelessKeyboardInterop](#) method takes a [Window](#) and adds a [System.Windows.Forms.IMessageFilter](#), which reroutes key-related messages to the WPF message loop. For more information, see [Windows Forms and WPF Interoperability Input Architecture](#).

## Opacity and Layering

The [HwndHost](#) class does not support layering. This means that setting the [Opacity](#) property on the [WindowsFormsHost](#) element has no effect, and no blending will occur with other WPF windows which have [AllowsTransparency](#) set to `true`.

## Dispose

Not disposing classes properly can leak resources. In your hybrid applications, make sure that the [WindowsFormsHost](#) and [ElementHost](#) classes are disposed, or you could leak resources. Windows Forms disposes [ElementHost](#) controls when its non-modal [Form](#) parent closes. WPF disposes [WindowsFormsHost](#) elements when your application shuts down. It is possible to show a [WindowsFormsHost](#) element in a [Window](#) in a Windows Forms message loop. In this case, your code may not receive notification that your application is shutting down.

## Enabling Visual Styles

Microsoft Windows XP visual styles on a Windows Forms control may not be enabled. The [Application.EnableVisualStyles](#) method is called in the template for a Windows Forms application. Although this method is not called by default, if you use Visual Studio to create a project, you will get Microsoft Windows XP visual styles for controls, if version 6.0 of Comctl32.dll is available. You must call the [EnableVisualStyles](#) method before handles are created on the thread. For more information, see [How to: Enable Visual Styles in a Hybrid Application](#).

## Licensed Controls

Licensed Windows Forms controls that display licensing information in a message box to the user might cause unexpected behavior for a hybrid application. Some licensed controls show a dialog box in response to handle creation. For example, a licensed control might inform the user that a license is required, or that the user has three remaining trial uses of the control.

The [WindowsFormsHost](#) element derives from the [HwndHost](#) class, and the child control's handle is created inside the [BuildWindowCore](#) method. The [HwndHost](#) class does not allow messages to be processed in the [BuildWindowCore](#) method, but displaying a dialog box causes messages to be sent. To enable this licensing scenario, call the [Control.CreateControl](#) method on the control before assigning it as the [WindowsFormsHost](#) element's child.

## WPF Designer

You can design your WPF content by using the WPF Designer for Visual Studio. The following sections list some common problems that can occur when authoring hybrid applications with the WPF Designer.

### **BackColorTransparent is ignored at design time**

The [BackColorTransparent](#) property might not work as expected at design time.

If a WPF control is not on a visible parent, the WPF runtime ignores the [BackColorTransparent](#) value. The reason that [BackColorTransparent](#) might be ignored is because [ElementHost](#) object is created in a separate [AppDomain](#). However, when you run the application, [BackColorTransparent](#) does work as expected.

### Design-time Error List appears when the obj folder is deleted

If the obj folder is deleted, the Design-time Error List appears.

When you design using [ElementHost](#), the Windows Forms Designer uses generated files in the Debug or Release folder within your project's obj folder. If you delete these files, the Design-time Error List appears. To fix this problem, rebuild your project. For more information, see [Design-Time Errors in the Windows Forms Designer](#).

## ElementHost and IME

WPF controls hosted in an [ElementHost](#) currently do not support the [ImeMode](#) property. Changes to [ImeMode](#) will be ignored by the hosted controls.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Interoperability in the WPF Designer](#)
- [Windows Forms and WPF Interoperability Input Architecture](#)
- [How to: Enable Visual Styles in a Hybrid Application](#)
- [Layout Considerations for the WindowsFormsHost Element](#)
- [Windows Forms and WPF Property Mapping](#)
- [Design-Time Errors in the Windows Forms Designer](#)
- [Migration and Interoperability](#)

# Walkthrough: Hosting a Windows Forms Control in WPF

11/12/2019 • 2 minutes to read • [Edit Online](#)

WPF provides many controls with a rich feature set. However, you may sometimes want to use Windows Forms controls on your WPF pages. For example, you may have a substantial investment in existing Windows Forms controls, or you may have a Windows Forms control that provides unique functionality.

This walkthrough shows you how to host a Windows Forms `System.Windows.Forms.MaskedTextBox` control on a WPF page by using code.

For a complete code listing of the tasks shown in this walkthrough, see [Hosting a Windows Forms Control in WPF Sample](#).

## Prerequisites

You need Visual Studio to complete this walkthrough.

## Hosting the Windows Forms Control

### To host the `MaskedTextBox` control

1. Create a WPF Application project named `HostingWFInWpf`.
2. Add references to the following assemblies.
  - `WindowsFormsIntegration`
  - `System.Windows.Forms`
3. Open `MainWindow.xaml` in the WPF Designer.
4. Name the `Grid` element `grid1`.

```
<Grid Name="grid1">  
  </Grid>
```

5. In Design view or XAML view, select the `Window` element.
6. In the Properties window, click the **Events** tab.
7. Double-click the `Loaded` event.
8. Insert the following code to handle the `Loaded` event.

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Create the interop host control.
    System.Windows.Forms.Integration.WindowsFormsHost host =
        new System.Windows.Forms.Integration.WindowsFormsHost();

    // Create the MaskedTextBox control.
    MaskedTextBox mtbDate = new MaskedTextBox("00/00/0000");

    // Assign the MaskedTextBox control as the host control's child.
    host.Child = mtbDate;

    // Add the interop host control to the Grid
    // control's collection of child controls.
    this.grid1.Children.Add(host);
}

```

```

Private Sub Window_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Create the interop host control.
    Dim host As New System.Windows.Forms.Integration.WindowsFormsHost()

    ' Create the MaskedTextBox control.
    Dim mtbDate As New MaskedTextBox("00/00/0000")

    ' Assign the MaskedTextBox control as the host control's child.
    host.Child = mtbDate

    ' Add the interop host control to the Grid
    ' control's collection of child controls.
    Me.grid1.Children.Add(host)

End Sub

```

- At the top of the file, add the following `Imports` or `using` statement.

```
using System.Windows.Forms;
```

```
Imports System.Windows.Forms
```

- Press **F5** to build and run the application.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Walkthrough: Hosting a Windows Forms Control in WPF by Using XAML](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)
- [Windows Forms Controls and Equivalent WPF Controls](#)
- [Hosting a Windows Forms Control in WPF Sample](#)

# Walkthrough: Hosting a Windows Forms Control in WPF by Using XAML

11/12/2019 • 2 minutes to read • [Edit Online](#)

WPF provides many controls with a rich feature set. However, you may sometimes want to use Windows Forms controls on your WPF pages. For example, you may have a substantial investment in existing Windows Forms controls, or you may have a Windows Forms control that provides unique functionality.

This walkthrough shows you how to host a Windows Forms `System.Windows.Forms.MaskedTextBox` control on a WPF page by using XAML.

For a complete code listing of the tasks shown in this walkthrough, see [Hosting a Windows Forms Control in WPF by Using XAML Sample](#).

## Prerequisites

You need Visual Studio to complete this walkthrough.

## Hosting the Windows Forms Control

### To host the `MaskedTextBox` control

1. Create a WPF Application project named `HostingWfInWpfWithXaml`.
2. Add references to the following assemblies.
  - `WindowsFormsIntegration`
  - `System.Windows.Forms`
3. Open `MainWindow.xaml` in the WPF Designer.
4. In the `Window` element, add the following namespace mapping. The `wf` namespace mapping establishes a reference to the assembly that contains the Windows Forms control.

```
xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

5. In the `Grid` element add the following XAML.

The `MaskedTextBox` control is created as a child of the `WindowsFormsHost` control.

```
<Grid>
    <WindowsFormsHost>
        <wf:MaskedTextBox x:Name="mtbDate" Mask="00/00/0000"/>
    </WindowsFormsHost>
</Grid>
```

6. Press F5 to build and run the application.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Walkthrough: Hosting a Windows Forms Control in WPF](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)
- [Windows Forms Controls and Equivalent WPF Controls](#)
- [Hosting a Windows Forms Control in WPF by Using XAML Sample](#)

# Walkthrough: Hosting a Windows Forms Composite Control in WPF

10/31/2019 • 15 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a rich environment for creating applications. However, when you have a substantial investment in Windows Forms code, it can be more effective to reuse at least some of that code in your WPF application rather than to rewrite it from scratch. The most common scenario is when you have existing Windows Forms controls. In some cases, you might not even have access to the source code for these controls. WPF provides a straightforward procedure for hosting such controls in a WPF application. For example, you can use WPF for most of your programming while hosting your specialized [DataGridView](#) controls.

This walkthrough steps you through an application that hosts a Windows Forms composite control to perform data entry in a WPF application. The composite control is packaged in a DLL. This general procedure can be extended to more complex applications and controls. This walkthrough is designed to be nearly identical in appearance and functionality to [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#). The primary difference is that the hosting scenario is reversed.

The walkthrough is divided into two sections. The first section briefly describes the implementation of the Windows Forms composite control. The second section discusses in detail how to host the composite control in a WPF application, receive events from the control, and access some of the control's properties.

Tasks illustrated in this walkthrough include:

- Implementing the Windows Forms composite control.
- Implementing the WPF host application.

For a complete code listing of the tasks illustrated in this walkthrough, see [Hosting a Windows Forms Composite Control in WPF Sample](#).

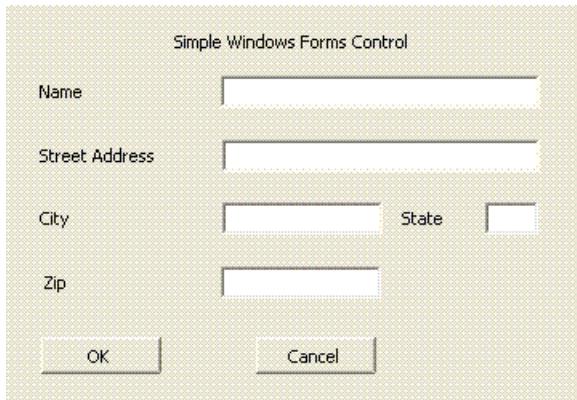
## Prerequisites

You need Visual Studio to complete this walkthrough.

## Implementing the Windows Forms Composite Control

The Windows Forms composite control used in this example is a simple data-entry form. This form takes the user's name and address and then uses a custom event to return that information to the host. The following illustration shows the rendered control.

The following image shows a Windows Forms composite control:



## Creating the Project

To start the project:

1. Launch Visual Studio, and open the **New Project** dialog box.
2. In the Window category, select the **Windows Forms Control Library** template.
3. Name the new project `MyControls`.
4. For the location, specify a conveniently named top-level folder, such as `WpfHostingWindowsFormsControl`.  
Later, you will put the host application in this folder.
5. Click **OK** to create the project. The default project contains a single control named `UserControl1`.
6. In Solution Explorer, rename `UserControl1` to `MyControl1`.

Your project should have references to the following system DLLs. If any of these DLLs are not included by default, add them to the project.

- System
- System.Data
- System.Drawing
- System.Windows.Forms
- System.Xml

## Adding Controls to the Form

To add controls to the form:

- Open `MyControl1` in the designer.

Add five **Label** controls and their corresponding **TextBox** controls, sized and arranged as they are in the preceding illustration, on the form. In the example, the **TextBox** controls are named:

- `txtName`
- `txtAddress`
- `txtCity`
- `txtState`
- `txtZip`

Add two **Button** controls labeled **OK** and **Cancel**. In the example, the button names are `btnOK` and `btnCancel`, respectively.

## Implementing the Supporting Code

Open the form in code view. The control returns the collected data to its host by raising the custom `OnButtonClick` event. The data is contained in the event argument object. The following code shows the event and delegate declaration.

Add the following code to the `MyControl1` class.

```
public delegate void MyControlEventHandler(object sender, MyControlEvents args);  
public event MyControlEventHandler OnButtonClick;
```

```
Public Delegate Sub MyControlEventHandler(ByVal sender As Object, ByVal args As MyControlEvents)  
Public Event OnButtonClick As MyControlEventHandler
```

The `MyControlEvents` class contains the information to be returned to the host.

Add the following class to the form.

```

public class MyControlEvents : EventArgs
{
    private string _Name;
    private string _StreetAddress;
    private string _City;
    private string _State;
    private string _Zip;
    private bool _IsOK;

    public MyControlEvents(bool result,
                          string name,
                          string address,
                          string city,
                          string state,
                          string zip)
    {
        _IsOK = result;
        _Name = name;
        _StreetAddress = address;
        _City = city;
        _State = state;
        _Zip = zip;
    }

    public string MyName
    {
        get { return _Name; }
        set { _Name = value; }
    }
    public string MyStreetAddress
    {
        get { return _StreetAddress; }
        set { _StreetAddress = value; }
    }
    public string MyCity
    {
        get { return _City; }
        set { _City = value; }
    }
    public string MyState
    {
        get { return _State; }
        set { _State = value; }
    }
    public string MyZip
    {
        get { return _Zip; }
        set { _Zip = value; }
    }
    public bool IsOK
    {
        get { return _IsOK; }
        set { _IsOK = value; }
    }
}

```

```

Public Class MyControlEvents
    Inherits EventArgs
    Private _Name As String
    Private _StreetAddress As String
    Private _City As String
    Private _State As String
    Private _Zip As String
    Private _IsOK As Boolean

```

```

    Public Sub New(ByVal result As Boolean, ByVal name As String, ByVal address As String, ByVal city As
String, ByVal state As String, ByVal zip As String)
        _IsOK = result
        _Name = name
        _StreetAddress = address
        _City = city
        _State = state
        _Zip = zip

    End Sub

    Public Property MyName() As String
        Get
            Return _Name
        End Get
        Set
            _Name = value
        End Set
    End Property

    Public Property MyStreetAddress() As String
        Get
            Return _StreetAddress
        End Get
        Set
            _StreetAddress = value
        End Set
    End Property

    Public Property MyCity() As String
        Get
            Return _City
        End Get
        Set
            _City = value
        End Set
    End Property

    Public Property MyState() As String
        Get
            Return _State
        End Get
        Set
            _State = value
        End Set
    End Property

    Public Property MyZip() As String
        Get
            Return _Zip
        End Get
        Set
            _Zip = value
        End Set
    End Property

    Public Property IsOK() As Boolean
        Get
            Return _IsOK
        End Get
        Set
            _IsOK = value
        End Set
    End Property
End Class

```

When the user clicks the **OK** or **Cancel** button, the `Click` event handlers create a `MyControlEventArgs` object that contains the data and raises the `OnButtonClick` event. The only difference between the two handlers is the event argument's `IsOK` property. This property enables the host to determine which button was clicked. It is set to `true` for the **OK** button, and `false` for the **Cancel** button. The following code shows the two button handlers.

Add the following code to the `MyControl1` class.

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    MyControlEventArgs retvals = new MyControlEventArgs(true,
        txtName.Text,
        txtAddress.Text,
        txtCity.Text,
        txtState.Text,
        txtZip.Text);
    OnButtonClick(this, retvals);
}

private void btnCancel_Click(object sender, System.EventArgs e)
{
    MyControlEventArgs retvals = new MyControlEventArgs(false,
        txtName.Text,
        txtAddress.Text,
        txtCity.Text,
        txtState.Text,
        txtZip.Text);
    OnButtonClick(this, retvals);
}
```

```
Private Sub btnOK_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnOK.Click
    Dim retvals As New MyControlEventArgs(True, txtName.Text, txtAddress.Text, txtCity.Text, txtState.Text,
    txtZip.Text)
    RaiseEvent OnButtonClick(Me, retvals)
End Sub

Private Sub btnCancel_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnCancel.Click
    Dim retvals As New MyControlEventArgs(False, txtName.Text, txtAddress.Text, txtCity.Text, txtState.Text,
    txtZip.Text)
    RaiseEvent OnButtonClick(Me, retvals)
End Sub
```

## Giving the Assembly a Strong Name and Building the Assembly

For this assembly to be referenced by a WPF application, it must have a strong name. To create a strong name, create a key file with `Sn.exe` and add it to your project.

1. Open a Visual Studio command prompt. To do so, click the **Start** menu, and then select **All Programs/Microsoft Visual Studio 2010/Visual Studio Tools/Visual Studio Command Prompt**. This launches a console window with customized environment variables.
2. At the command prompt, use the `cd` command to go to your project folder.
3. Generate a key file named `MyControls.snk` by running the following command.

```
Sn.exe -k MyControls.snk
```

4. To include the key file in your project, right-click the project name in Solution Explorer and then click

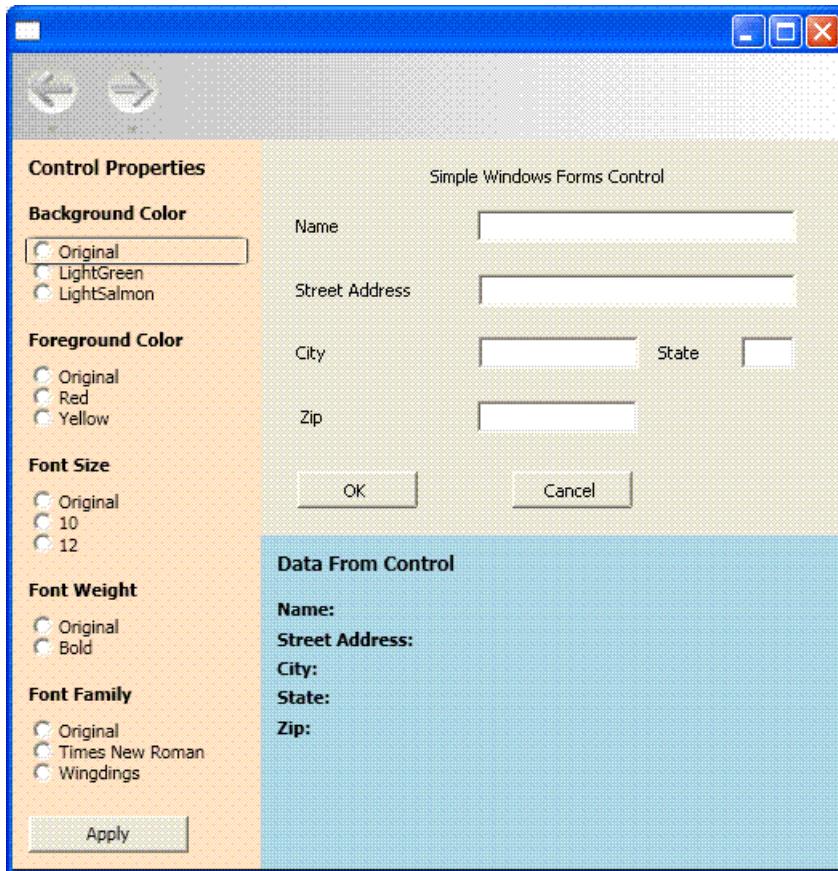
**Properties.** In the Project Designer, click the **Signing** tab, select the **Sign the assembly** check box and then browse to your key file.

5. Build the solution. The build will produce a DLL named MyControls.dll.

## Implementing the WPF Host Application

The WPF host application uses the [WindowsFormsHost](#) control to host [MyControl1](#). The application handles the [OnButtonClick](#) event to receive the data from the control. It also has a collection of option buttons that enable you to change some of the control's properties from the WPF application. The following illustration shows the finished application.

The following image shows the complete application, including the control embedded in the WPF application:



## Creating the Project

To start the project:

1. Open Visual Studio, and select **New Project**.
2. In the Window category, select the **WPF Application** template.
3. Name the new project [WpfHost](#).
4. For the location, specify the same top-level folder that contains the MyControls project.
5. Click **OK** to create the project.

You also need to add references to the DLL that contains [MyControl1](#) and other assemblies.

1. Right-click the project name in Solution Explorer and select **Add Reference**.
2. Click the **Browse** tab, and browse to the folder that contains MyControls.dll. For this walkthrough, this folder is MyControls\bin\Debug.

3. Select MyControls.dll, and then click **OK**.
4. Add a reference to the WindowsFormsIntegration assembly, which is named WindowsFormsIntegration.dll.

## Implementing the Basic Layout

The user interface (UI) of the host application is implemented in MainWindow.xaml. This file contains Extensible Application Markup Language (XAML) markup that defines the layout, and hosts the Windows Forms control. The application is divided into three regions:

- The **Control Properties** panel, which contains a collection of option buttons that you can use to modify various properties of the hosted control.
- The **Data from Control** panel, which contains several **TextBlock** elements that display the data returned from the hosted control.
- The hosted control itself.

The basic layout is shown in the following XAML. The markup that is needed to host **MyControl1** is omitted from this example, but will be discussed later.

Replace the XAML in MainWindow.xaml with the following. If you are using Visual Basic, change the class to

```
x:Class="MainWindow".
```

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WpfHost.MainWindow"
    xmlns:mcl="clr-namespace:MyControls;assembly=MyControls"
    Loaded="Init">
    <DockPanel>
        <DockPanel.Resources>
            <Style x:Key="inlineText" TargetType="{x:Type Inline}">
                <Setter Property="FontWeight" Value="Normal"/>
            </Style>
            <Style x:Key="titleText" TargetType="{x:Type TextBlock}">
                <Setter Property="DockPanel.Dock" Value="Top"/>
                <Setter Property="FontWeight" Value="Bold"/>
                <Setter Property="Margin" Value="10,5,10,0"/>
            </Style>
        </DockPanel.Resources>

        <StackPanel Orientation="Vertical"
            DockPanel.Dock="Left"
            Background="Bisque"
            Width="250">

            <TextBlock Margin="10,10,10,10"
                FontWeight="Bold"
                FontSize="12">Control Properties</TextBlock>
            <TextBlock Style="{StaticResource titleText}">Background Color</TextBlock>
            <StackPanel Margin="10,10,10,10">
                <RadioButton Name="rdbtnOriginalBackColor"
                    IsChecked="True"
                    Click="BackColorChanged">Original</RadioButton>
                <RadioButton Name="rdbtnBackGreen"
                    Click="BackColorChanged">LightGreen</RadioButton>
                <RadioButton Name="rdbtnBackSalmon"
                    Click="BackColorChanged">LightSalmon</RadioButton>
            </StackPanel>

            <TextBlock Style="{StaticResource titleText}">Foreground Color</TextBlock>
            <StackPanel Margin="10,10,10,10">
                <RadioButton Name="rdbtnOriginalForeColor"
                    IsChecked="True"
                    Click="ForeColorChanged">Original</RadioButton>
```

```

        Click="ForeColorChanged">Original</RadioButton>
<RadioButton Name="rdbtnForeRed"
            Click="ForeColorChanged">Red</RadioButton>
<RadioButton Name="rdbtnForeYellow"
            Click="ForeColorChanged">Yellow</RadioButton>
</StackPanel>

<TextBlock Style="{StaticResource titleText}">Font Family</TextBlock>
<StackPanel Margin="10,10,10,10">
    <RadioButton Name="rdbtnOriginalFamily"
                IsChecked="True"
                Click="FontChanged">Original</RadioButton>
    <RadioButton Name="rdbtnTimes"
                Click="FontChanged">Times New Roman</RadioButton>
    <RadioButton Name="rdbtnWingdings"
                Click="FontChanged">Wingdings</RadioButton>
</StackPanel>

<TextBlock Style="{StaticResource titleText}">Font Size</TextBlock>
<StackPanel Margin="10,10,10,10">
    <RadioButton Name="rdbtnOriginalSize"
                IsChecked="True"
                Click="FontSizeChanged">Original</RadioButton>
    <RadioButton Name="rdbtnTen"
                Click="FontSizeChanged">10</RadioButton>
    <RadioButton Name="rdbtnTwelve"
                Click="FontSizeChanged">12</RadioButton>
</StackPanel>

<TextBlock Style="{StaticResource titleText}">Font Style</TextBlock>
<StackPanel Margin="10,10,10,10">
    <RadioButton Name="rdbtnNormalStyle"
                IsChecked="True"
                Click="StyleChanged">Original</RadioButton>
    <RadioButton Name="rdbtnItalic"
                Click="StyleChanged">Italic</RadioButton>
</StackPanel>

<TextBlock Style="{StaticResource titleText}">Font Weight</TextBlock>
<StackPanel Margin="10,10,10,10">
    <RadioButton Name="rdbtnOriginalWeight"
                IsChecked="True"
                Click="WeightChanged">
        Original
    </RadioButton>
    <RadioButton Name="rdbtnBold"
                Click="WeightChanged">Bold</RadioButton>
</StackPanel>
</StackPanel>

<WindowsFormsHost Name="wfh"
                  DockPanel.Dock="Top"
                  Height="300">
    <mcl:MyControl1 Name="mc"/>
</WindowsFormsHost>

<StackPanel Orientation="Vertical"
            Height="Auto"
            Background="LightBlue">
    <TextBlock Margin="10,10,10,10"
              FontWeight="Bold"
              FontSize="12">Data From Control</TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        Name: <Span Name="txtName" Style="{StaticResource inlineText}"/>
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        Street Address: <Span Name="txtAddress" Style="{StaticResource inlineText}"/>
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">

```

```

        City: <Span Name="txtCity" Style="{StaticResource inlineText}"/>
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        State: <Span Name="txtState" Style="{StaticResource inlineText}"/>
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        Zip: <Span Name="txtZip" Style="{StaticResource inlineText}"/>
    </TextBlock>
</StackPanel>
</DockPanel>
</Window>

```

The first `StackPanel` element contains several sets of `RadioButton` controls that enable you to modify various default properties of the hosted control. That is followed by a `WindowsFormsHost` element, which hosts `MyControl1`. The final `StackPanel` element contains several `TextBlock` elements that display the data that is returned by the hosted control. The ordering of the elements and the `Dock` and `Height` attribute settings embed the hosted control into the window with no gaps or distortion.

### Hosting the Control

The following edited version of the previous XAML focuses on the elements that are needed to host `MyControl1`.

```

<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WpfHost.MainWindow"
    xmlns:mcl="clr-namespace:MyControls;assembly=MyControls"
    Loaded="Init">

```

```

<WindowsFormsHost Name="wfh"
    DockPanel.Dock="Top"
    Height="300">
    <mcl:MyControl1 Name="mc"/>
</WindowsFormsHost>

```

The `xmlns` namespace mapping attribute creates a reference to the `MyControls` namespace that contains the hosted control. This mapping enables you to represent `MyControl1` in XAML as `<mcl:MyControl1>`.

Two elements in the XAML handle the hosting:

- `WindowsFormsHost` represents the `WindowsFormsHost` element that enables you to host a Windows Forms control in a WPF application.
- `mcl:MyControl1`, which represents `MyControl1`, is added to the `WindowsFormsHost` element's child collection. As a result, this Windows Forms control is rendered as part of the WPF window, and you can communicate with the control from the application.

### Implementing the Code-Behind File

The code-behind file, `MainWindow.xaml.vb` or `MainWindow.xaml.cs`, contains the procedural code that implements the functionality of the UI discussed in the preceding section. The primary tasks are:

- Attaching an event handler to `MyControl1`'s `OnButtonClick` event.
- Modifying various properties of `MyControl1`, based on how the collection of option buttons are set.
- Displaying the data collected by the control.

### Initializing the Application

The initialization code is contained in an event handler for the window's `Loaded` event and attaches an event handler to the control's `OnButtonClick` event.

In MainWindow.xaml.vb or MainWindow.xaml.cs, add the following code to the `MainWindow` class.

```
private Application app;
private Window myWindow;
FontWeight initFontWeight;
Double initFontSize;
FontStyle initFontStyle;
SolidColorBrush initBackBrush;
SolidColorBrush initForeBrush;
FontFamily initFontFamily;
bool UIIsReady = false;

private void Init(object sender, EventArgs e)
{
    app = System.Windows.Application.Current;
    myWindow = (Window)app.MainWindow;
    myWindow.SizeToContent = SizeToContent.WidthAndHeight;
    wfh.TabIndex = 10;
    initFontSize = wfh.FontSize;
    initFontWeight = wfh.FontWeight;
    initFontFamily = wfh.FontFamily;
    initFontStyle = wfh.FontStyle;
    initBackBrush = (SolidColorBrush)wfh.Background;
    initForeBrush = (SolidColorBrush)wfh.Foreground;
    (wfh.Child as MyControl1).OnButtonClick += new MyControl1.MyControlEventHandler(Pane1_OnButtonClick);
    UIIsReady = true;
}
```

```
Private app As Application
Private myWindow As Window
Private initFontWeight As FontWeight
Private initFontSize As [Double]
Private initFontStyle As FontStyle
Private initBackBrush As SolidColorBrush
Private initForeBrush As SolidColorBrush
Private initFontFamily As FontFamily
Private UIIsReady As Boolean = False

Private Sub Init(ByVal sender As Object, ByVal e As RoutedEventArgs)
    app = System.Windows.Application.Current
    myWindow = CType(app.MainWindow, Window)
    myWindow.SizeToContent = SizeToContent.WidthAndHeight
    wfh.TabIndex = 10
    initFontSize = wfh.FontSize
    initFontWeight = wfh.FontWeight
    initFontFamily = wfh.FontFamily
    initFontStyle = wfh.FontStyle
    initBackBrush = CType(wfh.Background, SolidColorBrush)
    initForeBrush = CType(wfh.Foreground, SolidColorBrush)

    Dim mc As MyControl1 = wfh.Child

    AddHandler mc.OnButtonClick, AddressOf Pane1_OnButtonClick
    UIIsReady = True
End Sub
```

Because the XAML discussed previously added `MyControl1` to the `WindowsFormsHost` element's child element collection, you can cast the `WindowsFormsHost` element's `Child` to get the reference to `MyControl1`. You can then use that reference to attach an event handler to `OnButtonClick`.

In addition to providing a reference to the control itself, `WindowsFormsHost` exposes a number of the control's

properties, which you can manipulate from the application. The initialization code assigns those values to private global variables for later use in the application.

So that you can easily access the types in the `MyControls` DLL, add the following `Imports` or `using` statement to the top of the file.

```
Imports MyControls
```

```
using MyControls;
```

#### Handling the OnButtonClick Event

`MyControl1` raises the `OnButtonClick` event when the user clicks either of the control's buttons.

Add the following code to the `MainWindow` class.

```
//Handle button clicks on the Windows Form control
private void Panel1_OnButtonClick(object sender, MyControlEventArgs args)
{
    txtName.Inlines.Clear();
    txtAddress.Inlines.Clear();
    txtCity.Inlines.Clear();
    txtState.Inlines.Clear();
    txtZip.Inlines.Clear();

    if (args.IsOK)
    {
        txtName.Inlines.Add( " " + args.MyName );
        txtAddress.Inlines.Add( " " + args.MyStreetAddress );
        txtCity.Inlines.Add( " " + args.MyCity );
        txtState.Inlines.Add( " " + args.MyState );
        txtZip.Inlines.Add( " " + args.MyZip );
    }
}
```

```
'Handle button clicks on the Windows Form control
Private Sub Panel1_OnButtonClick(ByVal sender As Object, ByVal args As MyControlEventArgs)
    txtName.Inlines.Clear()
    txtAddress.Inlines.Clear()
    txtCity.Inlines.Clear()
    txtState.Inlines.Clear()
    txtZip.Inlines.Clear()

    If args.IsOK Then
        txtName.Inlines.Add(" " + args.MyName)
        txtAddress.Inlines.Add(" " + args.MyStreetAddress)
        txtCity.Inlines.Add(" " + args.MyCity)
        txtState.Inlines.Add(" " + args.MyState)
        txtZip.Inlines.Add(" " + args.MyZip)
    End If

End Sub
```

The data in the text boxes is packed into the `MyControlEventArgs` object. If the user clicks the **OK** button, the event handler extracts the data and displays it in the panel below `MyControl1`.

#### Modifying the Control's Properties

The `WindowsFormsHost` element exposes several of the hosted control's default properties. As a result, you can change the appearance of the control to match the style of your application more closely. The sets of option buttons in the left panel enable the user to modify several color and font properties. Each set of buttons has a

handler for the [Click](#) event, which detects the user's option button selections and changes the corresponding property on the control.

Add the following code to the `MainWindow` class.

```
private void BackColorChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnBackGreen)
        wfh.Background = new SolidColorBrush(Colors.LightGreen);
    else if (sender == rdbtnBackSalmon)
        wfh.Background = new SolidColorBrush(Colors.LightSalmon);
    else if (UIIsReady == true)
        wfh.Background = initBackBrush;
}

private void ForeColorChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnForeRed)
        wfh.Foreground = new SolidColorBrush(Colors.Red);
    else if (sender == rdbtnForeYellow)
        wfh.Foreground = new SolidColorBrush(Colors.Yellow);
    else if (UIIsReady == true)
        wfh.Foreground = initForegroundBrush;
}

private void FontChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnTimes)
        wfh.FontFamily = new FontFamily("Times New Roman");
    else if (sender == rdbtnWingdings)
        wfh.FontFamily = new FontFamily("Wingdings");
    else if (UIIsReady == true)
        wfh.FontFamily = initFontFamily;
}
private void FontSizeChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnTen)
        wfh.FontSize = 10;
    else if (sender == rdbtnTwelve)
        wfh.FontSize = 12;
    else if (UIIsReady == true)
        wfh.FontSize = initFontSize;
}
private void StyleChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnItalic)
        wfh.FontStyle = FontStyles.Italic;
    else if (UIIsReady == true)
        wfh.FontStyle = initFontStyle;
}
private void WeightChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnBold)
        wfh.FontWeight = FontWeights.Bold;
    else if (UIIsReady == true)
        wfh.FontWeight = initFontWeight;
}
```

```

Private Sub BackColorChanged(ByVal sender As Object, ByVal e As RoutedEventArgs)
    If sender.Equals(rdbtnBackGreen) Then
        wfh.Background = New SolidColorBrush(Colors.LightGreen)
    ElseIf sender.Equals(rdbtnBackSalmon) Then
        wfh.Background = New SolidColorBrush(Colors.LightSalmon)
    ElseIf UIIsReady = True Then
        wfh.Background = initBackBrush
    End If

End Sub

Private Sub ForeColorChanged(ByVal sender As Object, ByVal e As RoutedEventArgs)
    If sender.Equals(rdbtnForeRed) Then
        wfh.Foreground = New SolidColorBrush(Colors.Red)
    ElseIf sender.Equals(rdbtnForeYellow) Then
        wfh.Foreground = New SolidColorBrush(Colors.Yellow)
    ElseIf UIIsReady = True Then
        wfh.Foreground = initForeBrush
    End If

End Sub

Private Sub FontChanged(ByVal sender As Object, ByVal e As RoutedEventArgs)
    If sender.Equals(rdbtnTimes) Then
        wfh.FontFamily = New FontFamily("Times New Roman")
    ElseIf sender.Equals(rdbtnWingdings) Then
        wfh.FontFamily = New FontFamily("Wingdings")
    ElseIf UIIsReady = True Then
        wfh.FontFamily = initFontFamily
    End If

End Sub

Private Sub FontSizeChanged(ByVal sender As Object, ByVal e As RoutedEventArgs)
    If sender.Equals(rdbtnTen) Then
        wfh.FontSize = 10
    ElseIf sender.Equals(rdbtnTwelve) Then
        wfh.FontSize = 12
    ElseIf UIIsReady = True Then
        wfh.FontSize = initFontSize
    End If

End Sub

Private Sub StyleChanged(ByVal sender As Object, ByVal e As RoutedEventArgs)
    If sender.Equals(rdbtnItalic) Then
        wfh.FontStyle = FontStyles.Italic
    ElseIf UIIsReady = True Then
        wfh.FontStyle = initFontStyle
    End If

End Sub

Private Sub WeightChanged(ByVal sender As Object, ByVal e As RoutedEventArgs)
    If sender.Equals(rdbtnBold) Then
        wfh.FontWeight = FontWeights.Bold
    ElseIf UIIsReady = True Then
        wfh.FontWeight = initFontWeight
    End If

End Sub

```

Build and run the application. Add some text in the Windows Forms composite control and then click **OK**. The text appears in the labels. Click the different radio buttons to see the effect on the control.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Walkthrough: Hosting a Windows Forms Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)

# Walkthrough: Hosting an ActiveX Control in WPF

10/31/2019 • 3 minutes to read • [Edit Online](#)

To enable improved interaction with browsers, you can use Microsoft ActiveX controls in your WPF-based application. This walkthrough demonstrates how you can host the Microsoft Windows Media Player as a control on a WPF page.

Tasks illustrated in this walkthrough include:

- Creating the project.
- Creating the ActiveX control.
- Hosting the ActiveX control on a WPF Page.

When you have completed this walkthrough, you will understand how to use Microsoft ActiveX controls in your WPF-based application.

## Prerequisites

You need the following components to complete this walkthrough:

- Microsoft Windows Media Player installed on the computer where Visual Studio is installed.
- Visual Studio 2010.

## Creating the Project

### To create and set up the project

1. Create a WPF Application project named `HostingAxInWpf`.
2. Add a Windows Forms Control Library project to the solution, and name the project `WmpAxLib`.
3. In the `WmpAxLib` project, add a reference to the Windows Media Player assembly, which is named `wmp.dll`.
4. Open the **Toolbox**.
5. Right-click in the **Toolbox**, and then click **Choose Items**.
6. Click the **COM Components** tab, select the **Windows Media Player** control, and then click **OK**.

The Windows Media Player control is added to the **Toolbox**.

7. In Solution Explorer, right-click the **UserControl1** file, and then click **Rename**.
8. Change the name to `WmpAxControl.vb` or `WmpAxControl.cs`, depending on the language.
9. If you are prompted to rename all references, click **Yes**.

## Creating the ActiveX Control

Visual Studio automatically generates an `AxHost` wrapper class for a Microsoft ActiveX control when the control is added to a design surface. The following procedure creates a managed assembly named `AxInterop.WMPLib.dll`.

### To create the ActiveX control

1. Open `WmpAxControl.vb` or `WmpAxControl.cs` in the Windows Forms Designer.

2. From the **Toolbox**, add the Windows Media Player control to the design surface.
3. In the Properties window, set the value of the Windows Media Player control's **Dock** property to **Fill**.
4. Build the WmpAxLib control library project.

## Hosting the ActiveX Control on a WPF Page

### To host the ActiveX control

1. In the HostingAxInWpf project, add a reference to the generated ActiveX interoperability assembly.  
This assembly is named AxInterop.WMPLib.dll and was added to the Debug folder of the WmpAxLib project when you imported the Windows Media Player control.
2. Add a reference to the WindowsFormsIntegration assembly, which is named WindowsFormsIntegration.dll.
3. Add a reference to the Windows Forms assembly, which is named System.Windows.Forms.dll.
4. Open MainWindow.xaml in the WPF Designer.
5. Name the **Grid** element `grid1`.

```
<Grid Name="grid1">  
    </Grid>
```

6. In Design view or XAML view, select the **Window** element.
7. In the Properties window, click the **Events** tab.
8. Double-click the **Loaded** event.
9. Insert the following code to handle the **Loaded** event.

This code creates an instance of the **WindowsFormsHost** control and adds an instance of the **AxWindowsMediaPlayer** control as its child.

```
private void Window_Loaded(object sender, RoutedEventArgs e)  
{  
    // Create the interop host control.  
    System.Windows.Forms.Integration.WindowsFormsHost host =  
        new System.Windows.Forms.Integration.WindowsFormsHost();  
  
    // Create the ActiveX control.  
    WmpAxLib.AxWindowsMediaPlayer axWmp = new WmpAxLib.AxWindowsMediaPlayer();  
  
    // Assign the ActiveX control as the host control's child.  
    host.Child = axWmp;  
  
    // Add the interop host control to the Grid  
    // control's collection of child controls.  
    this.grid1.Children.Add(host);  
  
    // Play a .wav file with the ActiveX control.  
    axWmp.URL = @"C:\Windows\Media\tada.wav";  
}
```

```
Private Sub Window_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)

    ' Create the interop host control.
    Dim host As New System.Windows.Forms.Integration.WindowsFormsHost()

    ' Create the ActiveX control.
    Dim axWmp As New AxWMPLib.AxWindowsMediaPlayer()

    ' Assign the ActiveX control as the host control's child.
    host.Child = axWmp

    ' Add the interop host control to the Grid
    ' control's collection of child controls.
    Me.grid1.Children.Add(host)

    ' Play a .wav file with the ActiveX control.
    axWmp.URL = "C:\Windows\Media\tada.wav"

End Sub
```

10. Press F5 to build and run the application.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)

# How to: Enable Visual Styles in a Hybrid Application

12/10/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to enable visual styles on a Windows Forms control hosted in a WPF-based application.

If your application calls the [EnableVisualStyles](#) method, most of your Windows Forms controls will automatically use visual styles. For more information, see [Rendering Controls with Visual Styles](#).

For a complete code listing of the tasks illustrated in this topic, see [Enabling Visual Styles in a Hybrid Application Sample](#).

## Enabling Windows Forms Visual Styles

### To enable Windows Forms visual styles

1. Create a WPF Application project named `HostingWfWithVisualStyles`.
2. In Solution Explorer, add references to the following assemblies.
  - `WindowsFormsIntegration`
  - `System.Windows.Forms`
3. In the Toolbox, double-click the `Grid` icon to place a `Grid` element on the design surface.
4. In the Properties window, set the values of the `Height` and `Width` properties to `Auto`.
5. In Design view or XAML view, select the `Window`.
6. In the Properties window, click the **Events** tab.
7. Double-click the `Loaded` event.
8. In `MainWindow.xaml.vb` or `MainWindow.xaml.cs`, insert the following code to handle the `Loaded` event.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Comment out the following line to disable visual
    // styles for the hosted Windows Forms control.
    System.Windows.Forms.Application.EnableVisualStyles();

    // Create a WindowsFormsHost element to host
    // the Windows Forms control.
    System.Windows.Forms.Integration.WindowsFormsHost host =
        new System.Windows.Forms.Integration.WindowsFormsHost();

    // Create a Windows Forms tab control.
    System.Windows.Forms.TabControl tc = new System.Windows.Forms.TabControl();
    tc.TabPages.Add("Tab1");
    tc.TabPages.Add("Tab2");

    // Assign the Windows Forms tab control as the hosted control.
    host.Child = tc;

    // Assign the host element to the parent Grid element.
    this.grid1.Children.Add(host);
}
```

```

Private Sub Window_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Comment out the following line to disable visual
    ' styles for the hosted Windows Forms control.
    System.Windows.Forms.Application.EnableVisualStyles()

    ' Create a WindowsFormsHost element to host
    ' the Windows Forms control.
    Dim host As New System.Windows.Forms.Integration.WindowsFormsHost()

    ' Create a Windows Forms tab control.
    Dim tc As New System.Windows.Forms.TabControl()
    tc.TabPages.Add("Tab1")
    tc.TabPages.Add("Tab2")

    ' Assign the Windows Forms tab control as the hosted control.
    host.Child = tc

    ' Assign the host element to the parent Grid element.
    Me.grid1.Children.Add(host)

End Sub

```

9. Press F5 to build and run the application.

The Windows Forms control is painted with visual styles.

## Disabling Windows Forms Visual Styles

To disable visual styles, simply remove the call to the [EnableVisualStyles](#) method.

### **To disable Windows Forms visual styles**

1. Open MainWindow.xaml.vb or MainWindow.xaml.cs in the Code Editor.
2. Comment out the call to the [EnableVisualStyles](#) method.
3. Press F5 to build and run the application.

The Windows Forms control is painted with the default system style.

## See also

- [EnableVisualStyles](#)
- [System.Windows.Forms.VisualStyles](#)
- [WindowsFormsHost](#)
- [Rendering Controls with Visual Styles](#)
- [Walkthrough: Hosting a Windows Forms Control in WPF](#)

# Walkthrough: Arranging Windows Forms Controls in WPF

10/31/2019 • 9 minutes to read • [Edit Online](#)

This walkthrough shows you how to use WPF layout features to arrange Windows Forms controls in a hybrid application.

Tasks illustrated in this walkthrough include:

- Creating the project.
- Using default layout settings.
- Sizing to content.
- Using absolute positioning.
- Specifying size explicitly.
- Setting layout properties.
- Understanding z-order limitations.
- Docking.
- Setting visibility.
- Hosting a control that does not stretch.
- Scaling.
- Rotating.
- Setting padding and margins.
- Using dynamic layout containers.

For a complete code listing of the tasks illustrated in this walkthrough, see [Arranging Windows Forms Controls in WPF Sample](#).

When you are finished, you will have an understanding of Windows Forms layout features in WPF-based applications.

## Prerequisites

You need Visual Studio to complete this walkthrough.

## Creating the Project

To create and set up the project, follow these steps:

1. Create a WPF Application project named `WpfLayoutHostingWf`.
2. In Solution Explorer, add references to the following assemblies:
  - `WindowsFormsIntegration`
  - `System.Windows.Forms`
  - `System.Drawing`
3. Double-click `MainWindow.xaml` to open it in XAML view.
4. In the `Window` element, add the following Windows Forms namespace mapping.

```
xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

5. In the `Grid` element set the `ShowGridLines` property to `true` and define five rows and three columns.

```
<Grid ShowGridLines="true">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>

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

## Using Default Layout Settings

By default, the `WindowsFormsHost` element handles the layout for the hosted Windows Forms control.

To use default layout settings, follow these steps:

1. Copy the following XAML into the `Grid` element:

```
<!-- Default layout. -->
<Canvas Grid.Row="0" Grid.Column="0">
    <WindowsFormsHost Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

2. Press F5 to build and run the application. The Windows Forms `System.Windows.Forms.Button` control appears in the `Canvas`. The hosted control is sized based on its content, and the `WindowsFormsHost` element is sized to accommodate the hosted control.

## Sizing to Content

The `WindowsFormsHost` element ensures that the hosted control is sized to display its content properly.

To size to content, follow these steps:

1. Copy the following XAML into the `Grid` element:

```
<!-- Sizing to content. -->
<Canvas Grid.Row="1" Grid.Column="0">
    <WindowsFormsHost Background="Orange">
        <wf:Button Text="Windows Forms control with more content" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>

<Canvas Grid.Row="2" Grid.Column="0">
    <WindowsFormsHost FontSize="24" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

2. Press F5 to build and run the application. The two new button controls are sized to display the longer text string and larger font size properly, and the [WindowsFormsHost](#) elements are resized to accommodate the hosted controls.

## Using Absolute Positioning

You can use absolute positioning to place the [WindowsFormsHost](#) element anywhere in the user interface (UI).

To use absolute positioning, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Absolute positioning. -->
<Canvas Grid.Row="3" Grid.Column="0">
    <WindowsFormsHost Canvas.Top="20" Canvas.Left="20" Background="Yellow">
        <wf:Button Text="Windows Forms control with absolute positioning" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

2. Press F5 to build and run the application. The [WindowsFormsHost](#) element is placed 20 pixels from the top side of the grid cell and 20 pixels from the left.

## Specifying Size Explicitly

You can specify the size of the [WindowsFormsHost](#) element using the [Width](#) and [Height](#) properties.

To specify size explicitly, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Explicit sizing. -->
<Canvas Grid.Row="4" Grid.Column="0">
    <WindowsFormsHost Width="50" Height="70" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

2. Press F5 to build and run the application. The [WindowsFormsHost](#) element is set to a size of 50 pixels wide by 70 pixels high, which is smaller than the default layout settings. The content of the Windows Forms control is rearranged accordingly.

## Setting Layout Properties

Always set layout-related properties on the hosted control by using the properties of the [WindowsFormsHost](#) element. Setting layout properties directly on the hosted control will yield unintended results.

Setting layout-related properties on the hosted control in XAML has no effect.

To see the effects of setting properties on the hosted control, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```

<!-- Setting hosted control properties directly. -->
<Canvas Grid.Row="0" Grid.Column="1">
    <WindowsFormsHost Width="160" Height="50" Background="Yellow">
        <wf:Button Name="button1" Click="button1_Click" Text="Click me" FlatStyle="Flat" BackColor="Green"/>
    </WindowsFormsHost>
</Canvas>

```

2. In **Solution Explorer**, double-click *MainWindow.xaml.vb* or *MainWindow.xaml.cs* to open it in the Code Editor.
3. Copy the following code into the `MainWindow` class definition:

```

private void button1_Click(object sender, EventArgs e )
{
    System.Windows.Forms.Button b = sender as System.Windows.Forms.Button;

    b.Top = 20;
    b.Left = 20;
}

```

```

Private Sub button1_Click(ByVal sender As Object, ByVal e As EventArgs)
    Dim b As System.Windows.Forms.Button = sender

    b.Top = 20
    b.Left = 20

End Sub

```

4. Press F5 to build and run the application.
5. Click the **Click me** button. The `button1_Click` event handler sets the `Top` and `Left` properties on the hosted control. This causes the hosted control to be repositioned within the `WindowsFormsHost` element. The host maintains the same screen area, but the hosted control is clipped. Instead, the hosted control should always fill the `WindowsFormsHost` element.

## Understanding Z-Order Limitations

Visible `WindowsFormsHost` elements are always drawn on top of other WPF elements, and they are unaffected by z-order. To see this z-order behavior, do the following:

1. Copy the following XAML into the `Grid` element:

```

<!-- Z-order demonstration. -->
<Canvas Grid.Row="1" Grid.Column="1">
    <WindowsFormsHost Canvas.Top="20" Canvas.Left="20" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
    <Label Content="A WPF label" FontSize="24"/>
</Canvas>

```

2. Press F5 to build and run the application. The `WindowsFormsHost` element is painted over the label element.

## Docking

`WindowsFormsHost` element supports WPF docking. Set the `Dock` attached property to dock the hosted control in a `DockPanel` element.

To dock a hosted control, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Docking a WindowsFormsHost element. -->
<DockPanel LastChildFill="false" Grid.Row="2" Grid.Column="1">
    <WindowsFormsHost DockPanel.Dock="Right" Canvas.Top="20" Canvas.Left="20" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
</DockPanel>
```

2. Press F5 to build and run the application. The [WindowsFormsHost](#) element is docked to the right side of the [DockPanel](#) element.

## Setting Visibility

You can make your Windows Forms control invisible or collapse it by setting the [Visibility](#) property on the [WindowsFormsHost](#) element. When a control is invisible, it is not displayed, but it occupies layout space. When a control is collapsed, it is not displayed, nor does it occupy layout space.

To set the visibility of a hosted control, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Setting Visibility to hidden and collapsed. -->
<StackPanel Grid.Row="3" Grid.Column="1">
    <Button Name="button2" Click="button2_Click" Content="Click to make invisible" Background="OrangeRed"/>
    <WindowsFormsHost Name="host1" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
    <Button Name="button3" Click="button3_Click" Content="Click to collapse" Background="OrangeRed"/>
</StackPanel>
```

2. In *MainWindow.xaml.vb* or *MainWindow.xaml.cs*, copy the following code into the class definition:

```
private void button2_Click(object sender, EventArgs e)
{
    this.host1.Visibility = Visibility.Hidden;
}

private void button3_Click(object sender, EventArgs e)
{
    this.host1.Visibility = Visibility.Collapsed;
}
```

```
Private Sub button2_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Me.host1.Visibility = Windows.Visibility.Hidden
End Sub

Private Sub button3_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Me.host1.Visibility = Windows.Visibility.Collapsed
End Sub
```

3. Press F5 to build and run the application.
4. Click the **Click to make invisible** button to make the [WindowsFormsHost](#) element invisible.

5. Click the **Click to collapse** button to hide the [WindowsFormsHost](#) element from the layout entirely. When the Windows Forms control is collapsed, the surrounding elements are rearranged to occupy its space.

## Hosting a Control That Does Not Stretch

Some Windows Forms controls have a fixed size and do not stretch to fill available space in the layout. For example, the [MonthCalendar](#) control displays a month in a fixed space.

To host a control that does not stretch, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Hosting a control that does not stretch. -->
<!-- The MonthCalendar has a discrete size. -->
<StackPanel Grid.Row="4" Grid.Column="1">
    <Label Content="A WPF element" Background="OrangeRed"/>
    <WindowsFormsHost Background="Yellow">
        <wf:MonthCalendar/>
    </WindowsFormsHost>
    <Label Content="Another WPF element" Background="OrangeRed"/>
</StackPanel>
```

2. Press F5 to build and run the application. The [WindowsFormsHost](#) element is centered in the grid row, but it is not stretched to fill the available space. If the window is large enough, you may see two or more months displayed by the hosted [MonthCalendar](#) control, but these are centered in the row. The WPF layout engine centers elements that cannot be sized to fill the available space.

## Scaling

Unlike WPF elements, most Windows Forms controls are not continuously scalable. To provide custom scaling, you override the [WindowsFormsHost.ScaleChild](#) method.

To scale a hosted control by using the default behavior, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Scaling transformation. -->
<StackPanel Grid.Row="0" Grid.Column="2">

    <StackPanel.RenderTransform>
        <ScaleTransform CenterX="0" CenterY="0" ScaleX="0.5" ScaleY="0.5" />
    </StackPanel.RenderTransform>

    <Label Content="A WPF UIElement" Background="OrangeRed"/>

    <WindowsFormsHost Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>

    <Label Content="Another WPF UIElement" Background="OrangeRed"/>

</StackPanel>
```

2. Press F5 to build and run the application. The hosted control and its surrounding elements are scaled by a factor of 0.5. However, the hosted control's font is not scaled.

## Rotating

Unlike WPF elements, Windows Forms controls do not support rotation. The [WindowsFormsHost](#) element does

not rotate with other WPF elements when a rotation transformation is applied. Any rotation value other than 180 degrees raises the [LayoutError](#) event.

To see the effect of rotation in a hybrid application, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Rotation transformation. -->
<StackPanel Grid.Row="1" Grid.Column="2">

    <StackPanel.RenderTransform>
        <RotateTransform CenterX="200" CenterY="50" Angle="180" />
    </StackPanel.RenderTransform>

    <Label Content="A WPF element" Background="OrangeRed"/>

    <WindowsFormsHost Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>

    <Label Content="Another WPF element" Background="OrangeRed"/>

</StackPanel>
```

2. Press F5 to build and run the application. The hosted control is not rotated, but its surrounding elements are rotated by an angle of 180 degrees. You may have to resize the window to see the elements.

## Setting Padding and Margins

Padding and margins in WPF layout are similar to padding and margins in Windows Forms. Simply set the [Padding](#) and [Margin](#) properties on the [WindowsFormsHost](#) element.

To set padding and margins for a hosted control, follow these steps:

1. Copy the following XAML into the [Grid](#) element:

```
<!-- Padding. -->
<Canvas Grid.Row="2" Grid.Column="2">
    <WindowsFormsHost Padding="0, 20, 0, 0" Background="Yellow">
        <wf:Button Text="Windows Forms control with padding" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

```
<!-- Margin. -->
<Canvas Grid.Row="3" Grid.Column="2">
    <WindowsFormsHost Margin="20, 20, 0, 0" Background="Yellow">
        <wf:Button Text="Windows Forms control with margin" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

2. Press F5 to build and run the application. The padding and margin settings are applied to the hosted Windows Forms controls in the same way they would be applied in Windows Forms.

## Using Dynamic Layout Containers

Windows Forms provides two dynamic layout containers, [FlowLayoutPanel](#) and [TableLayoutPanel](#). You can also use these containers in WPF layouts.

To use a dynamic layout container, follow these steps:

1. Copy the following XAML into the **Grid** element:

```
<!-- Flow layout. -->
<DockPanel Grid.Row="4" Grid.Column="2">
    <WindowsFormsHost Name="flowLayoutPanelHost" Background="Yellow">
        <wf:FlowLayoutPanel/>
    </WindowsFormsHost>
</DockPanel>
```

2. In *MainWindow.xaml.vb* or *MainWindow.xaml.cs*, copy the following code into the class definition:

```
private void InitializeFlowLayoutPanel()
{
    System.Windows.Forms.FlowLayoutPanel flp =
        this.flowLayoutPanelHost.Child as System.Windows.Forms.FlowLayoutPanel;

    flp.WrapContents = true;

    const int numButtons = 6;

    for (int i = 0; i < numButtons; i++)
    {
        System.Windows.Forms.Button b = new System.Windows.Forms.Button();
        b.Text = "Button";
        b.BackColor = System.Drawing.Color.AliceBlue;
        b.FlatStyle = System.Windows.Forms.FlatStyle.Flat;

        flp.Controls.Add(b);
    }
}
```

```
Private Sub InitializeFlowLayoutPanel()
    Dim flp As System.Windows.Forms.FlowLayoutPanel = Me.flowLayoutPanelHost.Child

    flp.WrapContents = True

    Const numButtons As Integer = 6

    Dim i As Integer
    For i = 0 To numButtons
        Dim b As New System.Windows.Forms.Button()
        b.Text = "Button"
        b.BackColor = System.Drawing.Color.AliceBlue
        b.FlatStyle = System.Windows.Forms.FlatStyle.Flat

        flp.Controls.Add(b)
    Next i

End Sub
```

3. Add a call to the **InitializeFlowLayoutPanel** method in the constructor:

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

    this.InitializeFlowLayoutPanel();
}
```

```
Public Sub New()
    InitializeComponent()

    Me.InitializeFlowLayoutPanel()

End Sub
```

4. Press F5 to build and run the application. The [WindowsFormsHost](#) element fills the [DockPanel](#), and [FlowLayoutPanel](#) arranges its child controls in the default [FlowDirection](#).

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Layout Considerations for the WindowsFormsHost Element](#)
- [Arranging Windows Forms Controls in WPF Sample](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)

# Walkthrough: Binding to Data in Hybrid Applications

11/12/2019 • 6 minutes to read • [Edit Online](#)

Binding a data source to a control is essential for providing users with access to underlying data, whether you are using Windows Forms or WPF. This walkthrough shows how you can use data binding in hybrid applications that include both Windows Forms and WPF controls.

Tasks illustrated in this walkthrough include:

- Creating the project.
- Defining the data template.
- Specifying the form layout.
- Specifying data bindings.
- Displaying data by using interoperation.
- Adding the data source to the project.
- Binding to the data source.

For a complete code listing of the tasks illustrated in this walkthrough, see [Data Binding in Hybrid Applications Sample](#).

When you are finished, you will have an understanding of data binding features in hybrid applications.

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio.
- Access to the Northwind sample database running on Microsoft SQL Server.

## Creating the Project

### To create and set up the project

1. Create a WPF Application project named `WPFWithWFAndDatabinding`.
2. In Solution Explorer, add references to the following assemblies.
  - `WindowsFormsIntegration`
  - `System.Windows.Forms`
3. Open `MainWindow.xaml` in the WPF Designer.
4. In the `Window` element, add the following Windows Forms namespaces mapping.

```
xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

5. Name the default `Grid` element `mainGrid` by assigning the `Name` property.

```
<Grid x:Name="mainGrid">
```

## Defining the Data Template

The master list of customers is displayed in a [ListBox](#) control. The following code example defines a [DataTemplate](#) object named `ListItemsTemplate` that controls the visual tree of the [ListBox](#) control. This [DataTemplate](#) is assigned to the [ListBox](#) control's [ItemTemplate](#) property.

### To define the data template

- Copy the following XAML into the [Grid](#) element's declaration.

```
<Grid.Resources>
    <DataTemplate x:Key="ListItemsTemplate">
        <TextBlock Text="{Binding Path=ContactName}" />
    </DataTemplate>
</Grid.Resources>
```

## Specifying the Form Layout

The layout of the form is defined by a grid with three rows and three columns. [Label](#) controls are provided to identify each column in the Customers table.

### To set up the Grid layout

- Copy the following XAML into the [Grid](#) element's declaration.

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

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

### To set up the Label controls

- Copy the following XAML into the [Grid](#) element's declaration.

```
<StackPanel Orientation="Vertical" Grid.Row="0" Grid.Column="1">
    <Label Margin="20,38,5,2">First Name:</Label>
    <Label Margin="20,0,5,2">Company Name:</Label>
    <Label Margin="20,0,5,2">Phone:</Label>
    <Label Margin="20,0,5,2">Address:</Label>
    <Label Margin="20,0,5,2">City:</Label>
    <Label Margin="20,0,5,2">Region:</Label>
    <Label Margin="20,0,5,2">Postal Code:</Label>
</StackPanel>
```

## Specifying Data Bindings

The master list of customers is displayed in a [ListBox](#) control. The attached `ListItemsTemplate` binds a [TextBlock](#) control to the `ContactName` field from the database.

The details of each customer record are displayed in several [TextBox](#) controls.

### To specify data bindings

- Copy the following XAML into the [Grid](#) element's declaration.

The [Binding](#) class binds the [TextBox](#) controls to the appropriate fields in the database.

```
<StackPanel Orientation="Vertical" Grid.Row="0" Grid.Column="0">
    <Label Margin="20,5,5,0">List of Customers:</Label>
    <ListBox x:Name="listBox1" Height="200" Width="200" HorizontalAlignment="Left"
        ItemTemplate="{StaticResource ListItemsTemplate}" IsSynchronizedWithCurrentItem="True"
        Margin="20,5,5,5"/>
</StackPanel>

<StackPanel Orientation="Vertical" Grid.Row="0" Grid.Column="2">
    <TextBox Margin="5,38,5,2" Width="200" Text="{Binding Path=ContactName}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding Path=CompanyName}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding Path=Phone}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding Path=Address}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding Path=City}"/>
    <TextBox Margin="5,0,5,2" Width="30" HorizontalAlignment="Left" Text="{Binding Path=Region}"/>
    <TextBox Margin="5,0,5,2" Width="50" HorizontalAlignment="Left" Text="{Binding Path=PostalCode}"/>
</StackPanel>
```

## Displaying Data by Using Interoperation

The orders corresponding to the selected customer are displayed in a [System.Windows.Forms.DataGridView](#) control named `dataGridView1`. The `dataGridView1` control is bound to the data source in the code-behind file. A [WindowsFormsHost](#) control is the parent of this Windows Forms control.

### To display data in the DataGridView control

- Copy the following XAML into the [Grid](#) element's declaration.

```
<WindowsFormsHost Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" Margin="20,5,5,5" Height="300">
    <wf:DataGridView x:Name="dataGridView1"/>
</WindowsFormsHost>
```

## Adding the Data Source to the Project

With Visual Studio, you can easily add a data source to your project. This procedure adds a strongly typed data set to your project. Several other support classes, such as table adapters for each of the chosen tables, are also added.

### To add the data source

- From the **Data** menu, select **Add New Data Source**.
- In the **Data Source Configuration Wizard**, create a connection to the Northwind database by using a dataset. For more information, see [How to: Connect to Data in a Database](#).
- When you are prompted by the **Data Source Configuration Wizard**, save the connection string as `NorthwindConnectionString`.
- When you are prompted to choose your database objects, select the `Customers` and `Orders` tables, and name the generated data set `NorthwindDataSet`.

## Binding to the Data Source

The [System.Windows.Forms.BindingSource](#) component provides a uniform interface for the application's data

source. Binding to the data source is implemented in the code-behind file.

## To bind to the data source

1. Open the code-behind file, which is named MainWindow.xaml.vb or MainWindow.xaml.cs.
2. Copy the following code into the `MainWindow` class definition.

This code declares the `BindingSource` component and associated helper classes that connect to the database.

```
private System.Windows.Forms.BindingSource nwBindingSource;
private NorthwindDataSet nwDataSet;
private NorthwindDataSetTableAdapters.CustomersTableAdapter customersTableAdapter =
    new NorthwindDataSetTableAdapters.CustomersTableAdapter();
private NorthwindDataSetTableAdapters.OrdersTableAdapter ordersTableAdapter =
    new NorthwindDataSetTableAdapters.OrdersTableAdapter();
```

```
Private nwBindingSource As System.Windows.Forms.BindingSource
Private nwDataSet As NorthwindDataSet
Private customersTableAdapter As New NorthwindDataSetTableAdapters.CustomersTableAdapter()
Private ordersTableAdapter As New NorthwindDataSetTableAdapters.OrdersTableAdapter()
```

3. Copy the following code into the constructor.

This code creates and initializes the `BindingSource` component.

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

    // Create a DataSet for the Customers data.
    this.nwDataSet = new NorthwindDataSet();
    this.nwDataSet.DataSetName = "nwDataSet";

    // Create a BindingSource for the Customers data.
    this.nwBindingSource = new System.Windows.Forms.BindingSource();
    this.nwBindingSource.DataMember = "Customers";
    this.nwBindingSource.DataSource = this.nwDataSet;
}
```

```
Public Sub New()
    InitializeComponent()

    ' Create a DataSet for the Customers data.
    Me.nwDataSet = New NorthwindDataSet()
    Me.nwDataSet.DataSetName = "nwDataSet"

    ' Create a BindingSource for the Customers data.
    Me.nwBindingSource = New System.Windows.Forms.BindingSource()
    Me.nwBindingSource.DataMember = "Customers"
    Me.nwBindingSource.DataSource = Me.nwDataSet

End Sub
```

4. Open MainWindow.xaml.
5. In Design view or XAML view, select the `Window` element.
6. In the Properties window, click the **Events** tab.

7. Double-click the **Loaded** event.
8. Copy the following code into the **Loaded** event handler.

This code assigns the **BindingSource** component as the data context and populates the **Customers** and **Orders** adapter objects.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Fill the Customers table adapter with data.
    this.customersTableAdapter.ClearBeforeFill = true;
    this.customersTableAdapter.Fill(this.nwDataSet.Customers);

    // Fill the Orders table adapter with data.
    this.ordersTableAdapter.Fill(this.nwDataSet.Orders);

    // Assign the BindingSource to
    // the data context of the main grid.
    this.mainGrid.DataContext = this.nwBindingSource;

    // Assign the BindingSource to
    // the data source of the list box.
    this.listBox1.ItemsSource = this.nwBindingSource;

    // Because this is a master/details form, the DataGridView
    // requires the foreign key relating the tables.
    this.dataGridView1.DataSource = this.nwBindingSource;
    this.dataGridView1.DataMember = "FK_Orders_Customers";

    // Handle the currency management aspect of the data models.
    // Attach an event handler to detect when the current item
    // changes via the WPF ListBox. This event handler synchronizes
    // the list collection with the BindingSource.
    //

    BindingListCollectionView cv = CollectionViewSource.GetDefaultView(
        this.nwBindingSource) as BindingListCollectionView;

    cv.CurrentChanged += new EventHandler(WPF_CurrentChanged);
}
```

```

Private Sub Window_Loaded( _
    ByVal sender As Object, _
    ByVal e As RoutedEventArgs)

    ' Fill the Customers table adapter with data.
    Me.customersTableAdapter.ClearBeforeFill = True
    Me.customersTableAdapter.Fill(Me.nwDataSet.Customers)

    ' Fill the Orders table adapter with data.
    Me.ordersTableAdapter.Fill(Me.nwDataSet.Orders)

    ' Assign the BindingSource to
    ' the data context of the main grid.
    Me.mainGrid.DataContext = Me.nwBindingSource

    ' Assign the BindingSource to
    ' the data source of the list box.
    Me.listBox1.ItemsSource = Me.nwBindingSource

    ' Because this is a master/details form, the DataGridView
    ' requires the foreign key relating the tables.
    Me.dataGridView1.DataSource = Me.nwBindingSource
    Me.dataGridView1.DataMember = "FK_Orders_Customers"

    ' Handle the currency management aspect of the data models.
    ' Attach an event handler to detect when the current item
    ' changes via the WPF ListBox. This event handler synchronizes
    ' the list collection with the BindingSource.
    '

    Dim cv As BindingListCollectionView = _
        CollectionViewSource.GetDefaultView(Me.nwBindingSource)

    AddHandler cv.CurrentChanged, AddressOf WPF_CurrentChanged

End Sub

```

9. Copy the following code into the `MainWindow` class definition.

This method handles the `CurrentChanged` event and updates the current item of the data binding.

```

// This event handler updates the current item
// of the data binding.
void WPF_CurrentChanged(object sender, EventArgs e)
{
    BindingListCollectionView cv = sender as BindingListCollectionView;
    this.nwBindingSource.Position = cv.CurrentPosition;
}

```

```

' This event handler updates the current item
' of the data binding.
Private Sub WPF_CurrentChanged(ByVal sender As Object, ByVal e As EventArgs)
    Dim cv As BindingListCollectionView = sender
    Me.nwBindingSource.Position = cv.CurrentPosition
End Sub

```

10. Press F5 to build and run the application.

## See also

- [ElementHost](#)

- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Data Binding in Hybrid Applications Sample](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)

# Walkthrough: Hosting a 3-D WPF Composite Control in Windows Forms

11/12/2019 • 4 minutes to read • [Edit Online](#)

This walkthrough demonstrates how you can create a WPF composite control and host it in Windows Forms controls and forms by using the [ElementHost](#) control.

In this walkthrough, you will implement a WPF [UserControl](#) that contains two child controls. The [UserControl](#) displays a three-dimensional (3-D) cone. Rendering 3-D objects is much easier with the WPF than with Windows Forms. Therefore, it makes sense to host a WPF [UserControl](#) class to create 3-D graphics in Windows Forms.

Tasks illustrated in this walkthrough include:

- Creating the WPF [UserControl](#).
- Creating the Windows Forms host project.
- Hosting the WPF [UserControl](#).

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio 2017

## Create the UserControl

1. Create a **WPF User Control Library** project named `HostingWpfUserControlInWF`.
2. Open `UserControl1.xaml` in the WPF Designer.
3. Replace the generated code with the following code:

```
<UserControl x:Class="HostingWpfUserControlInWF.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <Grid>

        <!-- Place a Label control at the top of the view. -->
        <Label
            HorizontalAlignment="Center"
            TextBlock.TextAlignment="Center"
            FontSize="20"
            Foreground="Red"
            Content="Model: Cone"/>

        <!-- Viewport3D is the rendering surface. -->
        <Viewport3D Name="myViewport" >

            <!-- Add a camera. -->
            <Viewport3D.Camera>
                <PerspectiveCamera
                    FarPlaneDistance="20"
                    LookDirection="0,0,1"
                    UpDirection="0,1,0"
                    >
                        <!-- Set the camera's position. -->
                        <Vector3D X="0" Y="0" Z="100" />
                    </PerspectiveCamera>
                </Viewport3D.Camera>
            </Viewport3D>
        </Grid>
    </UserControl>
```

```

        NearPlaneDistance="1"
        Position="0,0,-3"
        FieldOfView="45" />
    </Viewport3D.Camera>

    <!-- Add models. -->
    <Viewport3D.Children>

        <ModelVisual3D>
            <ModelVisual3D.Content>

                <Model3DGroup>
                    <Model3DGroup.Children>

                        <!-- Lights, MeshGeometry3D and DiffuseMaterial objects are added to the
ModelVisual3D. -->
                        <DirectionalLight Color="#FFFFFF" Direction="3,-4,5" />

                        <!-- Define a red cone. -->
                        <GeometryModel3D>

                            <GeometryModel3D.Geometry>
                                <MeshGeometry3D
Positions="0.293893 -0.5 0.404509 0.475528 -0.5 0.154509 0 0.5 0 0.475528 -0.5 0.154509 0 0.5 0
0 0.5 0 0.475528 -0.5 0.154509 0.475528 -0.5 -0.154509 0 0.5 0 0.475528 -0.5 -0.154509 0 0.5 0
0 0.5 0 0.475528 -0.5 -0.154509 0.293893 -0.5 -0.404509 0 0.5 0 0.293893 -0.5 -0.404509 0 0.5 0
0 0.5 0 0.293893 -0.5 -0.404509 0 -0.5 -0.5 0 0.5 0 0 -0.5 -0.5 0 0.5 0 0 0.5 0 -0.5 -0.5 -
0.293893 -0.5 -0.404509 0 0.5 0 -0.293893 -0.5 -0.404509 0 0.5 0 0.050 0 0.5 0 -0.293893 -0.5 -0.404509
-0.475528 -0.5 -0.154509 0 0.5 0 -0.475528 -0.5 -0.154509 0 0.5 0 0 0.5 0 -0.475528 -0.5 -0.154509
-0.475528 -0.5 0.154509 0 0.5 0 -0.475528 -0.5 0.154509 0 0.5 0 0 0.5 0 -0.475528 -0.5 0.154509 -
0.293892 -0.5 0.404509 0 0.5 0 -0.293892 -0.5 0.404509 0 0.5 0 0 0.5 0 -0.293892 -0.5 0.404509 0 -
0.5 0.5 0 0.5 0 0 -0.5 0.5 0 0.5 0 0 -0.5 0.5 0 0.293893 -0.5 0.404509 0 0.5 0 0.293893 -
0.5 0.404509 0 0.5 0 0 0.5 0 "
Normals="0.7236065,0.4472139,0.5257313 0.2763934,0.4472138,0.8506507 0.5308242,0.4294462,0.7306172
0.2763934,0.4472138,0.8506507 0,0.4294458,0.9030925 0.5308242,0.4294462,0.7306172
0.2763934,0.4472138,0.8506507 -0.2763934,0.4472138,0.8506507 0,0.4294458,0.9030925 -
0.2763934,0.4472138,0.8506507 -0.5308242,0.4294462,0.7306172 0,0.4294458,0.9030925 -
0.2763934,0.4472138,0.8506507 -0.7236065,0.4472139,0.5257313 -0.5308242,0.4294462,0.7306172 -
0.7236065,0.4472139,0.5257313 -0.858892,0.429446,0.279071 -0.5308242,0.4294462,0.7306172 -
0.7236065,0.4472139,0.5257313 -0.8944269,0.4472139,0 -0.858892,0.429446,0.279071 -
0.8944269,0.4472139,0 -0.858892,0.429446,-0.279071 -0.858892,0.429446,0.279071 -0.8944269,0.4472139,0
-0.7236065,0.4472139,-0.5257313 -0.858892,0.429446,-0.279071 -0.7236065,0.4472139,-0.5257313 -
0.5308242,0.4294462,-0.7306172 -0.858892,0.429446,-0.279071 -0.7236065,0.4472139,-0.5257313 -
0.2763934,0.4472138,-0.8506507 -0.5308242,0.4294462,-0.7306172 -0.2763934,0.4472138,-0.8506507
0,0.4294458,-0.9030925 -0.5308242,0.4294462,-0.7306172 -0.2763934,0.4472138,-0.8506507
0.2763934,0.4472138,-0.8506507 0,0.4294458,-0.9030925 0.2763934,0.4472138,-0.8506507
0.5308242,0.4294459,-0.7306169 0,0.4294458,-0.9030925 0.2763934,0.4472138,-0.8506507
0.7236068,0.4472141,-0.5257306 0.5308249,0.4294459,-0.7306169 0.7236068,0.4472141,-0.5257306
0.8588922,0.4294461,-0.27907 0.5308249,0.4294459,-0.7306169 0.7236068,0.4472141,-0.5257306
0.8944269,0.4472139,0 0.8588922,0.4294461,-0.27907 0.8944269,0.4472139,0 0.858892,0.429446,0.279071
0.8588922,0.4294461,-0.27907 0.8944269,0.4472139,0 0.7236065,0.4472139,0.5257313
0.858892,0.429446,0.279071 0.7236065,0.4472139,0.5257313 0.5308242,0.4294462,0.7306172
0.858892,0.429446,0.279071 " TriangleIndices="0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 " />
                    </GeometryModel3D.Geometry>

                    <GeometryModel3D.Material>
                        <DiffuseMaterial>
                            <DiffuseMaterial.Brush>
                                <SolidColorBrush
Color="Red"
Opacity="1.0"/>
                            </DiffuseMaterial.Brush>
                        </DiffuseMaterial>
                    </GeometryModel3D.Material>
                </GeometryModel3D>
            </ModelVisual3D>
        </ModelVisual3D.Children>
    </ModelVisual3D>
</Viewport3D>

```

```
</Model3DGroup.Children>
</Model3DGroup>

</ModelVisual3D.Content>

</ModelVisual3D>

</Viewport3D.Children>

</Viewport3D>
</Grid>

</UserControl>
```

This code defines a [System.Windows.Controls.UserControl](#) that contains two child controls. The first child control is a [System.Windows.Controls.Label](#) control; the second is a [Viewport3D](#) control that displays a 3-D cone.

## Create the host project

1. Add a **Windows Forms App (.NET Framework)** project named `WpfUserControlHost` to the solution.
2. In **Solution Explorer**, add a reference to the WindowsFormsIntegration assembly, which is named WindowsFormsIntegration.dll.
3. Add references to the following WPF assemblies:
  - PresentationCore
  - PresentationFramework
  - WindowsBase
4. Add a reference to the `HostingWpfUserControlInWF` project.
5. In Solution Explorer, set the `WpfUserControlHost` project to be the startup project.

## Host the UserControl

1. In the Windows Forms Designer, open Form1.
2. In the Properties window, click **Events**, and then double-click the [Load](#) event to create an event handler.  
The Code Editor opens to the newly generated `Form1_Load` event handler.
3. Replace the code in Form1.cs with the following code.

The `Form1_Load` event handler creates an instance of `UserControl1` and adds it to the [ElementHost](#) control's collection of child controls. The [ElementHost](#) control is added to the form's collection of child controls.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using System.Windows.Forms.Integration;

namespace WpfUserControlHost
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Create the ElementHost control for hosting the
            // WPF UserControl.
            ElementHost host = new ElementHost();
            host.Dock = DockStyle.Fill;

            // Create the WPF UserControl.
            HostingWpfUserControlInWf.UserControl1 uc =
                new HostingWpfUserControlInWf.UserControl1();

            // Assign the WPF UserControl to the ElementHost control's
            // Child property.
            host.Child = uc;

            // Add the ElementHost control to the form's
            // collection of child controls.
            this.Controls.Add(host);
        }
    }
}
```

```

Imports System.Collections.Generic
Imports System.ComponentModel
Imports System.Data
Imports System.Drawing
Imports System.Text
Imports System.Windows.Forms

Imports System.Windows.Forms.Integration

Public Class Form1
    Inherits Form

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Create the ElementHost control for hosting the
        ' WPF UserControl.
        Dim host As New ElementHost()
        host.Dock = DockStyle.Fill

        ' Create the WPF UserControl.
        Dim uc As New HostingWpfUserControlInWf.UserControl1()

        ' Assign the WPF UserControl to the ElementHost control's
        ' Child property.
        host.Child = uc

        ' Add the ElementHost control to the form's
        ' collection of child controls.
        Me.Controls.Add(host)
    End Sub

End Class

```

4. Press **F5** to build and run the application.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Hosting a WPF Composite Control in Windows Forms Sample](#)

# Walkthrough: Hosting a WPF Composite Control in Windows Forms

11/3/2019 • 16 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a rich environment for creating applications. However, when you have a substantial investment in Windows Forms code, it can be more effective to extend your existing Windows Forms application with WPF rather than to rewrite it from scratch. A common scenario is when you want to embed one or more controls implemented with WPF within your Windows Forms application. For more information about customizing WPF controls, see [Control Customization](#).

This walkthrough steps you through an application that hosts a WPF composite control to perform data-entry in a Windows Forms application. The composite control is packaged in a DLL. This general procedure can be extended to more complex applications and controls. This walkthrough is designed to be nearly identical in appearance and functionality to [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#). The primary difference is that the hosting scenario is reversed.

The walkthrough is divided into two sections. The first section briefly describes the implementation of the WPF composite control. The second section discusses in detail how to host the composite control in a Windows Forms application, receive events from the control, and access some of the control's properties.

Tasks illustrated in this walkthrough include:

- Implementing the WPF composite control.
- Implementing the Windows Forms host application.

For a complete code listing of the tasks illustrated in this walkthrough, see [Hosting a WPF Composite Control in Windows Forms Sample](#).

## Prerequisites

You need Visual Studio to complete this walkthrough.

## Implementing the WPF Composite Control

The WPF composite control used in this example is a simple data-entry form that takes the user's name and address. When the user clicks one of two buttons to indicate that the task is finished, the control raises a custom event to return that information to the host. The following illustration shows the rendered control.

The following image shows a WPF composite control:



## Creating the Project

To start the project:

1. Launch Visual Studio, and open the **New Project** dialog box.
2. In Visual C# and the Windows category, select the **WPF User Control Library** template.
3. Name the new project `MyControls`.
4. For the location, specify a conveniently named top-level folder, such as `WindowsFormsHostingWpfControl`.  
Later, you will put the host application in this folder.
5. Click **OK** to create the project. The default project contains a single control named `UserControl1`.
6. In Solution Explorer, rename `UserControl1` to `MyControl1`.

Your project should have references to the following system DLLs. If any of these DLLs are not included by default, add them to your project.

- PresentationCore
- PresentationFramework
- System
- WindowsBase

### Creating the User Interface

The user interface (UI) for the composite control is implemented with Extensible Application Markup Language (XAML). The composite control UI consists of five `TextBox` elements. Each `TextBox` element has an associated `TextBlock` element that serves as a label. There are two `Button` elements at the bottom, **OK** and **Cancel**. When the user clicks either button, the control raises a custom event to return the information to the host.

#### Basic Layout

The various UI elements are contained in a `Grid` element. You can use `Grid` to arrange the contents of the composite control in much the same way you would use a `Table` element in HTML. WPF also has a `Table` element, but `Grid` is more lightweight and better suited for simple layout tasks.

The following XAML shows the basic layout. This XAML defines the overall structure of the control by specifying the number of columns and rows in the `Grid` element.

In `MyControl1.xaml`, replace the existing XAML with the following XAML.

```
<Grid xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="MyControls.MyControl1"
      Background="#DCDCDC"
      Width="375"
      Height="250"
      Name="rootElement"
      Loaded="Init">

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

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

```
</Grid>
```

#### Adding TextBlock and TextBox Elements to the Grid

You place a UI element in the grid by setting the element's [RowProperty](#) and [ColumnProperty](#) attributes to the appropriate row and column number. Remember that row and column numbering are zero-based. You can have an element span multiple columns by setting its [ColumnSpanProperty](#) attribute. For more information about [Grid](#) elements, see [Create a Grid Element](#).

The following XAML shows the composite control's [TextBox](#) and [TextBlock](#) elements with their [RowProperty](#) and [ColumnProperty](#) attributes, which are set to place the elements properly in the grid.

In MyControl1.xaml, add the following XAML within the [Grid](#) element.

```

<TextBlock Grid.Column="0"
    Grid.Row="0"
    Grid.ColumnSpan="4"
    Margin="10,5,10,0"
    HorizontalAlignment="Center"
    Style="{StaticResource titleText}">Simple WPF Control</TextBlock>

<TextBlock Grid.Column="0"
    Grid.Row="1"
    Style="{StaticResource inlineText}"
    Name="nameLabel">Name</TextBlock>
<TextBox Grid.Column="1"
    Grid.Row="1"
    Grid.ColumnSpan="3"
    Name="txtName"/>

<TextBlock Grid.Column="0"
    Grid.Row="2"
    Style="{StaticResource inlineText}"
    Name="addressLabel">Street Address</TextBlock>
<TextBox Grid.Column="1"
    Grid.Row="2"
    Grid.ColumnSpan="3"
    Name="txtAddress"/>

<TextBlock Grid.Column="0"
    Grid.Row="3"
    Style="{StaticResource inlineText}"
    Name="cityLabel">City</TextBlock>
<TextBox Grid.Column="1"
    Grid.Row="3"
    Width="100"
    Name="txtCity"/>

<TextBlock Grid.Column="2"
    Grid.Row="3"
    Style="{StaticResource inlineText}"
    Name="stateLabel">State</TextBlock>
<TextBox Grid.Column="3"
    Grid.Row="3"
    Width="50"
    Name="txtState"/>

<TextBlock Grid.Column="0"
    Grid.Row="4"
    Style="{StaticResource inlineText}"
    Name="zipLabel">Zip</TextBlock>
<TextBox Grid.Column="1"
    Grid.Row="4"
    Width="100"
    Name="txtZip"/>

```

### Styling the UI Elements

Many of the elements on the data-entry form have a similar appearance, which means that they have identical settings for several of their properties. Rather than setting each element's attributes separately, the previous XAML uses [Style](#) elements to define standard property settings for classes of elements. This approach reduces the complexity of the control and enables you to change the appearance of multiple elements through a single style attribute.

The [Style](#) elements are contained in the [Grid](#) element's [Resources](#) property, so they can be used by all elements in the control. If a style is named, you apply it to an element by adding a [Style](#) element set to the style's name. Styles that are not named become the default style for the element. For more information about WPF styles, see [Styling and Templating](#).

The following XAML shows the `Style` elements for the composite control. To see how the styles are applied to elements, see the previous XAML. For example, the last `TextBlock` element has the `inlineText` style, and the last `TextBox` element uses the default style.

In `MyControl1.xaml`, add the following XAML just after the `Grid` start element.

```
<Grid.Resources>
    <Style x:Key="inlineText" TargetType="{x:Type TextBlock}">
        <Setter Property="Margin" Value="10,5,10,0"/>
        <Setter Property="FontWeight" Value="Normal"/>
        <Setter Property="FontSize" Value="12"/>
    </Style>
    <Style x:Key="titleText" TargetType="{x:Type TextBlock}">
        <Setter Property="DockPanel.Dock" Value="Top"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="FontSize" Value="14"/>
        <Setter Property="Margin" Value="10,5,10,0"/>
    </Style>
    <Style TargetType="{x:Type Button}">
        <Setter Property="Margin" Value="10,5,10,0"/>
        <Setter Property="Width" Value="60"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="Margin" Value="10,5,10,0"/>
    </Style>
</Grid.Resources>
```

#### Adding the OK and Cancel Buttons

The final elements on the composite control are the **OK** and **CancelButton** elements, which occupy the first two columns of the last row of the `Grid`. These elements use a common event handler, `ButtonClicked`, and the default `Button` style defined in the previous XAML.

In `MyControl1.xaml`, add the following XAML after the last `TextBox` element. The XAML part of the composite control is now complete.

```
<Button Grid.Row="5"
        Grid.Column="0"
        Name="btnOK"
        Click="ButtonClicked">OK</Button>
<Button Grid.Row="5"
        Grid.Column="1"
        Name="btnCancel"
        Click="ButtonClicked">Cancel</Button>
```

#### Implementing the Code-Behind File

The code-behind file, `MyControl1.xaml.cs`, implements three essential tasks:

- Handles the event that occurs when the user clicks one of the buttons.
- Retrieves the data from the `TextBox` elements, and packages it in a custom event argument object.
- Raises the custom `OnButtonClick` event, which notifies the host that the user is finished and passes the data back to the host.

The control also exposes a number of color and font properties that enable you to change the appearance. Unlike the `WindowsFormsHost` class, which is used to host a Windows Forms control, the `ElementHost` class exposes the control's `Background` property only. To maintain the similarity between this code example and the example discussed in [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#), the control exposes the remaining properties directly.

### The Basic Structure of the Code-Behind File

The code-behind file consists of a single namespace, `MyControls`, which will contain two classes, `MyControl1` and `MyControlEventArgs`.

```
namespace MyControls
{
    public partial class MyControl1 : Grid
    {
        //...
    }
    public class MyControlEventArgs : EventArgs
    {
        //...
    }
}
```

The first class, `MyControl1`, is a partial class containing the code that implements the functionality of the UI defined in `MyControl1.xaml`. When `MyControl1.xaml` is parsed, the XAML is converted to the same partial class, and the two partial classes are merged to form the compiled control. For this reason, the class name in the code-behind file must match the class name assigned to `MyControl1.xaml`, and it must inherit from the root element of the control. The second class, `MyControlEventArgs`, is an event arguments class that is used to send the data back to the host.

Open `MyControl1.xaml.cs`. Change the existing class declaration so that it has the following name and inherits from `Grid`.

```
public partial class MyControl1 : Grid
```

### Initializing the Control

The following code implements several basic tasks:

- Declares a private event, `OnButtonClick`, and its associated delegate, `MyControlEventHandler`.
- Creates several private global variables that store the user's data. This data is exposed through corresponding properties.
- Implements a handler, `Init`, for the control's `Loaded` event. This handler initializes the global variables by assigning them the values defined in `MyControl1.xaml`. To do this, it uses the `Name` assigned to a typical `TextBlock` element, `nameLabel`, to access that element's property settings.

Delete the existing constructor and add the following code to your `MyControl1` class.

```

public delegate void MyControlEventHandler(object sender, MyControlEventArgs args);
public event MyControlEventHandler OnButtonClick;
private FontWeight _fontWeight;
private double _fontSize;
private FontFamily _fontFamily;
private FontStyle _fontStyle;
private SolidColorBrush _foreground;
private SolidColorBrush _background;

private void Init(object sender, EventArgs e)
{
    //They all have the same style, so use nameLabel to set initial values.
    _fontWeight = nameLabel.FontWeight;
    _fontSize = nameLabel.FontSize;
    _fontFamily = nameLabel.FontFamily;
    _fontStyle = nameLabel.FontStyle;
    _foreground = (SolidColorBrush)nameLabel.Foreground;
    _background = (SolidColorBrush)rootElement.Background;
}

```

#### Handling the Buttons' Click Events

The user indicates that the data-entry task is finished by clicking either the **OK** button or the **Cancel** button. Both buttons use the same **Click** event handler, `ButtonClicked`. Both buttons have a name, `btnOK` or `btnCancel`, that enables the handler to determine which button was clicked by examining the value of the `sender` argument. The handler does the following:

- Creates a `MyControlEventArgs` object that contains the data from the `TextBox` elements.
- If the user clicked the **Cancel** button, sets the `MyControlEventArgs` object's `IsOK` property to `false`.
- Raises the `OnButtonClick` event to indicate to the host that the user is finished, and passes back the collected data.

Add the following code to your `MyControl1` class, after the `Init` method.

```

private void ButtonClicked(object sender, RoutedEventArgs e)
{
    MyControlEventArgs retvals = new MyControlEventArgs(true,
        txtName.Text,
        txtAddress.Text,
        txtCity.Text,
        txtState.Text,
        txtZip.Text);

    if (sender == btnCancel)
    {
        retvals.IsOK = false;
    }
    if (OnButtonClick != null)
        OnButtonClick(this, retvals);
}

```

#### Creating Properties

The remainder of the class simply exposes properties that correspond to the global variables discussed previously. When a property changes, the set accessor modifies the appearance of the control by changing the corresponding element properties and updating the underlying global variables.

Add the following code to your `MyControl1` class.

```

public FontWeight MyControl_FontWeight
{
    get { return _fontWeight; }
}

```

```

        set
    {
        _fontWeight = value;
        nameLabel.FontWeight = value;
        addressLabel.FontWeight = value;
        cityLabel.FontWeight = value;
        stateLabel.FontWeight = value;
        zipLabel.FontWeight = value;
    }
}

public double MyControl_FontSize
{
    get { return _fontSize; }
    set
    {
        _fontSize = value;
        nameLabel.FontSize = value;
        addressLabel.FontSize = value;
        cityLabel.FontSize = value;
        stateLabel.FontSize = value;
        zipLabel.FontSize = value;
    }
}

public FontStyle MyControl_FontStyle
{
    get { return _fontStyle; }
    set
    {
        _fontStyle = value;
        nameLabel.FontStyle = value;
        addressLabel.FontStyle = value;
        cityLabel.FontStyle = value;
        stateLabel.FontStyle = value;
        zipLabel.FontStyle = value;
    }
}

public FontFamily MyControl_FontFamily
{
    get { return _fontFamily; }
    set
    {
        _fontFamily = value;
        nameLabel.FontFamily = value;
        addressLabel.FontFamily = value;
        cityLabel.FontFamily = value;
        stateLabel.FontFamily = value;
        zipLabel.FontFamily = value;
    }
}

public SolidColorBrush MyControl_Background
{
    get { return _background; }
    set
    {
        _background = value;
        rootElement.Background = value;
    }
}

public SolidColorBrush MyControl_Foreground
{
    get { return _foreground; }
    set
    {
        _foreground = value;
        nameLabel.Foreground = value;
        addressLabel.Foreground = value;
        cityLabel.Foreground = value;
        stateLabel.Foreground = value;
    }
}

```

```
        zipLabel.Foreground = value;
    }
}
```

#### Sending the Data Back to the Host

The final component in the file is the `MyControlEventArgs` class, which is used to send the collected data back to the host.

Add the following code to your `MyControls` namespace. The implementation is straightforward, and is not discussed further.

```

public class MyControlEventArgs : EventArgs
{
    private string _Name;
    private string _StreetAddress;
    private string _City;
    private string _State;
    private string _Zip;
    private bool _IsOK;

    public MyControlEventArgs(bool result,
                             string name,
                             string address,
                             string city,
                             string state,
                             string zip)
    {
        _IsOK = result;
        _Name = name;
        _StreetAddress = address;
        _City = city;
        _State = state;
        _Zip = zip;
    }

    public string MyName
    {
        get { return _Name; }
        set { _Name = value; }
    }
    public string MyStreetAddress
    {
        get { return _StreetAddress; }
        set { _StreetAddress = value; }
    }
    public string MyCity
    {
        get { return _City; }
        set { _City = value; }
    }
    public string MyState
    {
        get { return _State; }
        set { _State = value; }
    }
    public string MyZip
    {
        get { return _Zip; }
        set { _Zip = value; }
    }
    public bool IsOK
    {
        get { return _IsOK; }
        set { _IsOK = value; }
    }
}

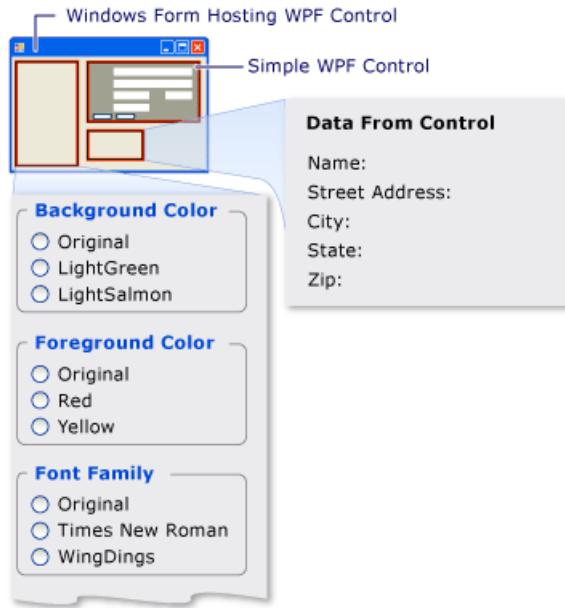
```

Build the solution. The build will produce a DLL named MyControls.dll.

## Implementing the Windows Forms Host Application

The Windows Forms host application uses an [ElementHost](#) object to host the WPF composite control. The application handles the `OnButtonClick` event to receive the data from the composite control. The application also has a set of option buttons that you can use to modify the control's appearance. The following illustration shows the application.

The following image shows a WPF composite control hosted in a Windows Forms application:



## Creating the Project

To start the project:

1. Launch Visual Studio, and open the **New Project** dialog box.
2. In Visual C# and the Windows category, select the **Windows Forms Application** template.
3. Name the new project `WFHost`.
4. For the location, specify the same top-level folder that contains the MyControls project.
5. Click **OK** to create the project.

You also need to add references to the DLL that contains `MyControl1` and other assemblies.

1. Right-click the project name in Solution Explorer, and select **Add Reference**.
2. Click the **Browse** tab, and browse to the folder that contains MyControls.dll. For this walkthrough, this folder is MyControls\bin\Debug.
3. Select MyControls.dll, and then click **OK**.
4. Add references to the following assemblies.
  - PresentationCore
  - PresentationFramework
  - System.Xaml
  - WindowsBase
  - WindowsFormsIntegration

## Implementing the User Interface for the Application

The UI for the Windows Form application contains several controls to interact with the WPF composite control.

1. Open Form1 in the Windows Form Designer.
2. Enlarge the form to accommodate the controls.
3. In the upper-right corner of the form, add a `System.Windows.Forms.Panel` control to hold the WPF

composite control.

4. Add the following `System.Windows.Forms.GroupBox` controls to the form.

NAME	TEXT
groupBox1	Background Color
groupBox2	Foreground Color
groupBox3	Font Size
groupBox4	Font Family
groupBox5	Font Style
groupBox6	Font Weight
groupBox7	Data from control

5. Add the following `System.Windows.Forms.RadioButton` controls to the `System.Windows.Forms.GroupBox` controls.

GROUPBOX	NAME	TEXT
groupBox1	radioBackgroundOriginal	Original
groupBox1	radioBackgroundLightGreen	LightGreen
groupBox1	radioBackgroundLightSalmon	LightSalmon
groupBox2	radioForegroundOriginal	Original
groupBox2	radioForegroundRed	Red
groupBox2	radioForegroundYellow	Yellow
groupBox3	radioSizeOriginal	Original
groupBox3	radioSizeTen	10
groupBox3	radioSizeTwelve	12
groupBox4	radioFamilyOriginal	Original
groupBox4	radioFamilyTimes	Times New Roman
groupBox4	radioFamilyWingDings	WingDings
groupBox5	radioStyleOriginal	Normal
groupBox5	radioStyleItalic	Italic

GROUPBOX	NAME	TEXT
groupBox6	radioWeightOriginal	Original
groupBox6	radioWeightBold	Bold

6. Add the following [System.Windows.Forms.Label](#) controls to the last [System.Windows.Forms.GroupBox](#). These controls display the data returned by the WPF composite control.

GROUPBOX	NAME	TEXT
groupBox7	lblName	Name:
groupBox7	lblAddress	Street Address:
groupBox7	lblCity	City:
groupBox7	lblState	State:
groupBox7	lblZip	Zip:

### Initializing the Form

You generally implement the hosting code in the form's [Load](#) event handler. The following code shows the [Load](#) event handler, a handler for the WPF composite control's [Loaded](#) event, and declarations for several global variables that are used later.

In the Windows Forms Designer, double-click the form to create a [Load](#) event handler. At the top of Form1.cs, add the following `using` statements.

```
using System.Windows;
using System.Windows.Forms.Integration;
using System.Windows.Media;
```

Replace the contents of the existing `Form1` class with the following code.

```

private ElementHost ctrlHost;
private MyControls.MyControl1 wpfAddressCtrl;
System.Windows.FontWeight initFontWeight;
double initFontSize;
System.Windows.FontStyle initFontStyle;
System.Windows.Media.SolidColorBrush initBackBrush;
System.Windows.Media.SolidColorBrush initForeBrush;
System.Windows.Media.FontFamily initFontFamily;

public Form1()
{
    InitializeComponent();
}

private void Form1_Load(object sender, EventArgs e)
{
    ctrlHost = new ElementHost();
    ctrlHost.Dock = DockStyle.Fill;
    panel1.Controls.Add(ctrlHost);
    wpfAddressCtrl = new MyControls.MyControl1();
    wpfAddressCtrl.InitializeComponent();
    ctrlHost.Child = wpfAddressCtrl;

    wpfAddressCtrl.OnButtonClick +=
        new MyControls.MyControl1.MyControlEventHandler(
            avAddressCtrl_OnButtonClick);
    wpfAddressCtrl.Loaded += new RoutedEventHandler(
        avAddressCtrl_Loaded);
}

void avAddressCtrl_Loaded(object sender, EventArgs e)
{
    initBackBrush = (SolidColorBrush)wpfAddressCtrl.MyControl_Background;
    initForeBrush = wpfAddressCtrl.MyControl_Foreground;
    initFontFamily = wpfAddressCtrl.MyControl_FontFamily;
    initFontSize = wpfAddressCtrl.MyControl_FontSize;
    initFontWeight = wpfAddressCtrl.MyControl_FontWeight;
    initFontStyle = wpfAddressCtrl.MyControl_FontStyle;
}

```

The `Form1_Load` method in the preceding code shows the general procedure for hosting a WPF control:

1. Create a new `ElementHost` object.
2. Set the control's `Dock` property to `DockStyle.Fill`.
3. Add the `ElementHost` control to the `Panel` control's `Controls` collection.
4. Create an instance of the WPF control.
5. Host the composite control on the form by assigning the control to the `ElementHost` control's `Child` property.

The remaining two lines in the `Form1_Load` method attach handlers to two control events:

- `OnButtonClick` is a custom event that is fired by the composite control when the user clicks the **OK** or **Cancel** button. You handle the event to get the user's response and to collect any data that the user specified.
- `Loaded` is a standard event that is raised by a WPF control when it is fully loaded. The event is used here because the example needs to initialize several global variables using properties from the control. At the time of the form's `Load` event, the control is not fully loaded and those values are still set to `null`. You need to wait until the control's `Loaded` event occurs before you can access those properties.

The **Loaded** event handler is shown in the preceding code. The `onButtonClick` handler is discussed in the next section.

## Handling OnButtonClick

The `OnButtonClick` event occurs when the user clicks the **OK** or **Cancel** button.

The event handler checks the event argument's `IsOK` field to determine which button was clicked. The `lbl` *data* variables correspond to the `Label` controls that were discussed earlier. If the user clicks the **OK** button, the data from the control's `TextBox` controls is assigned to the corresponding `Label` control. If the user clicks **Cancel**, the `Text` values are set to the default strings.

Add the following button click event handler code to the `Form1` class.

```
void avAddressCtrl_OnButtonClick(
    object sender,
    MyControls.MyControl1.MyControlEventArgs args)
{
    if (args.IsOK)
    {
        lblAddress.Text = "Street Address: " + args.MyStreetAddress;
        lblCity.Text = "City: " + args.MyCity;
        lblName.Text = "Name: " + args.MyName;
        lblState.Text = "State: " + args.MyState;
        lblZip.Text = "Zip: " + args.MyZip;
    }
    else
    {
        lblAddress.Text = "Street Address: ";
        lblCity.Text = "City: ";
        lblName.Text = "Name: ";
        lblState.Text = "State: ";
        lblZip.Text = "Zip: ";
    }
}
```

Build and run the application. Add some text in the WPF composite control and then click **OK**. The text appears in the labels. At this point, code has not been added to handle the radio buttons.

## Modifying the Appearance of the Control

The `RadioButton` controls on the form will enable the user to change the WPF composite control's foreground and background colors as well as several font properties. The background color is exposed by the `ElementHost` object. The remaining properties are exposed as custom properties of the control.

Double-click each `RadioButton` control on the form to create `CheckedChanged` event handlers. Replace the `CheckedChanged` event handlers with the following code.

```
private void radioBackgroundOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = initBackBrush;
}

private void radioBackgroundLightGreen_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = new SolidColorBrush(Colors.LightGreen);
}

private void radioBackgroundLightSalmon_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = new SolidColorBrush(Colors.LightSalmon);
}

private void radioForegroundOriginal_CheckedChanged(object sender, EventArgs e)
```

```

{
    wpfAddressCtrl.MyControl_Foreground = initForeBrush;
}

private void radioForegroundRed_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = new System.Windows.Media.SolidColorBrush(Colors.Red);
}

private void radioForegroundYellow_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = new System.Windows.Media.SolidColorBrush(Colors.Yellow);
}

private void radioFamilyOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = initFontFamily;
}

private void radioFamilyTimes_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = new System.Windows.Media.FontFamily("Times New Roman");
}

private void radioFamilyWingDings_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = new System.Windows.Media.FontFamily("WingDings");
}

private void radioSizeOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = initFontSize;
}

private void radioSizeTen_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = 10;
}

private void radioSizeTwelve_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = 12;
}

private void radioStyleOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontStyle = initFontStyle;
}

private void radioStyleItalic_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontStyle = System.Windows.FontStyles.Italic;
}

private void radioWeightOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontWeight = initFontWeight;
}

private void radioWeightBold_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontWeight = FontWeights.Bold;
}

```

Build and run the application. Click the different radio buttons to see the effect on the WPF composite control.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Design XAML in Visual Studio](#)
- [Walkthrough: Hosting a Windows Forms Composite Control in WPF](#)
- [Walkthrough: Hosting a 3-D WPF Composite Control in Windows Forms](#)

# Walkthrough: Mapping Properties Using the ElementHost Control

10/31/2019 • 7 minutes to read • [Edit Online](#)

This walkthrough shows you how to use the [PropertyMap](#) property to map Windows Forms properties to corresponding properties on a hosted WPF element.

Tasks illustrated in this walkthrough include:

- Creating the project.
- Defining a new property mapping.
- Removing a default property mapping.
- Extending a default property mapping.

For a complete code listing of the tasks illustrated in this walkthrough, see [Mapping Properties Using the ElementHost Control Sample](#).

When you are finished, you will be able to map Windows Forms properties to corresponding WPF properties on a hosted element.

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio 2017

## Creating the Project

### To create the project

1. Create a **Windows Forms App** project named `PropertyMappingWithElementHost`.

2. In **Solution Explorer**, add references to the following WPF assemblies.

- PresentationCore
- PresentationFramework
- WindowsBase
- WindowsFormsIntegration

3. Copy the following code into the top of the `Form1` code file.

```
using System.Windows;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Forms.Integration;
```

```
Imports System.Windows
Imports System.Windows.Media
Imports System.Windows.Media.Imaging
Imports System.Windows.Forms.Integration
```

4. Open `Form1` in the Windows Forms Designer. Double-click the form to add an event handler for the `Load` event.
5. Return to the Windows Forms Designer and add an event handler for the form's `Resize` event. For more information, see [How to: Create Event Handlers Using the Designer](#).
6. Declare an `ElementHost` field in the `Form1` class.

```
ElementHost elemHost = null;
```

```
Private elemHost As ElementHost = Nothing
```

## Defining New Property Mappings

The `ElementHost` control provides several default property mappings. You add a new property mapping by calling the `Add` method on the `ElementHost` control's `PropertyMap`.

### To define new property mappings

1. Copy the following code into the definition for the `Form1` class.

```
// The AddMarginMapping method adds a new property mapping
// for the Margin property.
private void AddMarginMapping()
{
    elemHost.PropertyMap.Add(
        "Margin",
        new PropertyTranslator(OnMarginChange));
}

// The OnMarginChange method implements the mapping
// from the Windows Forms Margin property to the
// Windows Presentation Foundation Margin property.
//
// The provided Padding value is used to construct
// a Thickness value for the hosted element's Margin
// property.
private void OnMarginChange(object h, String propertyName, object value)
{
    ElementHost host = h as ElementHost;
    Padding p = (Padding)value;
    System.Windows.Controls.Button wpfButton =
        host.Child as System.Windows.Controls.Button;

    Thickness t = new Thickness(p.Left, p.Top, p.Right, p.Bottom );

    wpfButton.Margin = t;
}
```

```

' The AddMarginMapping method adds a new property mapping
' for the Margin property.
Private Sub AddMarginMapping()

    elemHost.PropertyMap.Add( _
        "Margin", _
        New PropertyTranslator(AddressOf OnMarginChange))

End Sub

' The OnMarginChange method implements the mapping
' from the Windows Forms Margin property to the
' Windows Presentation Foundation Margin property.
'
' The provided Padding value is used to construct
' a Thickness value for the hosted element's Margin
' property.
Private Sub OnMarginChange( _
    ByVal h As Object, _
    ByVal propertyName As String, _
    ByVal value As Object)

    Dim host As ElementHost = h
    Dim p As Padding = CType(value, Padding)
    Dim wpfButton As System.Windows.Controls.Button = host.Child

    Dim t As New Thickness(p.Left, p.Top, p.Right, p.Bottom)

    wpfButton.Margin = t

End Sub

```

The `AddMarginMapping` method adds a new mapping for the [Margin](#) property.

The `OnMarginChange` method translates the [Margin](#) property to the WPF [Margin](#) property.

2. Copy the following code into the definition for the `Form1` class.

```
// The AddRegionMapping method assigns a custom
// mapping for the Region property.
private void AddRegionMapping()
{
    elemHost.PropertyMap.Add(
        "Region",
        new PropertyTranslator(OnRegionChange));
}

// The OnRegionChange method assigns an EllipseGeometry to
// the hosted element's Clip property.
private void OnRegionChange(
    object h,
    String propertyName,
    object value)
{
    ElementHost host = h as ElementHost;
    System.Windows.Controls.Button wpfButton =
        host.Child as System.Windows.Controls.Button;

    wpfButton.Clip = new EllipseGeometry(new Rect(
        0,
        0,
        wpfButton.ActualWidth,
        wpfButton.ActualHeight));
}

// The Form1_Resize method handles the form's Resize event.
// It calls the OnRegionChange method explicitly to
// assign a new clipping geometry to the hosted element.
private void Form1_Resize(object sender, EventArgs e)
{
    this.OnRegionChange(elemHost, "Region", null);
}
```

```

' The AddRegionMapping method assigns a custom
' mapping for the Region property.
Private Sub AddRegionMapping()

    elemHost.PropertyMap.Add( _
        "Region", _
        New PropertyTranslator(AddressOf OnRegionChange))

End Sub

' The OnRegionChange method assigns an EllipseGeometry to
' the hosted element's Clip property.
Private Sub OnRegionChange( _
    ByVal h As Object, _
    ByVal propertyName As String, _
    ByVal value As Object)

    Dim host As ElementHost = h

    Dim wpfButton As System.Windows.Controls.Button = host.Child

    wpfButton.Clip = New EllipseGeometry(New Rect( _
        0, _
        0, _
        wpfButton.ActualWidth, _
        wpfButton.ActualHeight))

End Sub

' The Form1_Resize method handles the form's Resize event.
' It calls the OnRegionChange method explicitly to
' assign a new clipping geometry to the hosted element.
Private Sub Form1_Resize( _
    ByVal sender As Object, _
    ByVal e As EventArgs) Handles MyBase.Resize

    If elemHost IsNot Nothing Then
        Me.OnRegionChange(elemHost, "Region", Nothing)
    End If

End Sub

```

The `AddRegionMapping` method adds a new mapping for the `Region` property.

The `OnRegionChange` method translates the `Region` property to the WPF `Clip` property.

The `Form1_Resize` method handles the form's `Resize` event and sizes the clipping region to fit the hosted element.

## Removing a Default Property Mapping

Remove a default property mapping by calling the `Remove` method on the `ElementHost` control's `PropertyMap`.

### To remove a default property mapping

- Copy the following code into the definition for the `Form1` class.

```
// The RemoveCursorMapping method deletes the default
// mapping for the Cursor property.
private void RemoveCursorMapping()
{
    elemHost.PropertyMap.Remove("Cursor");
}
```

```
' The RemoveCursorMapping method deletes the default
' mapping for the Cursor property.
Private Sub RemoveCursorMapping()
    elemHost.PropertyMap.Remove("Cursor")
End Sub
```

The `RemoveCursorMapping` method deletes the default mapping for the `Cursor` property.

## Extending a Default Property Mapping

You can use a default property mapping and also extend it with your own mapping.

### To extend a default property mapping

- Copy the following code into the definition for the `Form1` class.

```
// The ExtendBackColorMapping method adds a property
// translator if a mapping already exists.
private void ExtendBackColorMapping()
{
    if (elemHost.PropertyMap["BackColor"] != null)
    {
        elemHost.PropertyMap["BackColor"] +=
            new PropertyTranslator(OnBackColorChange);
    }
}

// The OnBackColorChange method assigns a specific image
// to the hosted element's Background property.
private void OnBackColorChange(object h, String propertyName, object value)
{
    ElementHost host = h as ElementHost;
    System.Windows.Controls.Button wpfButton =
        host.Child as System.Windows.Controls.Button;

    ImageBrush b = new ImageBrush(new BitmapImage(
        new Uri(@"file:///C:\WINDOWS\Santa Fe Stucco.bmp")));
    wpfButton.Background = b;
}
```

```

' The ExtendBackColorMapping method adds a property
' translator if a mapping already exists.
Private Sub ExtendBackColorMapping()

    If elemHost.PropertyMap("BackColor") IsNot Nothing Then

        elemHost.PropertyMap("BackColor") = PropertyTranslator.Combine( _
            elemHost.PropertyMap("BackColor"), _
            PropertyTranslator.CreateDelegate( _
                GetType(PropertyTranslator), _
                Me, _
                "OnBackColorChange"))
    End If

End Sub

' The OnBackColorChange method assigns a specific image
' to the hosted element's Background property.
Private Sub OnBackColorChange( _
    ByVal h As Object, _
    ByVal propertyName As String, _
    ByVal value As Object)

    Dim host As ElementHost = h
    Dim wpfButton As System.Windows.Controls.Button = host.Child
    Dim b As New ImageBrush(New BitmapImage( _
        New Uri("file:///C:\WINDOWS\Santa Fe Stucco.bmp")))
    wpfButton.Background = b

End Sub

```

The `ExtendBackColorMapping` method adds a custom property translator to the existing `BackColor` property mapping.

The `OnBackColorChange` method assigns a specific image to the hosted control's `Background` property. The `OnBackColorChange` method is called after the default property mapping is applied.

## Initialize your property mappings

1. Copy the following code into the definition for the `Form1` class.

```
private void Form1_Load(object sender, EventArgs e)
{
    // Create the ElementHost control.
    elemHost = new ElementHost();
    elemHost.Dock = DockStyle.Fill;
    this.Controls.Add(elemHost);

    // Create a Windows Presentation Foundation Button element
    // and assign it as the ElementHost control's child.
    System.Windows.Controls.Button wpfButton = new System.Windows.Controls.Button();
    wpfButton.Content = "Windows Presentation Foundation Button";
    elemHost.Child = wpfButton;

    // Map the Margin property.
    this.AddMarginMapping();

    // Remove the mapping for the Cursor property.
    this.RemoveCursorMapping();

    // Add a mapping for the Region property.
    this.AddRegionMapping();

    // Add another mapping for the BackColor property.
    this.ExtendBackColorMapping();

    // Cause the OnMarginChange delegate to be called.
    elemHost.Margin = new Padding(23, 23, 23, 23);

    // Cause the OnRegionChange delegate to be called.
    elemHost.Region = new Region();

    // Cause the OnBackColorChange delegate to be called.
    elemHost.BackColor = System.Drawing.Color.AliceBlue;
}
```

```

Private Sub Form1_Load( _
    ByVal sender As Object, _
    ByVal e As EventArgs) Handles MyBase.Load

    ' Create the ElementHost control.
    elemHost = New ElementHost()
    elemHost.Dock = DockStyle.Fill
    Me.Controls.Add(elemHost)

    ' Create a Windows Presentation Foundation Button element
    ' and assign it as the ElementHost control's child.
    Dim wpfButton As New System.Windows.Controls.Button()
    wpfButton.Content = "Windows Presentation Foundation Button"
    elemHost.Child = wpfButton

    ' Map the Margin property.
    Me.AddMarginMapping()

    ' Remove the mapping for the Cursor property.
    Me.RemoveCursorMapping()

    ' Add a mapping for the Region property.
    Me.AddRegionMapping()

    ' Add another mapping for the BackColor property.
    Me.ExtendBackColorMapping()

    ' Cause the OnMarginChange delegate to be called.
    elemHost.Margin = New Padding(23, 23, 23, 23)

    ' Cause the OnRegionChange delegate to be called.
    elemHost.Region = New [Region]()

    ' Cause the OnBackColorChange delegate to be called.
    elemHost.BackColor = System.Drawing.Color.AliceBlue

End Sub

```

The `Form1_Load` method handles the `Load` event and performs the following initialization.

- Creates a WPF `Button` element.
  - Calls the methods you defined earlier in the walkthrough to set up the property mappings.
  - Assigns initial values to the mapped properties.
2. Press F5 to build and run the application.

## See also

- [ElementHost.PropertyMap](#)
- [WindowsFormsHost.PropertyMap](#)
- [WindowsFormsHost](#)
- [Windows Forms and WPF Property Mapping](#)
- [Design XAML in Visual Studio](#)
- [Walkthrough: Hosting a WPF Composite Control in Windows Forms](#)

# Walkthrough: Mapping Properties Using the WindowsFormsHost Element

11/12/2019 • 9 minutes to read • [Edit Online](#)

This walkthrough shows you how to use the [PropertyMap](#) property to map WPF properties to corresponding properties on a hosted Windows Forms control.

Tasks illustrated in this walkthrough include:

- Creating the project.
- Defining the application layout.
- Defining a new property mapping.
- Removing a default property mapping.
- Replacing a default property mapping.
- Extending a default property mapping.

For a complete code listing of the tasks illustrated in this walkthrough, see [Mapping Properties Using the WindowsFormsHost Element Sample](#).

When you are finished, you will be able to map WPF properties to corresponding properties on a hosted Windows Forms control.

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio 2017

## Create and set up the project

1. Create a **WPF App** project named `PropertyMappingWithWfhSample`.
2. In **Solution Explorer**, add a reference to the WindowsFormsIntegration assembly, which is named WindowsFormsIntegration.dll.
3. In **Solution Explorer**, add references to the System.Drawing and System.Windows.Forms assemblies.

## Defining the Application Layout

The WPF-based application uses the [WindowsFormsHost](#) element to host a Windows Forms control.

### To define the application layout

1. Open Window1.xaml in the WPF Designer.
2. Replace the existing code with the following code.

```
<Window x:Class="PropertyMappingWithWfh.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="PropertyMappingWithWfh" Height="300" Width="300"
    Loaded="WindowLoaded">
    <DockPanel Name="panel1" LastChildFill="True">
        <WindowsFormsHost Name="wfHost" DockPanel.Dock="Left" SizeChanged="Window1_SizeChanged"
            FontSize="20" />
    </DockPanel>
</Window>
```

3. Open Window1.xaml.cs in the Code Editor.
4. At the top of the file, import the following namespaces.

```
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
using System.Windows.Forms.Integration;
```

```
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Windows.Forms
Imports System.Windows.Forms.Integration
```

## Defining a New Property Mapping

The [WindowsFormsHost](#) element provides several default property mappings. You add a new property mapping by calling the [Add](#) method on the [WindowsFormsHost](#) element's [PropertyMap](#).

### To define a new property mapping

- Copy the following code into the definition for the `Window1` class.

```
// The AddClipMapping method adds a custom
// mapping for the Clip property.
private void AddClipMapping()
{
    wfHost.PropertyMap.Add(
        "Clip",
        new PropertyTranslator(OnClipChange));
}

// The OnClipChange method assigns an elliptical clipping
// region to the hosted control's Region property.
private void OnClipChange(object h, String propertyName, object value)
{
    WindowsFormsHost host = h as WindowsFormsHost;
    System.Windows.Forms.CheckBox cb = host.Child as System.Windows.Forms.CheckBox;

    if (cb != null)
    {
        cb.Region = this.CreateClipRegion();
    }
}

// The Window1_SizeChanged method handles the window's
// SizeChanged event. It calls the OnClipChange method explicitly
// to assign a new clipping region to the hosted control.
private void Window1_SizeChanged(object sender, SizeChangedEventArgs e)
{
    this.OnClipChange(wfHost, "Clip", null);
}

// The CreateClipRegion method creates a Region from an
// elliptical GraphicsPath.
private Region CreateClipRegion()
{
    GraphicsPath path = new GraphicsPath();

    path.StartFigure();

    path.AddEllipse(new System.Drawing.Rectangle(
        0,
        0,
        (int)wfHost.ActualWidth,
        (int)wfHost.ActualHeight ) );

    path.CloseFigure();

    return( new Region(path) );
}
```

```

' The AddClipMapping method adds a custom mapping
' for the Clip property.
Private Sub AddClipMapping()

    wfHost.PropertyMap.Add( _
        "Clip", _
        New PropertyTranslator(AddressOf OnClipChange))

End Sub

' The OnClipChange method assigns an elliptical clipping
' region to the hosted control's Region property.
Private Sub OnClipChange( _
    ByVal h As Object, _
    ByVal propertyName As String, _
    ByVal value As Object)

    Dim host As WindowsFormsHost = h

    Dim cb As System.Windows.Forms.CheckBox = host.Child

    If cb IsNot Nothing Then
        cb.Region = Me.CreateClipRegion()
    End If

End Sub

' The Window1_SizeChanged method handles the window's
' SizeChanged event. It calls the OnClipChange method explicitly
' to assign a new clipping region to the hosted control.
Private Sub Window1_SizeChanged( _
    ByVal sender As Object, _
    ByVal e As SizeChangedEventArgs)

    Me.OnClipChange(wfHost, "Clip", Nothing)

End Sub

' The CreateClipRegion method creates a Region from an
' elliptical GraphicsPath.
Private Function CreateClipRegion() As [Region]
    Dim path As New GraphicsPath()

    path.StartFigure()

    path.AddEllipse(New System.Drawing.Rectangle( _
        0, _
        0, _
        wfHost.ActualWidth, _
        wfHost.ActualHeight))

    path.CloseFigure()

    Return New [Region](path)
End Function

```

The `AddClipMapping` method adds a new mapping for the `Clip` property.

The `OnClipChange` method translates the `Clip` property to the Windows Forms `Region` property.

The `Window1_SizeChanged` method handles the window's `SizeChanged` event and sizes the clipping region to fit the application window.

## Removing a Default Property Mapping

Remove a default property mapping by calling the [Remove](#) method on the [WindowsFormsHost](#) element's [PropertyMap](#).

### To remove a default property mapping

- Copy the following code into the definition for the `Window1` class.

```
// The RemoveCursorMapping method deletes the default  
// mapping for the Cursor property.  
private void RemoveCursorMapping()  
{  
    wfHost.PropertyMap.Remove("Cursor");  
}
```

```
' The RemoveCursorMapping method deletes the default  
' mapping for the Cursor property.  
Private Sub RemoveCursorMapping()  
    wfHost.PropertyMap.Remove("Cursor")  
End Sub
```

The `RemoveCursorMapping` method deletes the default mapping for the [Cursor](#) property.

## Replacing a Default Property Mapping

Replace a default property mapping by removing the default mapping and calling the [Add](#) method on the [WindowsFormsHost](#) element's [PropertyMap](#).

### To replace a default property mapping

- Copy the following code into the definition for the `Window1` class.

```
// The ReplaceFlowDirectionMapping method replaces the
// default mapping for the FlowDirection property.
private void ReplaceFlowDirectionMapping()
{
    wfHost.PropertyMap.Remove("FlowDirection");

    wfHost.PropertyMap.Add(
        "FlowDirection",
        new PropertyTranslator(OnFlowDirectionChange));
}

// The OnFlowDirectionChange method translates a
// Windows Presentation Foundation FlowDirection value
// to a Windows Forms RightToLeft value and assigns
// the result to the hosted control's RightToLeft property.
private void OnFlowDirectionChange(object h, String propertyName, object value)
{
    WindowsFormsHost host = h as WindowsFormsHost;
    System.Windows.FlowDirection fd = (System.Windows.FlowDirection)value;
    System.Windows.Forms.CheckBox cb = host.Child as System.Windows.Forms.CheckBox;

    cb.RightToLeft = (fd == System.Windows.FlowDirection.RightToLeft) ?
        RightToLeft.Yes : RightToLeft.No;
}

// The cb_CheckedChanged method handles the hosted control's
// CheckedChanged event. If the Checked property is true,
// the flow direction is set to RightToLeft, otherwise it is
// set to LeftToRight.
private void cb_CheckedChanged(object sender, EventArgs e)
{
    System.Windows.Forms.CheckBox cb = sender as System.Windows.Forms.CheckBox;

    wfHost.FlowDirection = ( cb.CheckState == CheckState.Checked ) ?
        System.Windows.FlowDirection.RightToLeft :
        System.Windows.FlowDirection.LeftToRight;
}
```

```

' The ReplaceFlowDirectionMapping method replaces the
' default mapping for the FlowDirection property.
Private Sub ReplaceFlowDirectionMapping()

    wfHost.PropertyMap.Remove("FlowDirection")

    wfHost.PropertyMap.Add( _
        "FlowDirection", _
        New PropertyTranslator(AddressOf OnFlowDirectionChange))
End Sub

' The OnFlowDirectionChange method translates a
' Windows Presentation Foundation FlowDirection value
' to a Windows Forms RightToLeft value and assigns
' the result to the hosted control's RightToLeft property.
Private Sub OnFlowDirectionChange( _
    ByVal h As Object, _
    ByVal propertyName As String, _
    ByVal value As Object)

    Dim host As WindowsFormsHost = h

    Dim fd As System.Windows.FlowDirection = _
        CType(value, System.Windows.FlowDirection)

    Dim cb As System.Windows.Forms.CheckBox = host.Child

    cb.RightToLeft = IIf(fd = System.Windows.FlowDirection.RightToLeft, _
        RightToLeft.Yes, _
        RightToLeft.No)

End Sub

' The cb_CheckedChanged method handles the hosted control's
' CheckedChanged event. If the Checked property is true,
' the flow direction is set to RightToLeft, otherwise it is
' set to LeftToRight.
Private Sub cb_CheckedChanged( _
    ByVal sender As Object, _
    ByVal e As EventArgs)

    Dim cb As System.Windows.Forms.CheckBox = sender

    wfHost.FlowDirection = IIf(cb.CheckState = CheckState.Checked, _
        System.Windows.FlowDirection.RightToLeft, _
        System.Windows.FlowDirection.LeftToRight)

End Sub

```

The `ReplaceFlowDirectionMapping` method replaces the default mapping for the `FlowDirection` property.

The `OnFlowDirectionChange` method translates the `FlowDirection` property to the Windows Forms `RightToLeft` property.

The `cb_CheckedChanged` method handles the `CheckedChanged` event on the `CheckBox` control. It assigns the `FlowDirection` property based on the value of the `CheckState` property.

## Extending a Default Property Mapping

You can use a default property mapping and also extend it with your own mapping.

### To extend a default property mapping

- Copy the following code into the definition for the `Window1` class.

```
// The ExtendBackgroundMapping method adds a property
// translator if a mapping already exists.
private void ExtendBackgroundMapping()
{
    if (wfHost.PropertyMap["Background"] != null)
    {
        wfHost.PropertyMap["Background"] += new PropertyTranslator(OnBackgroundChange);
    }
}

// The OnBackgroundChange method assigns a specific image
// to the hosted control's BackgroundImage property.
private void OnBackgroundChange(object h, String propertyName, object value)
{
    WindowsFormsHost host = h as WindowsFormsHost;
    System.Windows.Forms.CheckBox cb = host.Child as System.Windows.Forms.CheckBox;
    ImageBrush b = value as ImageBrush;

    if (b != null)
    {
        cb.BackgroundImage = new System.Drawing.Bitmap(@"C:\WINDOWS\Santa Fe Stucco.bmp");
    }
}
```

```
' The ExtendBackgroundMapping method adds a property
' translator if a mapping already exists.
Private Sub ExtendBackgroundMapping()
    If wfHost.PropertyMap("Background") IsNot Nothing Then

        wfHost.PropertyMap("Background") = PropertyTranslator.Combine( _
            wfHost.PropertyMap("Background"), _
            PropertyTranslator.CreateDelegate( _
                GetType(PropertyTranslator), _
                Me, _
                "OnBackgroundChange"))
    End If

End Sub

' The OnBackgroundChange method assigns a specific image
' to the hosted control's BackgroundImage property.
Private Sub OnBackgroundChange(ByVal h As Object, ByVal propertyName As String, ByVal value As Object)
    Dim host As WindowsFormsHost = h
    Dim cb As System.Windows.Forms.CheckBox = host.Child
    Dim b As ImageBrush = value

    If Not (b Is Nothing) Then
        cb.BackgroundImage = New System.Drawing.Bitmap("C:\WINDOWS\Santa Fe Stucco.bmp")
    End If

End Sub
```

The `ExtendBackgroundMapping` method adds a custom property translator to the existing `Background` property mapping.

The `OnBackgroundChange` method assigns a specific image to the hosted control's `BackgroundImage` property. The `OnBackgroundChange` method is called after the default property mapping is applied.

## Initializing Your Property Mappings

Set up your property mappings by calling the previously described methods in the [Loaded](#) event handler.

## To initialize your property mappings

1. Copy the following code into the definition for the `Window1` class.

```
// The WindowLoaded method handles the Loaded event.  
// It enables Windows Forms visual styles, creates  
// a Windows Forms checkbox control, and assigns the  
// control as the child of the WindowsFormsHost element.  
// This method also modifies property mappings on the  
// WindowsFormsHost element.  
private void WindowLoaded(object sender, RoutedEventArgs e)  
{  
    System.Windows.Forms.Application.EnableVisualStyles();  
  
    // Create a Windows Forms checkbox control and assign  
    // it as the WindowsFormsHost element's child.  
    System.Windows.Forms.CheckBox cb = new System.Windows.Forms.CheckBox();  
    cb.Text = "Windows Forms checkbox";  
    cb.Dock = DockStyle.Fill;  
    cb.TextAlign = ContentAlignment.MiddleCenter;  
    cb.CheckedChanged += new EventHandler(cb_CheckedChanged);  
    wfHost.Child = cb;  
  
    // Replace the default mapping for the FlowDirection property.  
    this.ReplaceFlowDirectionMapping();  
  
    // Remove the mapping for the Cursor property.  
    this.RemoveCursorMapping();  
  
    // Add the mapping for the Clip property.  
    this.AddClipMapping();  
  
    // Add another mapping for the Background property.  
    this.ExtendBackgroundMapping();  
  
    // Cause the OnFlowDirectionChange delegate to be called.  
    wfHost.FlowDirection = System.Windows.FlowDirection.LeftToRight;  
  
    // Cause the OnClipChange delegate to be called.  
    wfHost.Clip = new RectangleGeometry();  
  
    // Cause the OnBackgroundChange delegate to be called.  
    wfHost.Background = new ImageBrush();  
}
```

```

' The WindowLoaded method handles the Loaded event.
' It enables Windows Forms visual styles, creates
' a Windows Forms checkbox control, and assigns the
' control as the child of the WindowsFormsHost element.
' This method also modifies property mappings on the
' WindowsFormsHost element.
Private Sub WindowLoaded( _
ByVal sender As Object, _
ByVal e As RoutedEventArgs)

    System.Windows.Forms.Application.EnableVisualStyles()

    ' Create a Windows Forms checkbox control and assign
    ' it as the WindowsFormsHost element's child.
    Dim cb As New System.Windows.Forms.CheckBox()
    cb.Text = "Windows Forms checkbox"
    cb.Dock = DockStyle.Fill
    cb.TextAlign = ContentAlignment.MiddleCenter
    AddHandler cb.CheckedChanged, AddressOf cb_CheckedChanged
    wfHost.Child = cb

    ' Replace the default mapping for the FlowDirection property.
    Me.ReplaceFlowDirectionMapping()

    ' Remove the mapping for the Cursor property.
    Me.RemoveCursorMapping()

    ' Add the mapping for the Clip property.
    Me.AddClipMapping()

    ' Add another mapping for the Background property.
    Me.ExtendBackgroundMapping()

    ' Cause the OnFlowDirectionChange delegate to be called.
    wfHost.FlowDirection = System.Windows.FlowDirection.LeftToRight

    ' Cause the OnClipChange delegate to be called.
    wfHost.Clip = New RectangleGeometry()

    ' Cause the OnBackgroundChange delegate to be called.
    wfHost.Background = New ImageBrush()

End Sub

```

The `WindowLoaded` method handles the `Loaded` event and performs the following initialization.

- Creates a Windows Forms`CheckBox` control.
  - Calls the methods you defined earlier in the walkthrough to set up the property mappings.
  - Assigns initial values to the mapped properties.
2. Press **F5** to build and run the application. Click the check box to see the effect of the `FlowDirection` mapping. When you click the check box, the layout reverses its left-right orientation.

## See also

- [WindowsFormsHost.PropertyMap](#)
- [ElementHost.PropertyMap](#)
- [WindowsFormsHost](#)
- [Windows Forms and WPF Property Mapping](#)
- [Design XAML in Visual Studio](#)

- [Walkthrough: Hosting a Windows Forms Control in WPF](#)

# Walkthrough: Localizing a Hybrid Application

11/12/2019 • 3 minutes to read • [Edit Online](#)

This walkthrough shows you how to localize WPF elements in a Windows Forms-based hybrid application.

Tasks illustrated in this walkthrough include:

- Creating the Windows Forms host project.
- Adding localizable content.
- Enabling localization.
- Assigning resource identifiers.
- Using the LocBaml tool to produce a satellite assembly.

For a complete code listing of the tasks illustrated in this walkthrough, see [Localizing a Hybrid Application Sample](#).

When you are finished, you will have a localized hybrid application.

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio 2017

## Creating the Windows Forms Host Project

The first step is to create the Windows Forms application project and add a WPF element with content that you will localize.

### To create the host project

1. Create a **WPF App** project named `LocalizingWpfInWF`. (**File > New > Project > Visual C# or Visual Basic > Classic Desktop > WPF Application**).
2. Add a WPF `UserControl` element called `SimpleControl` to the project.
3. Use the `ElementHost` control to place a `SimpleControl` element on the form. For more information, see [Walkthrough: Hosting a 3-D WPF Composite Control in Windows Forms](#).

## Adding Localizable Content

Next, you will add a Windows Forms label control and set the WPF element's content to a localizable string.

### To add localizable content

1. In **Solution Explorer**, double-click `SimpleControl.xaml` to open it in the WPF Designer.
2. Set the content of the `Button` control using the following code.

```

<UserControl x:Class="LocalizingWpfInWf.SimpleControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <Canvas>
        <Button Content="Hello"/>
    </Canvas>
</UserControl>

```

3. In **Solution Explorer**, double-click **Form1** to open it in the Windows Forms Designer.
4. Open the **Toolbox** and double-click **Label** to add a label control to the form. Set the value of its **Text** property to `"Hello"`.
5. Press **F5** to build and run the application.

Both the `SimpleControl` element and the label control display the text **"Hello"**.

## Enabling Localization

The Windows Forms Designer provides settings for enabling localization in a satellite assembly.

### To enable localization

1. In **Solution Explorer**, double-click **Form1.cs** to open it in the Windows Forms Designer.
2. In the **Properties** window, set the value of the form's **Localizable** property to `true`.
3. In the **Properties** window, set the value of the **Language** property to **Spanish (Spain)**.
4. In the Windows Forms Designer, select the label control.
5. In the **Properties** window, set the value of the **Text** property to `"Hola"`.

A new resource file named Form1.es-ES.resx is added to the project.

6. In **Solution Explorer**, right-click **Form1.cs** and click **View Code** to open it in the Code Editor.
7. Copy the following code into the `Form1` constructor, preceding the call to `InitializeComponent`.

```

public Form1()
{
    System.Threading.Thread.CurrentThread.CurrentCulture = new System.Globalization.CultureInfo("es-ES");

    InitializeComponent();
}

```

8. In **Solution Explorer**, right-click **LocalizingWpfInWf** and click **Unload Project**.

The project name is labeled **(unavailable)**.

9. Right-click **LocalizingWpfInWf**, and click **Edit LocalizingWpfInWf.csproj**.

The project file opens in the Code Editor.

10. Copy the following line into the first `PropertyGroup` in the project file.

```
<UICulture>en-US</UICulture>
```

11. Save the project file and close it.
12. In **Solution Explorer**, right-click **LocalizingWpfInWf** and click **Reload Project**.

## Assigning Resource Identifiers

You can map your localizable content to resource assemblies by using resource identifiers. The MsBuild.exe application automatically assigns resource identifiers when you specify the `updateuid` option.

### To assign resource identifiers

1. From the Start Menu, open the Developer Command Prompt for Visual Studio.
2. Use the following command to assign resource identifiers to your localizable content.

```
msbuild -t:updateuid LocalizingWpfInWf.csproj
```

3. In **Solution Explorer**, double-click **SimpleControl.xaml** to open it in the Code Editor. You will see that the `msbuild` command has added the `Uid` attribute to all the elements. This facilitates localization through the assignment of resource identifiers.

```
<UserControl x:Uid="UserControl_1" x:Class="LocalizingWpfInWf.SimpleControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <Canvas x:Uid="Canvas_1">
        <Button x:Uid="Button_1" Content="Hello"/>
    </Canvas>
</UserControl>
```

4. Press **F6** to build the solution.

## Using LocBaml to Produce a Satellite Assembly

Your localized content is stored in a resource-only *satellite assembly*. Use the command-line tool LocBaml.exe to produce a localized assembly for your WPF content.

### To produce a satellite assembly

1. Copy LocBaml.exe to your project's obj\Debug folder. For more information, see [Localize an Application](#).
2. In the Command Prompt window, use the following command to extract resource strings into a temporary file.

```
LocBaml /parse LocalizingWpfInWf.g.en-US.resources /out:temp.csv
```

3. Open the temp.csv file with Visual Studio or another text editor. Replace the string `"Hello"` with its Spanish translation, `"Hola"`.
4. Save the temp.csv file.
5. Use the following command to generate the localized resource file.

```
LocBaml /generate /trans:temp.csv LocalizingWpfInWf.g.en-US.resources /out:.. /cul:es-ES
```

The LocalizingWpfInWf.g.es-ES.resources file is created in the obj\Debug folder.

6. Use the following command to build the localized satellite assembly.

```
A1.exe /out:LocalizingWpfInWF.resources.dll /culture:es-ES /embed:LocalizingWpfInWF.Form1.es-ES.resources /embed:LocalizingWpfInWF.g.es-ES.resources
```

The LocalizingWpfInWF.resources.dll file is created in the obj\Debug folder.

7. Copy the LocalizingWpfInWF.resources.dll file to the project's bin\Debug\es-ES folder. Replace the existing file.
8. Run LocalizingWpfInWF.exe, which is located in your project's bin\Debug folder. Do not rebuild the application or the satellite assembly will be overwritten.

The application shows the localized strings instead of the English strings.

## See also

- [ElementHost](#)
- [WindowsFormsHost](#)
- [Localize an Application](#)
- [Walkthrough: Localizing Windows Forms](#)
- [Design XAML in Visual Studio](#)

# WPF and Win32 Interoperation

10/24/2019 • 10 minutes to read • [Edit Online](#)

This topic provides an overview of how to interoperate WPF and Win32 code. Windows Presentation Foundation (WPF) provides a rich environment for creating applications. However, when you have a substantial investment in Win32 code, it might be more effective to reuse some of that code.

## WPF and Win32 Interoperation Basics

There are two basic techniques for interoperation between WPF and Win32 code.

- Host WPF content in a Win32 window. With this technique, you can use the advanced graphics capabilities of WPF within the framework of a standard Win32 window and application.
- Host a Win32 window in WPF content. With this technique, you can use an existing custom Win32 control in the context of other WPF content, and pass data across the boundaries.

Each of these techniques is conceptually introduced in this topic. For a more code-oriented illustration of hosting WPF in Win32, see [Walkthrough: Hosting WPF Content in Win32](#). For a more code-oriented illustration of hosting Win32 in WPF, see [Walkthrough: Hosting a Win32 Control in WPF](#).

## WPF Interoperation Projects

WPF APIs are managed code, but most existing Win32 programs are written in unmanaged C++. You cannot call WPF APIs from a true unmanaged program. However, by using the `/clr` option with the Microsoft Visual C++ compiler, you can create a mixed managed-unmanaged program where you can seamlessly mix managed and unmanaged API calls.

One project-level complication is that you cannot compile Extensible Application Markup Language (XAML) files into a C++ project. There are several project division techniques to compensate for this.

- Create a C# DLL that contains all your XAML pages as a compiled assembly, and then have your C++ executable include that DLL as a reference.
- Create a C# executable for the WPF content, and have it reference a C++ DLL that contains the Win32 content.
- Use [Load](#) to load any XAML at run time, instead of compiling your XAML.
- Do not use XAML at all, and write all your WPF in code, building up the element tree from [Application](#).

Use whatever approach works best for you.

### NOTE

If you have not used C++/CLI before, you might notice some "new" keywords such as `gcnew` and `nullptr` in the interoperation code examples. These keywords supersede the older double-underscore syntax (`__gc`) and provide a more natural syntax for managed code in C++. To learn more about the C++/CLI managed features, see [Component Extensions for Runtime Platforms](#).

## How WPF Uses Hwnds

To make the most of WPF "HWND interop", you need to understand how WPF uses HWNDs. For any HWND, you cannot mix WPF rendering with DirectX rendering or GDI / GDI+ rendering. This has a number of implications. Primarily, in order to mix these rendering models at all, you must create an interoperation solution, and use designated segments of interoperation for each rendering model that you choose to use. Also, the rendering behavior creates an "airspace" restriction for what your interoperation solution can accomplish. The "airspace" concept is explained in greater detail in the topic [Technology Regions Overview](#).

All WPF elements on the screen are ultimately backed by a HWND. When you create a WPF [Window](#), WPF creates a top-level HWND, and uses an [HwndSource](#) to put the [Window](#) and its WPF content inside the HWND. The rest of your WPF content in the application shares that singular HWND. An exception is menus, combo box drop downs, and other pop-ups. These elements create their own top-level window, which is why a WPF menu can potentially go past the edge of the window HWND that contains it. When you use [HwndHost](#) to put an HWND inside WPF, WPF informs Win32 how to position the new child HWND relative to the WPF [Window](#) HWND.

A related concept to HWND is transparency within and between each HWND. This is also discussed in the topic [Technology Regions Overview](#).

## Hosting WPF Content in a Microsoft Win32 Window

The key to hosting a WPF on a Win32 window is the [HwndSource](#) class. This class wraps the WPF content in a Win32 window, so that the WPF content can be incorporated into your user interface (UI) as a child window. The following approach combines the Win32 and WPF in a single application.

1. Implement your WPF content (the content root element) as a managed class. Typically, the class inherits from one of the classes that can contain multiple child elements and/or used as a root element, such as [DockPanel](#) or [Page](#). In subsequent steps, this class is referred to as the WPF content class, and instances of the class are referred to as WPF content objects.
2. Implement a Windows application with C++/CLI. If you are starting with an existing unmanaged C++ application, you can usually enable it to call managed code by changing your project settings to include the `/clr` compiler flag (the full scope of what might be necessary to support `/clr` compilation is not described in this topic).
3. Set the threading model to Single Threaded Apartment (STA). WPF uses this threading model.
4. Handle the WM\_CREATE notification in your window procedure.
5. Within the handler (or a function that the handler calls), do the following:
  - a. Create a new [HwndSource](#) object with the parent window HWND as its `parent` parameter.
  - b. Create an instance of your WPF content class.
  - c. Assign a reference to the WPF content object to the [HwndSource](#) object [RootVisual](#) property.
  - d. The [HwndSource](#) object [Handle](#) property contains the window handle (HWND). To get an HWND that you can use in the unmanaged part of your application, cast `Handle.ToPointer()` to an HWND.
6. Implement a managed class that contains a static field that holds a reference to your WPF content object. This class allows you to get a reference to the WPF content object from your Win32 code, but more importantly it prevents your [HwndSource](#) from being inadvertently garbage collected.
7. Receive notifications from the WPF content object by attaching a handler to one or more of the WPF content object events.
8. Communicate with the WPF content object by using the reference that you stored in the static field to set properties, call methods, etc.

#### NOTE

You can do some or all of the WPF content class definition for Step One in XAML using the default partial class of the content class, if you produce a separate assembly and then reference it. Although you typically include an [Application](#) object as part of compiling the XAML into an assembly, you do not end up using that [Application](#) as part of the interoperation, you just use one or more of the root classes for XAML files referred to by the application and reference their partial classes. The remainder of the procedure is essentially similar to that outlined above.

Each of these steps is illustrated through code in the topic [Walkthrough: Hosting WPF Content in Win32](#).

## Hosting a Microsoft Win32 Window in WPF

The key to hosting a Win32 window within other WPF content is the [HwndHost](#) class. This class wraps the window in a WPF element which can be added to a WPF element tree. [HwndHost](#) also supports APIs that allow you to do such tasks as process messages for the hosted window. The basic procedure is:

1. Create an element tree for a WPF application (can be through code or markup). Find an appropriate and permissible point in the element tree where the [HwndHost](#) implementation can be added as a child element. In the remainder of these steps, this element is referred to as the reserving element.
2. Derive from [HwndHost](#) to create an object that holds your Win32 content.
3. In that host class, override the [HwndHost](#) method [BuildWindowCore](#). Return the HWND of the hosted window. You might want to wrap the actual control(s) as a child window of the returned window; wrapping the controls in a host window provides a simple way for your WPF content to receive notifications from the controls. This technique helps correct for some Win32 issues regarding message handling at the hosted control boundary.
4. Override the [HwndHost](#) methods [DestroyWindowCore](#) and [WndProc](#). The intention here is to process cleanup and remove references to the hosted content, particularly if you created references to unmanaged objects.
5. In your code-behind file, create an instance of the control hosting class and make it a child of the reserving element. Typically you would use an event handler such as [Loaded](#), or use the partial class constructor. But you could also add the interoperation content through a runtime behavior.
6. Process selected window messages, such as control notifications. There are two approaches. Both provide identical access to the message stream, so your choice is largely a matter of programming convenience.
  - Implement message processing for all messages (not just shutdown messages) in your override of the [HwndHost](#) method [WndProc](#).
  - Have the hosting WPF element process the messages by handling the [MessageHook](#) event. This event is raised for every message that is sent to the main window procedure of the hosted window.
  - You cannot process messages from windows that are out of process using [WndProc](#).
7. Communicate with the hosted window by using platform invoke to call the unmanaged [SendMessage](#) function.

Following these steps creates an application that works with mouse input. You can add tabbing support for your hosted window by implementing the [IKeyboardInputSink](#) interface.

Each of these steps is illustrated through code in the topic [Walkthrough: Hosting a Win32 Control in WPF](#).

### Hwnds Inside WPF

You can think of [HwndHost](#) as a special control. (Technically, [HwndHost](#) is a [FrameworkElement](#) derived class,

not a [Control](#) derived class, but it can be considered a control for purposes of interoperation.) [HwndHost](#) abstracts the underlying Win32 nature of the hosted content such that the remainder of WPF considers the hosted content to be another control-like object, which should render and process input. [HwndHost](#) generally behaves like any other WPF [FrameworkElement](#), although there are some important differences around output (drawing and graphics) and input (mouse and keyboard) based on limitations of what the underlying HWNDs can support.

#### Notable Differences in Output Behavior

- [FrameworkElement](#), which is the [HwndHost](#) base class, has quite a few properties that imply changes to the UI. These include properties such as [FrameworkElement.FlowDirection](#), which changes the layout of elements within that element as a parent. However, most of these properties are not mapped to possible Win32 equivalents, even if such equivalents might exist. Too many of these properties and their meanings are too rendering-technology specific for mappings to be practical. Therefore, setting properties such as [FlowDirection](#) on [HwndHost](#) has no effect.
- [HwndHost](#) cannot be rotated, scaled, skewed, or otherwise affected by a [Transform](#).
- [HwndHost](#) does not support the [Opacity](#) property (alpha blending). If content inside the [HwndHost](#) performs [System.Drawing](#) operations that include alpha information, that is itself not a violation, but the [HwndHost](#) as a whole only supports Opacity = 1.0 (100%).
- [HwndHost](#) will appear on top of other WPF elements in the same top-level window. However, a [ToolTip](#) or [ContextMenu](#) generated menu is a separate top-level window, and so will behave correctly with [HwndHost](#).
- [HwndHost](#) does not respect the clipping region of its parent [UIElement](#). This is potentially an issue if you attempt to put an [HwndHost](#) class inside a scrolling region or [Canvas](#).

#### Notable Differences in Input Behavior

- In general, while input devices are scoped within the [HwndHost](#) hosted Win32 region, input events go directly to Win32.
- While the mouse is over the [HwndHost](#), your application does not receive WPF mouse events, and the value of the WPF property [IsMouseOver](#) will be `false`.
- While the [HwndHost](#) has keyboard focus, your application will not receive WPF keyboard events and the value of the WPF property [IsKeyboardFocusWithin](#) will be `false`.
- When focus is within the [HwndHost](#) and changes to another control inside the [HwndHost](#), your application will not receive the WPF events [GotFocus](#) or [LostFocus](#).
- Related stylus properties and events are analogous, and do not report information while the stylus is over [HwndHost](#).

## Tabbing, Mnemonics, and Accelerators

The [IKeyboardInputSink](#) and [IKeyboardInputSite](#) interfaces allow you to create a seamless keyboard experience for mixed WPF and Win32 applications:

- Tabbing between Win32 and WPF components
- Mnemonics and accelerators that work both when focus is within a Win32 component and when it is within a WPF component.

The [HwndHost](#) and [HwndSource](#) classes both provide implementations of [IKeyboardInputSink](#), but they may not handle all the input messages that you want for more advanced scenarios. Override the appropriate methods to get the keyboard behavior you want.

The interfaces only provide support for what happens on the transition between the WPF and Win32 regions. Within the Win32 region, tabbing behavior is entirely controlled by the Win32 implemented logic for tabbing, if any.

## See also

- [HwndHost](#)
- [HwndSource](#)
- [System.Windows.Interop](#)
- [Walkthrough: Hosting a Win32 Control in WPF](#)
- [Walkthrough: Hosting WPF Content in Win32](#)

# Technology Regions Overview

8/29/2019 • 3 minutes to read • [Edit Online](#)

If multiple presentation technologies are used in an application, such as WPF, Win32, or DirectX, they must share the rendering areas within a common top-level window. This topic describes issues that might influence the presentation and input for your WPF interoperation application.

## Regions

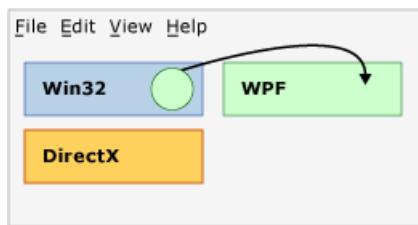
Within a top-level window, you can conceptualize that each HWND that comprises one of the technologies of an interoperation application has its own region (also called "airspace"). Each pixel within the window belongs to exactly one HWND, which constitutes the region of that HWND. (Strictly speaking, there is more than one WPF region if there is more than one WPF HWND, but for purposes of this discussion, you can assume there is only one). The region implies that all layers or other windows that attempt to render above that pixel during application lifetime must be part of the same render-level technology. Attempting to render WPF pixels over Win32 leads to undesirable results, and is disallowed as much as possible through the interoperation APIs.

### Region Examples

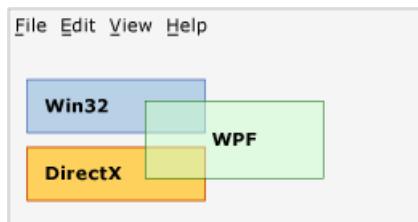
The following illustration shows an application that mixes Win32, DirectX, and WPF. Each technology uses its own separate, non-overlapping set of pixels, and there are no region issues.



Suppose that this application uses the mouse pointer position to create an animation that attempts to render over any of these three regions. No matter which technology was responsible for the animation itself, that technology would violate the region of the other two. The following illustration shows an attempt to render a WPF circle over a Win32 region.



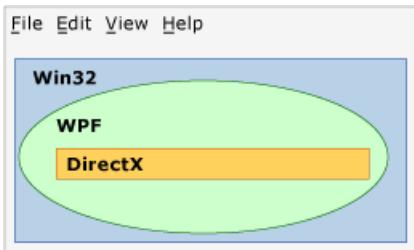
Another violation is if you try to use transparency/alpha blending between different technologies. In the following illustration, the WPF box violates the Win32 and DirectX regions. Because pixels in that WPF box are semi-transparent, they would have to be owned jointly by both DirectX and WPF, which is not possible. So this is another violation and cannot be built.



The previous three examples used rectangular regions, but different shapes are possible. For example, a region can have a hole. The following illustration shows a Win32 region with a rectangular hole this is the size of the WPF and DirectX regions combined.



Regions can also be completely nonrectangular, or any shape describable by a Win32 HRGN (region).



## Transparency and Top-Level Windows

The window manager in Windows only really processes Win32 HWNDs. Therefore, every WPF [Window](#) is an HWND. The [Window](#) HWND must abide by the general rules for any HWND. Within that HWND, WPF code can do whatever the overall WPF APIs support. But for interactions with other HWNDs on the desktop, WPF must abide by Win32 processing and rendering rules. WPF supports non-rectangular windows by using Win32 APIs—HRGNs for non-rectangular windows, and layered windows for a per-pixel alpha.

Constant alpha and color keys are not supported. Win32 layered window capabilities vary by platform.

Layered windows can make the entire window translucent (semi-transparent) by specifying an alpha value to apply to every pixel in the window. (Win32 in fact supports per-pixel alpha, but this is very difficult to use in practical programs because in this mode you would need to draw any child HWND yourself, including dialogs and dropdowns).

WPF supports HRGNs; however, there are no managed APIs for this functionality. You can use `platform invoke` and [HwndSource](#) to call the relevant Win32 APIs. For more information, see [Calling Native Functions from Managed Code](#).

WPF layered windows have different capabilities on different operating systems. This is because WPF uses DirectX to render, and layered windows were primarily designed for GDI rendering, not DirectX rendering.

- WPF supports hardware accelerated layered windows.
- WPF does not support transparency color keys, because WPF cannot guarantee to render the exact color you requested, particularly when rendering is hardware-accelerated.

## See also

- [WPF and Win32 Interoperation](#)
- [Walkthrough: Hosting a WPF Clock in Win32](#)
- [Hosting Win32 Content in WPF](#)

# Sharing Message Loops Between Win32 and WPF

7/12/2019 • 3 minutes to read • [Edit Online](#)

This topic describes how to implement a message loop for interoperation with Windows Presentation Foundation (WPF), either by using existing message loop exposure in [Dispatcher](#) or by creating a separate message loop on the Win32 side of your interoperation code.

## ComponentDispatcher and the Message Loop

A normal scenario for interoperation and keyboard event support is to implement [IKeyboardInputSink](#), or to subclass from classes that already implement [IKeyboardInputSink](#), such as [HwndSource](#) or [HwndHost](#). However, keyboard sink support does not address all possible message loop needs you might have when sending and receiving messages across your interoperation boundaries. To help formalize an application message loop architecture, Windows Presentation Foundation (WPF) provides the [ComponentDispatcher](#) class, which defines a simple protocol for a message loop to follow.

[ComponentDispatcher](#) is a static class that exposes several members. The scope of each method is implicitly tied to the calling thread. A message loop must call some of those APIs at critical times (as defined in the next section).

[ComponentDispatcher](#) provides events that other components (such as the keyboard sink) can listen for. The [Dispatcher](#) class calls all the appropriate [ComponentDispatcher](#) methods in an appropriate sequence. If you are implementing your own message loop, your code is responsible for calling [ComponentDispatcher](#) methods in a similar fashion.

Calling [ComponentDispatcher](#) methods on a thread will only invoke event handlers that were registered on that thread.

## Writing Message Loops

The following is a checklist of [ComponentDispatcher](#) members you will use if you write your own message loop:

- [PushModal](#): your message loop should call this to indicate that the thread is modal.
- [PopModal](#): your message loop should call this to indicate that the thread has reverted to nonmodal.
- [Raiseldle](#): your message loop should call this to indicate that [ComponentDispatcher](#) should raise the [ThreadIdle](#) event. [ComponentDispatcher](#) will not raise [ThreadIdle](#) if [IsThreadModal](#) is `true`, but message loops may choose to call [Raiseldle](#) even if [ComponentDispatcher](#) cannot respond to it while in modal state.
- [RaiseThreadMessage](#): your message loop should call this to indicate that a new message is available. The return value indicates whether a listener to a [ComponentDispatcher](#) event handled the message. If [RaiseThreadMessage](#) returns `true` (handled), the dispatcher should do nothing further with the message. If the return value is `false`, the dispatcher is expected to call the Win32 function [TranslateMessage](#), then call [DispatchMessage](#).

## Using ComponentDispatcher and Existing Message Handling

The following is a checklist of [ComponentDispatcher](#) members you will use if you rely on the inherent WPF message loop.

- [IsThreadModal](#): returns whether the application has gone modal (e.g., a modal message loop has been pushed). [ComponentDispatcher](#) can track this state because the class maintains a count of [PushModal](#) and

[PopModal](#) calls from the message loop.

- [ThreadFilterMessage](#) and [ThreadPreprocessMessage](#) events follow the standard rules for delegate invocations. Delegates are invoked in an unspecified order, and all delegates are invoked even if the first one marks the message as handled.
- [ThreadIdle](#): indicates an appropriate and efficient time to do idle processing (there are no other pending messages for the thread). [ThreadIdle](#) will not be raised if the thread is modal.
- [ThreadFilterMessage](#): raised for all messages that the message pump processes.
- [ThreadPreprocessMessage](#): raised for all messages that were not handled during [ThreadFilterMessage](#).

A message is considered handled if after the [ThreadFilterMessage](#) event or [ThreadPreprocessMessage](#) event, the `handled` parameter passed by reference in event data is `true`. Event handlers should ignore the message if `handled` is `true`, because that means the different handler handled the message first. Event handlers to both events may modify the message. The dispatcher should dispatch the modified message and not the original unchanged message. [ThreadPreprocessMessage](#) is delivered to all listeners, but the architectural intention is that only the top-level window containing the HWND at which the messages targeted should invoke code in response to the message.

## How HwndSource Treats ComponentDispatcher Events

If the [HwndSource](#) is a top-level window (no parent HWND), it will register with [ComponentDispatcher](#). If [ThreadPreprocessMessage](#) is raised, and if the message is intended for the [HwndSource](#) or child windows, [HwndSource](#) calls its [IKeyboardInputSink.TranslateAccelerator](#), [TranslateChar](#), [OnMnemonic](#) keyboard sink sequence.

If the [HwndSource](#) is not a top-level window (has a parent HWND), there will be no handling. Only the top level window is expected to do the handling, and there is expected to be a top level window with keyboard sink support as part of any interoperation scenario.

If [WndProc](#) on an [HwndSource](#) is called without an appropriate keyboard sink method being called first, your application will receive the higher level keyboard events such as [KeyDown](#). However, no keyboard sink methods will be called, which circumvents desirable keyboard input model features such as access key support. This might happen because the message loop did not properly notify the relevant thread on the [ComponentDispatcher](#), or because the parent HWND did not invoke the proper keyboard sink responses.

A message that goes to the keyboard sink might not be sent to the HWND if you added hooks for that message by using the [AddHook](#) method. The message might have been handled at the message pump level directly and not submitted to the `DispatchMessage` function.

## See also

- [ComponentDispatcher](#)
- [IKeyboardInputSink](#)
- [WPF and Win32 Interoperation](#)
- [Threading Model](#)
- [Input Overview](#)

# Hosting Win32 Content in WPF

10/25/2019 • 8 minutes to read • [Edit Online](#)

## Prerequisites

See [WPF and Win32 Interoperation](#).

## A Walkthrough of Win32 Inside Windows Presentation Framework (HwndHost)

To reuse Win32 content inside WPF applications, use [HwndHost](#), which is a control that makes HWNDs look like WPF content. Like [HwndSource](#), [HwndHost](#) is straightforward to use: derive from [HwndHost](#) and implement [BuildWindowCore](#) and [DestroyWindowCore](#) methods, then instantiate your [HwndHost](#) derived class and place it inside your WPF application.

If your Win32 logic is already packaged as a control, then your [BuildWindowCore](#) implementation is little more than a call to [CreateWindow](#). For example, to create a Win32 LISTBOX control in C++:

```
virtual HandleRef BuildWindowCore(HandleRef hwndParent) override {
    HWND handle = CreateWindowEx(0, L"LISTBOX",
        L"this is a Win32 listbox",
        WS_CHILD | WS_VISIBLE | LBS_NOTIFY
        | WS_VSCROLL | WS_BORDER,
        0, 0, // x, y
        30, 70, // height, width
        (HWND) hwndParent.Handle.ToPointer(), // parent hwnd
        0, // hmenu
        0, // hinstance
        0); // lparam

    return HandleRef(this, IntPtr(handle));
}

virtual void DestroyWindowCore(HandleRef hwnd) override {
    // HwndHost will dispose the hwnd for us
}
```

But suppose the Win32 code is not quite so self-contained? If so, you can create a Win32 dialog box and embed its contents into a larger WPF application. The sample shows this in Visual Studio and C++, although it is also possible to do this in a different language or at the command line.

Start with a simple dialog, which is compiled into a C++ DLL project.

Next, introduce the dialog into the larger WPF application:

- Compile the DLL as managed ([/clr](#))
- Turn the dialog into a control
- Define the derived class of [HwndHost](#) with [BuildWindowCore](#) and [DestroyWindowCore](#) methods
- Override [TranslateAccelerator](#) method to handle dialog keys
- Override [TabInto](#) method to support tabbing
- Override [OnMnemonic](#) method to support mnemonics

- Instantiate the `HwndHost` subclass and put it under the right WPF element

### Turn the Dialog into a Control

You can turn a dialog box into a child HWND using the WS\_CHILD and DS\_CONTROL styles. Go into the resource file (.rc) where the dialog is defined, and find the beginning of the definition of the dialog:

```
IDD_DIALOG1 DIALOGEX 0, 0, 303, 121
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION | WS_SYSMENU
```

Change the second line to:

```
STYLE DS_SETFONT | WS_CHILD | WS_BORDER | DS_CONTROL
```

This action does not fully package it into a self-contained control; you still need to call `IsDialogMessage()` so Win32 can process certain messages, but the control change does provide a straightforward way of putting those controls inside another HWND.

## Subclass HwndHost

Import the following namespaces:

```
namespace ManagedCpp
{
    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Input;
    using namespace System::Windows::Media;
    using namespace System::Runtime::InteropServices;
```

Then create a derived class of `HwndHost` and override the `BuildWindowCore` and `DestroyWindowCore` methods:

```
public ref class MyHwndHost : public HwndHost, IKeyboardInputSink {
private:
    HWND dialog;

protected:
    virtual HandleRef BuildWindowCore(HandleRef hwndParent) override {
        InitializeGlobals();
        dialog = CreateDialog(hInstance,
            MAKEINTRESOURCE(IDD_DIALOG1),
            (HWND) hwndParent.Handle.ToPointer(),
            (DLGPROC) About);
        return HandleRef(this, IntPtr(dialog));
    }

    virtual void DestroyWindowCore(HandleRef hwnd) override {
        // hwnd will be disposed for us
    }
}
```

Here you use the `CreateDialog` to create the dialog box that is really a control. Since this is one of the first methods called inside the DLL, you should also do some standard Win32 initialization by calling a function you will define later, called `InitializeGlobals()`:

```
bool initialized = false;
void InitializeGlobals() {
    if (initialized) return;
    initialized = true;

    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_TYPICALWIN32DIALOG, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);
```

### Override TranslateAccelerator Method to Handle Dialog Keys

If you ran this sample now, you would get a dialog control that displays, but it would ignore all of the keyboard processing that makes a dialog box a functional dialog box. You should now override the `TranslateAccelerator` implementation (which comes from `IKeyboardInputSink`, an interface that `HwndHost` implements). This method gets called when the application receives WM\_KEYDOWN and WM\_SYSKEYDOWN.

```

#define TranslateAccelerator
    virtual bool TranslateAccelerator(System::Windows::Interop::MSG% msg,
        ModifierKeys modifiers) override
    {
        ::MSG m = ConvertMessage(msg);

        // Win32's IsDialogMessage() will handle most of our tabbing, but doesn't know
        // what to do when it reaches the last tab stop
        if (m.message == WM_KEYDOWN && m.wParam == VK_TAB) {
            HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);
            HWND lastTabStop = GetDlgItem(dialog, IDCANCEL);
            TraversalRequest^ request = nullptr;

            if (GetKeyState(VK_SHIFT) && GetFocus() == firstTabStop) {
                // this code should work, but there's a bug with interop shift-tab in current builds
                request = gcnew TraversalRequest(FocusNavigationDirection::Last);
            }
            else if (!GetKeyState(VK_SHIFT) && GetFocus() == lastTabStop) {
                request = gcnew TraversalRequest(FocusNavigationDirection::Next);
            }

            if (request != nullptr)
                return ((IKeyboardInputSink^) this)->KeyboardInputSite->OnNoMoreTabStops(request);
        }

        // Only call IsDialogMessage for keys it will do something with.
        if (msg.message == WM_SYSKEYDOWN || msg.message == WM_KEYDOWN) {
            switch (m.wParam) {
                case VK_TAB:
                case VK_LEFT:
                case VK_UP:
                case VK_RIGHT:
                case VK_DOWN:
                case VK_EXECUTE:
                case VK_RETURN:
                case VK_ESCAPE:
                case VK_CANCEL:
                    IsDialogMessage(dialog, &m);
                    // IsDialogMessage should be called ProcessDialogMessage --
                    // it processes messages without ever really telling you
                    // if it handled a specific message or not
                    return true;
            }
        }
    }

    return false; // not a key we handled
}

```

This is a lot of code in one piece, so it could use some more detailed explanations. First, the code using C++ and C++ macros; you need to be aware that there is already a macro named `TranslateAccelerator`, which is defined in `winuser.h`:

```
#define TranslateAccelerator TranslateAcceleratorW
```

So make sure to define a `TranslateAccelerator` method and not a `TranslateAcceleratorW` method.

Similarly, there is both the unmanaged `winuser.h` `MSG` and the managed `Microsoft::Win32::MSG` struct. You can disambiguate between the two using the C++ `::` operator.

```

virtual bool TranslateAccelerator(System::Windows::Interop::MSG% msg,
    ModifierKeys modifiers) override
{
    ::MSG m = ConvertMessage(msg);
}

```

Both MSGs have the same data, but sometimes it is easier to work with the unmanaged definition, so in this sample you can define the obvious conversion routine:

```

```cpp
::MSG ConvertMessage(System::Windows::Interop::MSG% msg) {
    ::MSG m;
    m.hwnd = (HWND) msg(hwnd).ToPointer();
    m.lParam = (LPARAM) msg.lParam.ToPointer();
    m.message = msg.message;
    m.wParam = (WPARAM) msg.wParam.ToPointer();

    m.time = msg.time;

    POINT pt;
    pt.x = msg.pt_x;
    pt.y = msg.pt_y;
    m.pt = pt;

    return m;
}

```

Back to `TranslateAccelerator`. The basic principle is to call the Win32 function `IsDialogMessage` to do as much work as possible, but `IsDialogMessage` does not have access to anything outside the dialog. As a user tabs around the dialog, when tabbing runs past the last control in our dialog, you need to set focus to the WPF portion by calling `IKeyboardInputSite::OnNoMoreStops`.

```

// Win32's IsDialogMessage() will handle most of the tabbing, but doesn't know
// what to do when it reaches the last tab stop
if (m.message == WM_KEYDOWN && m.wParam == VK_TAB) {
    HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);
    HWND lastTabStop = GetDlgItem(dialog, IDCANCEL);
    TraversalRequest^ request = nullptr;

    if (GetKeyState(VK_SHIFT) && GetFocus() == firstTabStop) {
        request = gcnew TraversalRequest(FocusNavigationDirection::Last);
    }
    else if (!GetKeyState(VK_SHIFT) && GetFocus() == lastTabStop) {
        request = gcnew TraversalRequest(FocusNavigationDirection::Next);
    }

    if (request != nullptr)
        return ((IKeyboardInputSink^) this)->KeyboardInputSite->OnNoMoreTabStops(request);
}

```

Finally, call `IsDialogMessage`. But one of the responsibilities of a `TranslateAccelerator` method is telling WPF whether you handled the keystroke or not. If you did not handle it, the input event can tunnel and bubble through the rest of the application. Here, you will expose a quirk of keyboard message handling and the nature of the input architecture in Win32. Unfortunately, `IsDialogMessage` does not return in any way whether it handles a particular keystroke. Even worse, it will call `DispatchMessage()` on keystrokes it should not handle! So you will have to reverse-engineer `IsDialogMessage`, and only call it for the keys you know it will handle:

```

// Only call IsDialogMessage for keys it will do something with.
if (msg.message == WM_SYSKEYDOWN || msg.message == WM_KEYDOWN) {
    switch (m.wParam) {
        case VK_TAB:
        case VK_LEFT:
        case VK_UP:
        case VK_RIGHT:
        case VK_DOWN:
        case VK_EXECUTE:
        case VK_RETURN:
        case VK_ESCAPE:
        case VK_CANCEL:
            IsDialogMessage(dialog, &m);
            // IsDialogMessage should be called ProcessDialogMessage --
            // it processes messages without ever really telling you
            // if it handled a specific message or not
            return true;
    }
}

```

## Override TabInto Method to Support Tabbing

Now that you have implemented `TranslateAccelerator`, a user can tab around inside the dialog box and tab out of it into the greater WPF application. But a user cannot tab back into the dialog box. To solve that, you override `TabInto`:

```

public:
    virtual bool TabInto(TraversalRequest^ request) override {
        if (request->FocusNavigationDirection == FocusNavigationDirection::Last) {
            HWND lastTabStop = GetDlgItem(dialog, IDCANCEL);
            SetFocus(lastTabStop);
        }
        else {
            HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);
            SetFocus(firstTabStop);
        }
        return true;
    }
}

```

The `TraversalRequest` parameter tells you whether the tab action is a tab or shift tab.

## Override OnMnemonic Method to Support Mnemonics

Keyboard handling is almost complete, but there is one thing missing – mnemonics do not work. If a user presses alt-F, focus does not jump to the "First name:" edit box. So, you override the `OnMnemonic` method:

```

virtual bool OnMnemonic(System::Windows::Interop::MSG% msg, ModifierKeys modifiers) override {
    ::MSG m = ConvertMessage(msg);

    // If it's one of our mnemonics, set focus to the appropriate hwnd
    if (msg.message == WM_SYSCHAR && GetKeyState(VK_MENU /*alt*/) < 0) {
        int dialogitem = 9999;
        switch (m.wParam) {
            case 's': dialogitem = IDOK; break;
            case 'c': dialogitem = IDCANCEL; break;
            case 'f': dialogitem = IDC_EDIT1; break;
            case 'l': dialogitem = IDC_EDIT2; break;
            case 'p': dialogitem = IDC_EDIT3; break;
            case 'a': dialogitem = IDC_EDIT4; break;
            case 'i': dialogitem = IDC_EDIT5; break;
            case 't': dialogitem = IDC_EDIT6; break;
            case 'z': dialogitem = IDC_EDIT7; break;
        }
        if (dialogitem != 9999) {
            HWND hwnd = GetDlgItem(dialog, dialogitem);
            SetFocus(hwnd);
            return true;
        }
    }
    return false; // key unhandled
};


```

Why not call `IsDialogMessage` here? You have the same issue as before--you need to be able to inform WPF code whether your code handled the keystroke or not, and `IsDialogMessage` cannot do that. There is also a second issue, because `IsDialogMessage` refuses to process the mnemonic if the focused HWND is not inside the dialog box.

### Instantiate the `HwndHost` Derived Class

Finally, now that all the key and tab support is in place, you can put your `HwndHost` into the larger WPF application. If the main application is written in XAML, the easiest way to put it in the right place is to leave an empty `Border` element where you want to put the `HwndHost`. Here you create a `Border` named

`insertHwndHostHere`:

```

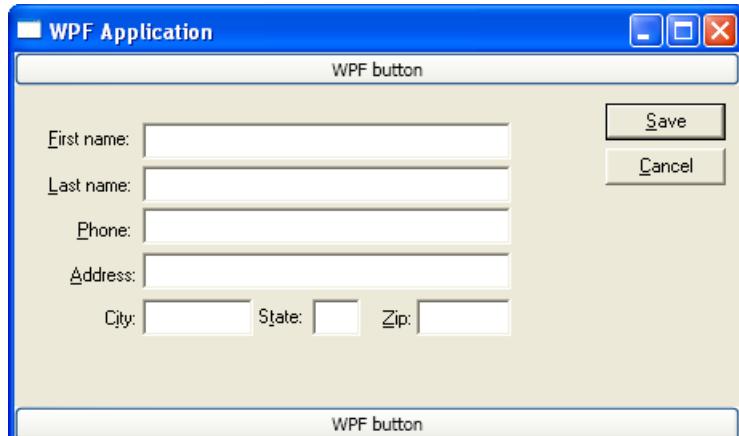
<Window x:Class="WPFApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Windows Presentation Framework Application"
    Loaded="Window1_Loaded"
    >
    <StackPanel>
        <Button Content="WPF button"/>
        <Border Name="insertHwndHostHere" Height="200" Width="500"/>
        <Button Content="WPF button"/>
    </StackPanel>
</Window>

```

Then all that remains is to find a good place in code sequence to instantiate the `HwndHost` and connect it to the `Border`. In this example, you will put it inside the constructor for the `Window` derived class:

```
public partial class Window1 : Window {  
    public Window1() {  
    }  
  
    void Window1_Loaded(object sender, RoutedEventArgs e) {  
        HwndHost host = new ManagedCpp.MyHwndHost();  
        insertHwndHostHere.Child = host;  
    }  
}
```

Which gives you:



## See also

- [WPF and Win32 Interoperation](#)

# Walkthrough: Hosting a Win32 Control in WPF

10/29/2019 • 15 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a rich environment for creating applications. However, when you have a substantial investment in Win32 code, it may be more effective to reuse at least some of that code in your WPF application rather than rewrite it completely. WPF provides a straightforward mechanism for hosting a Win32 window, on a WPF page.

This topic walks you through an application, [Hosting a Win32 ListBox Control in WPF Sample](#), that hosts a Win32 list box control. This general procedure can be extended to hosting any Win32 window.

## Requirements

This topic assumes a basic familiarity with both WPF and Windows API programming. For a basic introduction to WPF programming, see [Getting Started](#). For an introduction to Windows API programming, see any of the numerous books on the subject, in particular *Programming Windows* by Charles Petzold.

Because the sample that accompanies this topic is implemented in C#, it makes use of Platform Invocation Services (PInvoke) to access the Windows API. Some familiarity with PInvoke is helpful but not essential.

### NOTE

This topic includes a number of code examples from the associated sample. However, for readability, it does not include the complete sample code. You can obtain or view complete code from [Hosting a Win32 ListBox Control in WPF Sample](#).

## The Basic Procedure

This section outlines the basic procedure for hosting a Win32 window on a WPF page. The remaining sections go through the details of each step.

The basic hosting procedure is:

1. Implement a WPF page to host the window. One technique is to create a [Border](#) element to reserve a section of the page for the hosted window.
2. Implement a class to host the control that inherits from [HwndHost](#).
3. In that class, override the [HwndHost](#) class member [BuildWindowCore](#).
4. Create the hosted window as a child of the window that contains the WPF page. Although conventional WPF programming does not need to explicitly make use of it, the hosting page is a window with a handle (HWND). You receive the page HWND through the `hwndParent` parameter of the [BuildWindowCore](#) method. The hosted window should be created as a child of this HWND.
5. Once you have created the host window, return the HWND of the hosted window. If you want to host one or more Win32 controls, you typically create a host window as a child of the HWND and make the controls children of that host window. Wrapping the controls in a host window provides a simple way for your WPF page to receive notifications from the controls, which deals with some particular Win32 issues with notifications across the HWND boundary.
6. Handle selected messages sent to the host window, such as notifications from child controls. There are two ways to do this.

- If you prefer to handle messages in your hosting class, override the [WndProc](#) method of the [HwndHost](#) class.
  - If you prefer to have the WPF handle the messages, handle the [HwndHost](#) class [MessageHook](#) event in your code-behind. This event occurs for every message that is received by the hosted window. If you choose this option, you must still override [WndProc](#), but you only need a minimal implementation.
7. Override the [DestroyWindowCore](#) and [WndProc](#) methods of [HwndHost](#). You must override these methods to satisfy the [HwndHost](#) contract, but you may only need to provide a minimal implementation.
  8. In your code-behind file, create an instance of the control hosting class and make it a child of the [Border](#) element that is intended to host the window.
  9. Communicate with the hosted window by sending it Microsoft Windows messages and handling messages from its child windows, such as notifications sent by controls.

## Implement the Page Layout

The layout for the WPF page that hosts the ListBox Control consists of two regions. The left side of the page hosts several WPF controls that provide a user interface (UI) that allows you to manipulate the Win32 control. The upper right corner of the page has a square region for the hosted ListBox Control.

The code to implement this layout is quite simple. The root element is a [DockPanel](#) that has two child elements. The first is a [Border](#) element that hosts the ListBox Control. It occupies a 200x200 square in the upper right corner of the page. The second is a [StackPanel](#) element that contains a set of WPF controls that display information and allow you to manipulate the ListBox Control by setting exposed interoperation properties. For each of the elements that are children of the [StackPanel](#), see the reference material for the various elements used for details on what these elements are or what they do, these are listed in the example code below but will not be explained here (the basic interoperation model does not require any of them, they are provided to add some interactivity to the sample).

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WPF_Hosting_Win32_Control.HostWindow"
    Name="mainWindow"
    Loaded="On_UIReady">

    <DockPanel Background="LightGreen">
        <Border Name="ControlHostElement"
            Width="200"
            Height="200"
            HorizontalAlignment="Right"
            VerticalAlignment="Top"
            BorderBrush="LightGray"
            BorderThickness="3"
            DockPanel.Dock="Right"/>
        <StackPanel>
            <Label HorizontalAlignment="Center"
                Margin="0,10,0,0"
                FontSize="14"
                FontWeight="Bold">Control the Control</Label>
            <TextBlock Margin="10,10,10,10" >Selected Text: <TextBlock Name="selectedText"/></TextBlock>
            <TextBlock Margin="10,10,10,10" >Number of Items: <TextBlock Name="numItems"/></TextBlock>

            <Line X1="0" X2="200"
                Stroke="LightYellow"
                StrokeThickness="2"
                HorizontalAlignment="Center"
                Margin="0,20,0,0"/>

            <Label HorizontalAlignment="Center"
                Margin="10,10,10,10">Append an Item to the List</Label>
            <StackPanel Orientation="Horizontal">
                <Label HorizontalAlignment="Left"
                    Margin="10,10,10,10">Item Text</Label>
                <TextBox HorizontalAlignment="Left"
                    Name="txtAppend"
                    Width="200"
                    Margin="10,10,10,10"></TextBox>
            </StackPanel>

            <Button HorizontalAlignment="Left"
                Click="AppendText"
                Width="75"
                Margin="10,10,10,10">Append</Button>

            <Line X1="0" X2="200"
                Stroke="LightYellow"
                StrokeThickness="2"
                HorizontalAlignment="Center"
                Margin="0,20,0,0"/>

            <Label HorizontalAlignment="Center"
                Margin="10,10,10,10">Delete the Selected Item</Label>

            <Button Click="DeleteText"
                Width="125"
                Margin="10,10,10,10"
                HorizontalAlignment="Left">Delete</Button>
        </StackPanel>
    </DockPanel>
</Window>

```

## Implement a Class to Host the Microsoft Win32 Control

The core of this sample is the class that actually hosts the control, ControlHost.cs. It inherits from [HwndHost](#). The constructor takes two parameters, height and width, which correspond to the height and width of the [Border](#) element that hosts the ListBox control. These values are used later to ensure that the size of the control matches the [Border](#) element.

```
public class ControlHost : HwndHost
{
    IntPtr hwndControl;
    IntPtr hwndHost;
    int hostHeight, hostWidth;

    public ControlHost(double height, double width)
    {
        hostHeight = (int)height;
        hostWidth = (int)width;
    }
}
```

```
Public Class ControlHost
    Inherits HwndHost
    Private hwndControl As IntPtr
    Private hwndHost As IntPtr
    Private hostHeight, hostWidth As Integer

    Public Sub New(ByVal height As Double, ByVal width As Double)
        hostHeight = CInt(height)
        hostWidth = CInt(width)
    End Sub
```

There is also a set of constants. These constants are largely taken from Winuser.h, and allow you to use conventional names when calling Win32 functions.

```
internal const int
WS_CHILD = 0x40000000,
WS_VISIBLE = 0x10000000,
LBS_NOTIFY = 0x00000001,
HOST_ID = 0x00000002,
LISTBOX_ID = 0x00000001,
WS_VSCROLL = 0x00200000,
WS_BORDER = 0x00800000;
```

```
Friend Const WS_CHILD As Integer = &H40000000, WS_VISIBLE As Integer = &H10000000, LBS_NOTIFY As Integer =
&H00000001, HOST_ID As Integer = &H00000002, LISTBOX_ID As Integer = &H00000001, WS_VSCROLL As Integer =
&H00200000, WS_BORDER As Integer = &H00800000
```

## Override [BuildWindowCore](#) to Create the Microsoft Win32 Window

You override this method to create the Win32 window that will be hosted by the page, and make the connection between the window and the page. Because this sample involves hosting a ListBox Control, two windows are created. The first is the window that is actually hosted by the WPF page. The ListBox Control is created as a child of that window.

The reason for this approach is to simplify the process of receiving notifications from the control. The [HwndHost](#) class allows you to process messages sent to the window that it is hosting. If you host a Win32 control directly, you receive the messages sent to the internal message loop of the control. You can display the control and send it messages, but you do not receive the notifications that the control sends to its parent window. This means, among other things, that you have no way of detecting when the user interacts with the control. Instead, create a host window and make the control a child of that window. This allows you to process the messages for the host

window including the notifications sent to it by the control. For convenience, since the host window is little more than a simple wrapper for the control, the package will be referred to as a ListBox control.

#### Create the Host Window and ListBox Control

You can use PInvoke to create a host window for the control by creating and registering a window class, and so on. However, a much simpler approach is to create a window with the predefined "static" window class. This provides you with the window procedure you need in order to receive notifications from the control, and requires minimal coding.

The HWND of the control is exposed through a read-only property, such that the host page can use it to send messages to the control.

```
public IntPtr hwndListBox
{
    get { return hwndControl; }
}
```

```
Public ReadOnly Property hwndListBox() As IntPtr
    Get
        Return hwndControl
    End Get
End Property
```

The ListBox control is created as a child of the host window. The height and width of both windows are set to the values passed to the constructor, discussed above. This ensures that the size of the host window and control is identical to the reserved area on the page. After the windows are created, the sample returns a [HandleRef](#) object that contains the HWND of the host window.

```
protected override HandleRef BuildWindowCore(HandleRef hwndParent)
{
    hwndControl = IntPtr.Zero;
    hwndHost = IntPtr.Zero;

    hwndHost = CreateWindowEx(0, "static", "",
                            WS_CHILD | WS_VISIBLE,
                            0, 0,
                            hostWidth, hostHeight,
                            hwndParent.Handle,
                            (IntPtr)HOST_ID,
                            IntPtr.Zero,
                            0);

    hwndControl = CreateWindowEx(0, "listbox", "",
                                WS_CHILD | WS_VISIBLE | LBS_NOTIFY
                                | WS_VSCROLL | WS_BORDER,
                                0, 0,
                                hostWidth, hostHeight,
                                hwndHost,
                                (IntPtr) LISTBOX_ID,
                                IntPtr.Zero,
                                0);

    return new HandleRef(this, hwndHost);
}
```

```

Protected Overrides Function BuildWindowCore(ByVal hwndParent As HandleRef) As HandleRef
    hwndControl = IntPtr.Zero
    hwndHost = IntPtr.Zero

    hwndHost = CreateWindowEx(0, "static", "", WS_CHILD Or WS_VISIBLE, 0, 0, hostWidth, hostHeight,
    hwndParent.Handle, New IntPtr(HOST_ID), IntPtr.Zero, 0)

    hwndControl = CreateWindowEx(0, "listbox", "", WS_CHILD Or WS_VISIBLE Or LBS_NOTIFY Or WS_VSCROLL Or
    WS_BORDER, 0, 0, hostWidth, hostHeight, hwndHost, New IntPtr(LISTBOX_ID), IntPtr.Zero, 0)

    Return New HandleRef(Me, hwndHost)
End Function

```

```

//PInvoke declarations
[DllImport("user32.dll", EntryPoint = "CreateWindowEx", CharSet = CharSet.Unicode)]
internal static extern IntPtr CreateWindowEx(int dwExStyle,
   string lpszClassName,
   string lpszWindowName,
   int style,
   int x, int y,
   int width, int height,
   IntPtr hwndParent,
   IntPtr hMenu,
   IntPtr hInst,
   [MarshalAs(UnmanagedType.AsAny)] object pvParam);

```

```

'PInvoke declarations
<DllImport("user32.dll", EntryPoint := "CreateWindowEx", CharSet := CharSet.Unicode)>
Friend Shared Function CreateWindowEx(ByVal dwExStyle As Integer, ByVal lpszClassName As String, ByVal
lpszWindowName As String, ByVal style As Integer, ByVal x As Integer, ByVal y As Integer, ByVal width As
Integer, ByVal height As Integer, ByVal hwndParent As IntPtr, ByVal hMenu As IntPtr, ByVal hInst As IntPtr,
<MarshalAs(UnmanagedType.AsAny)> ByVal pvParam As Object) As IntPtr
End Function

```

## Implement DestroyWindow and WndProc

In addition to [BuildWindowCore](#), you must also override the [WndProc](#) and [DestroyWindowCore](#) methods of the [HwndHost](#). In this example, the messages for the control are handled by the [MessageHook](#) handler, thus the implementation of [WndProc](#) and [DestroyWindowCore](#) is minimal. In the case of [WndProc](#), set `handled` to `false` to indicate that the message was not handled and return 0. For [DestroyWindowCore](#), simply destroy the window.

```

protected override IntPtr WndProc(IntPtr hwnd, int msg, IntPtr wParam, IntPtr lParam, ref bool handled)
{
    handled = false;
    return IntPtr.Zero;
}

protected override void DestroyWindowCore(HandleRef hwnd)
{
    DestroyWindow(hwnd.Handle);
}

```

```

Protected Overrides Function WndProc(ByVal hwnd As IntPtr, ByVal msg As Integer, ByVal wParam As IntPtr, ByVal lParam As IntPtr, ByRef handled As Boolean) As IntPtr
    handled = False
    Return IntPtr.Zero
End Function

Protected Overrides Sub DestroyWindowCore(ByVal hwnd As HandleRef)
    DestroyWindow(hwnd.Handle)
End Sub

```

```
[DllImport("user32.dll", EntryPoint = "DestroyWindow", CharSet = CharSet.Unicode)]
internal static extern bool DestroyWindow(IntPtr hwnd);
```

```
<DllImport("user32.dll", EntryPoint := "DestroyWindow", CharSet := CharSet.Unicode)>
Friend Shared Function DestroyWindow(ByVal hwnd As IntPtr) As Boolean
End Function
```

## Host the Control on the Page

To host the control on the page, you first create a new instance of the `ControlHost` class. Pass the height and width of the border element that contains the control (`ControlHostElement`) to the `controlHost` constructor. This ensures that the `ListBox` is sized correctly. You then host the control on the page by assigning the `ControlHost` object to the `Child` property of the host `Border`.

The sample attaches a handler to the `MessageHook` event of the `ControlHost` to receive messages from the control. This event is raised for every message sent to the hosted window. In this case, these are the messages sent to window that wraps the actual `ListBox` control, including notifications from the control. The sample calls `SendMessage` to obtain information from the control and modify its contents. The details of how the page communicates with the control are discussed in the next section.

### NOTE

Notice that there are two PInvoke declarations for `SendMessage`. This is necessary because one uses the `wParam` parameter to pass a string and the other uses it to pass an integer. You need a separate declaration for each signature to ensure that the data is marshaled correctly.

```

public partial class HostWindow : Window
{
    int selectedItem;
    IntPtr hwndListBox;
    ControlHost listControl;
    Application app;
    Window myWindow;
    int itemCount;

    private void On_UIReady(object sender, EventArgs e)
    {
        app = System.Windows.Application.Current;
        myWindow = app.MainWindow;
        myWindow.SizeToContent = SizeToContent.WidthAndHeight;
        listControl = new ControlHost(ControlHostElement.ActualHeight, ControlHostElement.ActualWidth);
        ControlHostElement.Child = listControl;
        listControl.MessageHook += new HwndSourceHook(ControlMsgFilter);
        hwndListBox = listControl(hwndListBox);
        for (int i = 0; i < 15; i++) //populate listbox
        {
            string itemText = "Item" + i.ToString();
            SendMessage(hwndListBox, LB_ADDSTRING, IntPtr.Zero, itemText);
        }
        itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero, IntPtr.Zero);
        numItems.Text = "" + itemCount.ToString();
    }
}

```

```

Partial Public Class HostWindow
    Inherits Window
    Private selectedItem As Integer
    Private hwndListBox As IntPtr
    Private listControl As ControlHost
    Private app As Application
    Private myWindow As Window
    Private itemCount As Integer

    Private Sub On_UIReady(ByVal sender As Object, ByVal e As EventArgs)
        app = System.Windows.Application.Current
        myWindow = app.MainWindow
        myWindow.SizeToContent = SizeToContent.WidthAndHeight
        listControl = New ControlHost(ControlHostElement.ActualHeight, ControlHostElement.ActualWidth)
        ControlHostElement.Child = listControl
        AddHandler listControl.MessageHook, AddressOf ControlMsgFilter
        hwndListBox = listControl(hwndListBox)
        For i As Integer = 0 To 14 'populate listbox
            Dim itemText As String = "Item" & i.ToString()
            SendMessage(hwndListBox, LB_ADDSTRING, IntPtr.Zero, itemText)
        Next i
        itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero, IntPtr.Zero)
        numItems.Text = "" & itemCount.ToString()
    End Sub

```



```

Private Function ControlMsgFilter(ByVal hwnd As IntPtr, ByVal msg As Integer, ByVal wParam As IntPtr, ByVal lParam As IntPtr, ByRef handled As Boolean) As IntPtr
    Dim textLength As Integer

    handled = False
    If msg = WM_COMMAND Then
        Select Case CUInt(wParam.ToInt32()) >> 16 And &HFFFF 'extract the HIWORD
            Case LBN_SELCHANGE 'Get the item text and display it
                selectedItem = SendMessage(listControl.hwndListBox, LB_GETCURSEL, IntPtr.Zero, IntPtr.Zero)
                textLength = SendMessage(listControl.hwndListBox, LB_GETTEXTLEN, IntPtr.Zero, IntPtr.Zero)
                Dim itemText As New StringBuilder()
                SendMessage(hwndListBox, LB_GETTEXT, selectedItem, itemText)
                selectedText.Text = itemText.ToString()
                handled = True
        End Select
    End If
    Return IntPtr.Zero
End Function

Friend Const LBN_SELCHANGE As Integer = &H1, WM_COMMAND As Integer = &H111, LB_GETCURSEL As Integer = &H188,
LB_GETTEXTLEN As Integer = &H18A, LB_ADDSTRING As Integer = &H180, LB_GETTEXT As Integer = &H189,
LB_DELETESTRING As Integer = &H182, LB_GETCOUNT As Integer = &H18B

<DllImport("user32.dll", EntryPoint:="SendMessage", CharSet:=CharSet.Unicode)>
Friend Shared Function SendMessage(ByVal hwnd As IntPtr, ByVal msg As Integer, ByVal wParam As IntPtr, ByVal lParam As IntPtr) As Integer
End Function

<DllImport("user32.dll", EntryPoint:="SendMessage", CharSet:=CharSet.Unicode)>
Friend Shared Function SendMessage(ByVal hwnd As IntPtr, ByVal msg As Integer, ByVal wParam As Integer,
<MarshalAs(UnmanagedType.LPWStr)> ByVal lParam As StringBuilder) As Integer
End Function

<DllImport("user32.dll", EntryPoint:="SendMessage", CharSet:=CharSet.Unicode)>
Friend Shared Function SendMessage(ByVal hwnd As IntPtr, ByVal msg As Integer, ByVal wParam As IntPtr, ByVal lParam As String) As IntPtr
End Function

```

## Implement Communication Between the Control and the Page

You manipulate the control by sending it Windows messages. The control notifies you when the user interacts with it by sending notifications to its host window. The [Hosting a Win32 ListBox Control in WPF](#) sample includes a UI that provides several examples of how this works:

- Append an item to the list.
- Delete the selected item from the list
- Display the text of the currently selected item.
- Display the number of items in the list.

The user can also select an item in the list box by clicking on it, just as they would for a conventional Win32 application. The displayed data is updated each time the user changes the state of the list box by selecting, adding, or appending an item.

To append items, send the list box an [LB\\_ADDSTRING message](#). To delete items, send [LB\\_GETCURSEL](#) to get the index of the current selection and then [LB\\_DELETESTRING](#) to delete the item. The sample also sends [LB\\_GETCOUNT](#), and uses the returned value to update the display that shows the number of items. Both these instances of [SendMessage](#) use one of the PInvoke declarations discussed in the previous section.

```

private void AppendText(object sender, EventArgs args)
{
    if (!string.IsNullOrEmpty(txtAppend.Text))
    {
        SendMessage(hwndListBox, LB_ADDSTRING, IntPtr.Zero, txtAppend.Text);
    }
    itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero, IntPtr.Zero);
    numItems.Text = "" + itemCount.ToString();
}
private void DeleteText(object sender, EventArgs args)
{
    selectedItem = SendMessage(listControl(hwndListBox, LB_GETCURSEL, IntPtr.Zero, IntPtr.Zero);
    if (selectedItem != -1) //check for selected item
    {
        SendMessage(hwndListBox, LB_DELETESTRING, (IntPtr)selectedItem, IntPtr.Zero);
    }
    itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero, IntPtr.Zero);
    numItems.Text = "" + itemCount.ToString();
}

```

```

Private Sub AppendText(ByVal sender As Object, ByVal args As EventArgs)
    If txtAppend.Text <> String.Empty Then
        SendMessage(hwndListBox, LB_ADDSTRING, IntPtr.Zero, txtAppend.Text)
    End If
    itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero, IntPtr.Zero)
    numItems.Text = "" & itemCount.ToString()
End Sub
Private Sub DeleteText(ByVal sender As Object, ByVal args As EventArgs)
    selectedItem = SendMessage(listControl(hwndListBox, LB_GETCURSEL, IntPtr.Zero, IntPtr.Zero)
    If selectedItem <> -1 Then 'check for selected item
        SendMessage(hwndListBox, LB_DELETESTRING, New IntPtr(selectedItem), IntPtr.Zero)
    End If
    itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero, IntPtr.Zero)
    numItems.Text = "" & itemCount.ToString()
End Sub

```

When the user selects an item or changes their selection, the control notifies the host window by sending it a [WM\\_COMMAND message](#), which raises the [MessageHook](#) event for the page. The handler receives the same information as the main window procedure of the host window. It also passes a reference to a Boolean value, [handled](#). You set [handled](#) to [true](#) to indicate that you have handled the message and no further processing is needed.

[WM\\_COMMAND](#) is sent for a variety of reasons, so you must examine the notification ID to determine whether it is an event that you wish to handle. The ID is contained in the high word of the [wParam](#) parameter. The sample uses bitwise operators to extract the ID. If the user has made or changed their selection, the ID will be [LBN\\_SELCHANGE](#).

When [LBN\\_SELCHANGE](#) is received, the sample gets the index of the selected item by sending the control a [LB\\_GETCURSEL message](#). To get the text, you first create a [StringBuilder](#). You then send the control an [LB\\_GETTEXT message](#). Pass the empty [StringBuilder](#) object as the [wParam](#) parameter. When [SendMessage](#) returns, the [StringBuilder](#) will contain the text of the selected item. This use of [SendMessage](#) requires yet another PInvoke declaration.

Finally, set [handled](#) to [true](#) to indicate that the message has been handled.

## See also

- [HwndHost](#)
- [WPF and Win32 Interoperation](#)
- [Walkthrough: My first WPF desktop application](#)



# Walkthrough: Hosting WPF Content in Win32

10/24/2019 • 15 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides a rich environment for creating applications. However, when you have a substantial investment in Win32 code, it might be more effective to add WPF functionality to your application rather than rewriting your original code. WPF provides a straightforward mechanism for hosting WPF content in a Win32 window.

This tutorial describes how to write a sample application, [Hosting WPF Content in a Win32 Window Sample](#), that hosts WPF content in a Win32 window. You can extend this sample to host any Win32 window. Because it involves mixing managed and unmanaged code, the application is written in C++/CLI.

## Requirements

This tutorial assumes a basic familiarity with both WPF and Win32 programming. For a basic introduction to WPF programming, see [Getting Started](#). For an introduction to Win32 programming, you should reference any of the numerous books on the subject, in particular *Programming Windows* by Charles Petzold.

Because the sample that accompanies this tutorial is implemented in C++/CLI, this tutorial assumes familiarity with the use of C++ to program the Windows API plus an understanding of managed code programming. Familiarity with C++/CLI is helpful but not essential.

### NOTE

This tutorial includes a number of code examples from the associated sample. However, for readability, it does not include the complete sample code. For the complete sample code, see [Hosting WPF Content in a Win32 Window Sample](#).

## The Basic Procedure

This section outlines the basic procedure you use to host WPF content in a Win32 window. The remaining sections explain the details of each step.

The key to hosting WPF content on a Win32 window is the [HwndSource](#) class. This class wraps the WPF content in a Win32 window, allowing it to be incorporated into your user interface (UI) as a child window. The following approach combines the Win32 and WPF in a single application.

1. Implement your WPF content as a managed class.
2. Implement a Windows application with C++/CLI. If you are starting with an existing application and unmanaged C++ code, you can usually enable it to call managed code by changing your project settings to include the `/clr` compiler flag.
3. Set the threading model to single-threaded apartment (STA).
4. Handle the [WM\\_CREATE](#) notification in your window procedure and do the following:
  - a. Create a new [HwndSource](#) object with the parent window as its `parent` parameter.
  - b. Create an instance of your WPF content class.
  - c. Assign a reference to the WPF content object to the [RootVisual](#) property of the [HwndSource](#).
  - d. Get the HWND for the content. The [Handle](#) property of the [HwndSource](#) object contains the

window handle (HWND). To get an HWND that you can use in the unmanaged part of your application, cast `Handle.ToPointer()` to an HWND.

5. Implement a managed class that contains a static field to hold a reference to your WPF content. This class allows you to get a reference to the WPF content from your Win32 code.
6. Assign the WPF content to the static field.
7. Receive notifications from the WPF content by attaching a handler to one or more of the WPF events.
8. Communicate with the WPF content by using the reference that you stored in the static field to set properties, and so on.

#### NOTE

You can also use Extensible Application Markup Language (XAML) to implement your WPF content. However, you will have to compile it separately as a dynamic-link library (DLL) and reference that DLL from your Win32 application. The remainder of the procedure is similar to that outlined above.

## Implementing the Host Application

This section describes how to host WPF content in a basic Win32 application. The content itself is implemented in C++/CLI as a managed class. For the most part, it is straightforward WPF programming. The key aspects of the content implementation are discussed in [Implementing the WPF Content](#).

- [The Basic Application](#)
- [Hosting the WPF Content](#)
- [Holding a Reference to the WPF Content](#)
- [Communicating with the WPF Content](#)

### The Basic Application

The starting point for the host application was to create a Visual Studio 2005 template.

1. Open Visual Studio 2005, and select **New Project** from the **File** menu.
2. Select **Win32** from the list of Visual C++ project types. If your default language is not C++, you will find these project types under **Other Languages**.
3. Select a **Win32 Project** template, assign a name to the project and click **OK** to launch the **Win32 Application Wizard**.
4. Accept the wizard's default settings and click **Finish** to start the project.

The template creates a basic Win32 application, including:

- An entry point for the application.
- A window, with an associated window procedure (WndProc).
- A menu with **File** and **Help** headings. The **File** menu has an **Exit** item that closes the application. The **Help** menu has an **About** item that launches a simple dialog box.

Before you start writing code to host the WPF content, you need to make two modifications to the basic template.

The first is to compile the project as managed code. By default, the project compiles as unmanaged code. However, because WPF is implemented in managed code, the project must be compiled accordingly.

1. Right-click the project name in **Solution Explorer** and select **Properties** from the context menu to launch the **Property Pages** dialog box.
2. Select **Configuration Properties** from the tree view in the left pane.
3. Select **Common Language Runtime** support from the **Project Defaults** list in the right pane.
4. Select **Common Language Runtime Support (/clr)** from the drop-down list box.

**NOTE**

This compiler flag allows you to use managed code in your application, but your unmanaged code will still compile as before.

WPF uses the single-threaded apartment (STA) threading model. In order to work properly with the WPF content code, you must set the application's threading model to STA by applying an attribute to the entry point.

```
[System::STAThreadAttribute] //Needs to be an STA thread to play nicely with WPF
int APIENTRY _tWinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPTSTR    lpCmdLine,
                      int       nCmdShow)
{
```

### Hosting the WPF Content

The WPF content is a simple address entry application. It consists of several [TextBox](#) controls to take user name, address, and so on. There are also two [Button](#) controls, **OK** and **Cancel**. When the user clicks **OK**, the button's [Click](#) event handler collects the data from the [TextBox](#) controls, assigns it to corresponding properties, and raises a custom event, [OnButtonClicked](#). When the user clicks **Cancel**, the handler simply raises [OnButtonClicked](#). The event argument object for [OnButtonClicked](#) contains a Boolean field that indicates which button was clicked.

The code to host the WPF content is implemented in a handler for the [WM\\_CREATE](#) notification on the host window.

```
case WM_CREATE :
    GetClientRect(hWnd, &rect);
    wpfHwnd = GetHwnd(hWnd, rect.right-375, 0, 375, 250);
    CreateDataDisplay(hWnd, 275, rect.right-375, 375);
    CreateRadioButtons(hWnd);
    break;
```

The [GetHwnd](#) method takes size and position information plus the parent window handle and returns the window handle of the hosted WPF content.

**NOTE**

You cannot use a [#using](#) directive for the [System::Windows::Interop](#) namespace. Doing so creates a name collision between the [MSG](#) structure in that namespace and the [MSG](#) structure declared in [winuser.h](#). You must instead use fully-qualified names to access the contents of that namespace.

```

HWND GetHwnd(HWND parent, int x, int y, int width, int height)
{
    System::Windows::Interop::HwndSourceParameters^ sourceParams = gcnew
    System::Windows::Interop::HwndSourceParameters(
        "hi" // NAME
    );
    sourceParams->PositionX = x;
    sourceParams->PositionY = y;
    sourceParams->Height = height;
    sourceParams->Width = width;
    sourceParams->ParentWindow = IntPtr(parent);
    sourceParams->WindowStyle = WS_VISIBLE | WS_CHILD; // style
    System::Windows::Interop::HwndSource^ source = gcnew System::Windows::Interop::HwndSource(*sourceParams);
    WPFPAGE ^myPage = gcnew WPFPAGE(width, height);
    //Assign a reference to the WPF page and a set of UI properties to a set of static properties in a class
    //that is designed for that purpose.
    WPFPAGEHOST::hostedPage = myPage;
    WPFPAGEHOST::initBackBrush = myPage->Background;
    WPFPAGEHOST::initFontFamily = myPage->DefaultFontFamily;
    WPFPAGEHOST::initFontSize = myPage->DefaultFontSize;
    WPFPAGEHOST::initFontStyle = myPage->DefaultFontStyle;
    WPFPAGEHOST::initFontWeight = myPage->DefaultFontWeight;
    WPFPAGEHOST::initForeBrush = myPage->DefaultForeBrush;
    myPage->OnButtonClicked += gcnew WPFPAGE::ButtonClickHandler(WPFPAGEHOST::OnButtonClicked);
    source->RootVisual = myPage;
    return (HWND) source->Handle.ToPointer();
}

```

You cannot host the WPF content directly in your application window. Instead, you first create an [HwndSource](#) object to wrap the WPF content. This object is basically a window that is designed to host a WPF content. You host the [HwndSource](#) object in the parent window by creating it as a child of a Win32 window that is part of your application. The [HwndSource](#) constructor parameters contain much the same information that you would pass to `CreateWindow` when you create a Win32 child window.

You next create an instance of the WPF content object. In this case, the WPF content is implemented as a separate class, [WPFPAGE](#), using C++/CLI. You could also implement the WPF content with XAML. However, to do so you need to set up a separate project and build the WPF content as a DLL. You can add a reference to that DLL to your project, and use that reference to create an instance of the WPF content.

You display the WPF content in your child window by assigning a reference to the WPF content to the [RootVisual](#) property of the [HwndSource](#).

The next line of code attaches an event handler, [WPFPAGEHOST::OnButtonClicked](#), to the WPF content [OnButtonClicked](#) event. This handler is called when the user clicks the **OK** or **Cancel** button. See [communicating\\_with\\_the\\_WPF\\_content](#) for further discussion of this event handler.

The final line of code shown returns the window handle (HWND) that is associated with the [HwndSource](#) object. You can use this handle from your Win32 code to send messages to the hosted window, although the sample does not do so. The [HwndSource](#) object raises an event every time it receives a message. To process the messages, call the [AddHook](#) method to attach a message handler and then process the messages in that handler.

## Holding a Reference to the WPF Content

For many applications, you will want to communicate with the WPF content later. For example, you might want to modify the WPF content properties, or perhaps have the [HwndSource](#) object host different WPF content. To do this, you need a reference to the [HwndSource](#) object or the WPF content. The [HwndSource](#) object and its associated WPF content remain in memory until you destroy the window handle. However, the variable you assign to the [HwndSource](#) object will go out of scope as soon as you return from the window procedure. The customary way to handle this issue with Win32 applications is to use a static or global variable. Unfortunately, you cannot assign a managed object to those types of variables. You can assign the window handle associated with

`HwndSource` object to a global or static variable, but that does not provide access to the object itself.

The simplest solution to this issue is to implement a managed class that contains a set of static fields to hold references to any managed objects that you need access to. The sample uses the `WPFPAGEHOST` class to hold a reference to the WPF content, plus the initial values of a number of its properties that might be changed later by the user. This is defined in the header.

```
public ref class WPFPAGEHOST
{
public:
    WPFPAGEHOST();
    static WPFPAGE^ hostedPage;
    //initial property settings
    static System::Windows::Media::Brush^ initBackBrush;
    static System::Windows::Media::Brush^ initForeBrush;
    static System::Windows::Media::FontFamily^ initFontFamily;
    static System::Windows::FontStyle initFontStyle;
    static System::Windows::FontWeight initFontWeight;
    static double initFontSize;
};
```

The latter part of the `GetHwnd` function assigns values to those fields for later use while `myPage` is still in scope.

### Communicating with the WPF Content

There are two types of communication with the WPF content. The application receives information from the WPF content when the user clicks the **OK** or **Cancel** buttons. The application also has a UI that allows the user to change various WPF content properties, such as the background color or default font size.

As mentioned above, when the user clicks either button the WPF content raises an `OnButtonClicked` event. The application attaches a handler to this event to receive these notifications. If the **OK** button was clicked, the handler gets the user information from the WPF content and displays it in a set of static controls.

```

void WPFPButtonClicked(Object ^sender, MyPageEventArgs ^args)
{
    if(args->IsOK) //display data if OK button was clicked
    {
        WPFPPage ^myPage = WPFPPageHost::hostedPage;
        LPCWSTR userName = (LPCWSTR) InteropServices::Marshal::StringToHGlobalAuto("Name: " + myPage-
>EnteredName).ToPointer();
        SetWindowText(nameLabel, userName);
        LPCWSTR userAddress = (LPCWSTR) InteropServices::Marshal::StringToHGlobalAuto("Address: " + myPage-
>EnteredAddress).ToPointer();
        SetWindowText(addressLabel, userAddress);
        LPCWSTR userCity = (LPCWSTR) InteropServices::Marshal::StringToHGlobalAuto("City: " + myPage-
>EnteredCity).ToPointer();
        SetWindowText(cityLabel, userCity);
        LPCWSTR userState = (LPCWSTR) InteropServices::Marshal::StringToHGlobalAuto("State: " + myPage-
>EnteredState).ToPointer();
        SetWindowText(stateLabel, userState);
        LPCWSTR userZip = (LPCWSTR) InteropServices::Marshal::StringToHGlobalAuto("Zip: " + myPage-
>EnteredZip).ToPointer();
        SetWindowText(zipLabel, userZip);
    }
    else
    {
        SetWindowText(nameLabel, L"Name: ");
        SetWindowText(addressLabel, L"Address: ");
        SetWindowText(cityLabel, L"City: ");
        SetWindowText(stateLabel, L"State: ");
        SetWindowText(zipLabel, L"Zip: ");
    }
}

```

The handler receives a custom event argument object from the WPF content, `MyPageEventArgs`. The object's `IsOK` property is set to `true` if the **OK** button was clicked, and `false` if the **Cancel** button was clicked.

If the **OK** button was clicked, the handler gets a reference to the WPF content from the container class. It then collects the user information that is held by the associated WPF content properties and uses the static controls to display the information on the parent window. Because the WPF content data is in the form of a managed string, it has to be marshaled for use by a Win32 control. If the **Cancel** button was clicked, the handler clears the data from the static controls.

The application UI provides a set of radio buttons that allow the user to modify the background color of the WPF content, and several font-related properties. The following example is an excerpt from the application's window procedure (WndProc) and its message handling that sets various properties on different messages, including the background color. The others are similar, and are not shown. See the complete sample for details and context.

```

case WM_COMMAND:
    wMid      = LOWORD(wParam);
    wParamEvent = HIWORD(wParam);

    switch (wMid)
    {
        //Menu selections
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        //RadioButtons
        case IDC_ORIGINALBACKGROUND :
            WPFPAGEHOST::hostedPage->Background = WPFPAGEHOST::initBackBrush;
            break;
        case IDC_LIGHTGREENBACKGROUND :
            WPFPAGEHOST::hostedPage->Background = gcnew SolidColorBrush(Colors::LightGreen);
            break;
        case IDC_LIGHTSALMONBACKGROUND :
            WPFPAGEHOST::hostedPage->Background = gcnew SolidColorBrush(Colors::LightSalmon);
            break;
    }
}

```

To set the background color, get a reference to the WPF content (`hostedPage`) from `WPFPAGEHOST` and set the background color property to the appropriate color. The sample uses three color options: the original color, light green, or light salmon. The original background color is stored as a static field in the `WPFPAGEHOST` class. To set the other two, you create a new `SolidColorBrush` object and pass the constructor a static colors value from the `Colors` object.

## Implementing the WPF Page

You can host and use the WPF content without any knowledge of the actual implementation. If the WPF content had been packaged in a separate DLL, it could have been built in any common language runtime (CLR) language. Following is a brief walkthrough of the C++/CLI implementation that is used in the sample. This section contains the following subsections.

- [Layout](#)
- [Returning the Data to the Host Window](#)
- [Setting the WPF Properties](#)

### Layout

The UI elements in the WPF content consist of five `TextBox` controls, with associated `Label` controls: Name, Address, City, State, and Zip. There are also two `Button` controls, **OK** and **Cancel**.

The WPF content is implemented in the `WPFPAGE` class. Layout is handled with a `Grid` layout element. The class inherits from `Grid`, which effectively makes it the WPF content root element.

The WPF content constructor takes the required width and height, and sizes the `Grid` accordingly. It then defines the basic layout by creating a set of `ColumnDefinition` and `RowDefinition` objects and adding them to the `Grid` object base `ColumnDefinitions` and `RowDefinitions` collections, respectively. This defines a grid of five rows and seven columns, with the dimensions determined by the contents of the cells.

```

WPFPPage::WPFPPage(int allottedWidth, int allottedHeight)
{
    array<ColumnDefinition ^> ^ columnDef = gcnew array<ColumnDefinition ^> (4);
    array<RowDefinition ^> ^ rowDef = gcnew array<RowDefinition ^> (6);

    this->Height = allottedHeight;
    this->Width = allottedWidth;
    this->Background = gcnew SolidColorBrush(Colors::LightGray);

    //Set up the Grid's row and column definitions
    for(int i=0; i<4; i++)
    {
        columnDef[i] = gcnew ColumnDefinition();
        columnDef[i]->Width = GridLength(1, GridUnitType::Auto);
        this->ColumnDefinitions->Add(columnDef[i]);
    }
    for(int i=0; i<6; i++)
    {
        rowDef[i] = gcnew RowDefinition();
        rowDef[i]->Height = GridLength(1, GridUnitType::Auto);
        this->RowDefinitions->Add(rowDef[i]);
    }
}

```

Next, the constructor adds the UI elements to the [Grid](#). The first element is the title text, which is a [Label](#) control that is centered in the first row of the grid.

```

//Add the title
titleText = gcnew Label();
titleText->Content = "Simple WPF Control";
titleText->HorizontalAlignment = System::Windows::HorizontalAlignment::Center;
titleText->Margin = Thickness(10, 5, 10, 0);
titleText->FontWeight = FontWeights::Bold;
titleText->FontSize = 14;
Grid::SetColumn(titleText, 0);
Grid::SetRow(titleText, 0);
Grid::SetColumnSpan(titleText, 4);
this->Children->Add(titleText);

```

The next row contains the Name [Label](#) control and its associated [TextBox](#) control. Because the same code is used for each label/textbox pair, it is placed in a pair of private methods and used for all five label/textbox pairs. The methods create the appropriate control, and call the [Grid](#) class static [SetColumn](#) and [SetRow](#) methods to place the controls in the appropriate cell. After the control is created, the sample calls the [Add](#) method on the [Children](#) property of the [Grid](#) to add the control to the grid. The code to add the remaining label/textbox pairs is similar. See the sample code for details.

```

//Add the Name Label and TextBox
nameLabel = CreateLabel(0, 1, "Name");
this->Children->Add(nameLabel);
nameTextBox = CreateTextBox(1, 1, 3);
this->Children->Add(nameTextBox);

```

The implementation of the two methods is as follows:

```

Label ^WPFPage::CreateLabel(int column, int row, String ^ text)
{
    Label ^ newLabel = gcnew Label();
    newLabel->Content = text;
    newLabel->Margin = Thickness(10, 5, 10, 0);
    newLabel->FontWeight = FontWeights::Normal;
    newLabel->FontSize = 12;
    Grid::SetColumn(newLabel, column);
    Grid::SetRow(newLabel, row);
    return newLabel;
}

TextBox ^WPFPage::CreateTextBox(int column, int row, int span)
{
    TextBox ^newTextBox = gcnew TextBox();
    newTextBox->Margin = Thickness(10, 5, 10, 0);
    Grid::SetColumn(newTextBox, column);
    Grid::SetRow(newTextBox, row);
    Grid::SetColumnSpan(newTextBox, span);
    return newTextBox;
}

```

Finally, the sample adds the **OK** and **Cancel** buttons and attaches an event handler to their [Click](#) events.

```

//Add the Buttons and attach event handlers
okButton = CreateButton(0, 5, "OK");
cancelButton = CreateButton(1, 5, "Cancel");
this->Children->Add(okButton);
this->Children->Add(cancelButton);
okButton->Click += gcnew RoutedEventHandler(this, &WPFPage::ButtonClicked);
cancelButton->Click += gcnew RoutedEventHandler(this, &WPFPage::ButtonClicked);

```

## Returning the Data to the Host Window

When either button is clicked, its [Click](#) event is raised. The host window could simply attach handlers to these events and get the data directly from the [TextBox](#) controls. The sample uses a somewhat less direct approach. It handles the [Click](#) within the WPF content, and then raises a custom event [OnButtonClicked](#), to notify the WPF content. This allows the WPF content to do some parameter validation before notifying the host. The handler gets the text from the [TextBox](#) controls and assigns it to public properties, from which the host can retrieve the information.

The event declaration, in [WPFPage.h](#):

```

public:
    delegate void ButtonClickHandler(Object ^, MyPageEventArgs ^);
    WPFPage();
    WPFPage(int height, int width);
    event ButtonClickHandler ^OnButtonClicked;

```

The [Click](#) event handler, in [WPFPage.cpp](#):

```

void WPFPAGE::ButtonClicked(Object ^sender, RoutedEventArgs ^args)
{
    //TODO: validate input data
    bool okClicked = true;
    if(sender == cancelButton)
        okClicked = false;
    EnteredName = nameTextBox->Text;
    EnteredAddress = addressTextBox->Text;
    EnteredCity = cityTextBox->Text;
    EnteredState = stateTextBox->Text;
    EnteredZip = zipTextBox->Text;
    OnButtonClicked(this, gcnew MyPageEventArgs(okClicked));
}

```

## Setting the WPF Properties

The Win32 host allows the user to change several WPF content properties. From the Win32 side, it is simply a matter of changing the properties. The implementation in the WPF content class is somewhat more complicated, because there is no single global property that controls the fonts for all controls. Instead, the appropriate property for each control is changed in the properties' set accessors. The following example shows the code for the `DefaultFontFamily` property. Setting the property calls a private method that in turn sets the `FontFamily` properties for the various controls.

From `WPFPAGE.h`:

```

property FontFamily^ DefaultFontFamily
{
    FontFamily^ get() {return _defaultFontFamily;}
    void set(FontFamily^ value) {SetFontFamily(value);}
};

```

From `WPFPAGE.cpp`:

```

void WPFPAGE::SetFontFamily(FontFamily^ newFontFamily)
{
    _defaultFontFamily = newFontFamily;
    titleText->FontFamily = newFontFamily;
    nameLabel->FontFamily = newFontFamily;
    addressLabel->FontFamily = newFontFamily;
    cityLabel->FontFamily = newFontFamily;
    stateLabel->FontFamily = newFontFamily;
    zipLabel->FontFamily = newFontFamily;
}

```

## See also

- [HwndSource](#)
- [WPF and Win32 Interoperation](#)

# Walkthrough: Hosting a WPF Clock in Win32

10/25/2019 • 8 minutes to read • [Edit Online](#)

To put WPF inside Win32 applications, use [HwndSource](#), which provides the HWND that contains your WPF content. First you create the [HwndSource](#), giving it parameters similar to CreateWindow. Then you tell the [HwndSource](#) about the WPF content you want inside it. Finally, you get the HWND out of the [HwndSource](#). This walkthrough illustrates how to create a mixed WPF inside Win32 application that reimplements the operating system **Date and Time Properties** dialog.

## Prerequisites

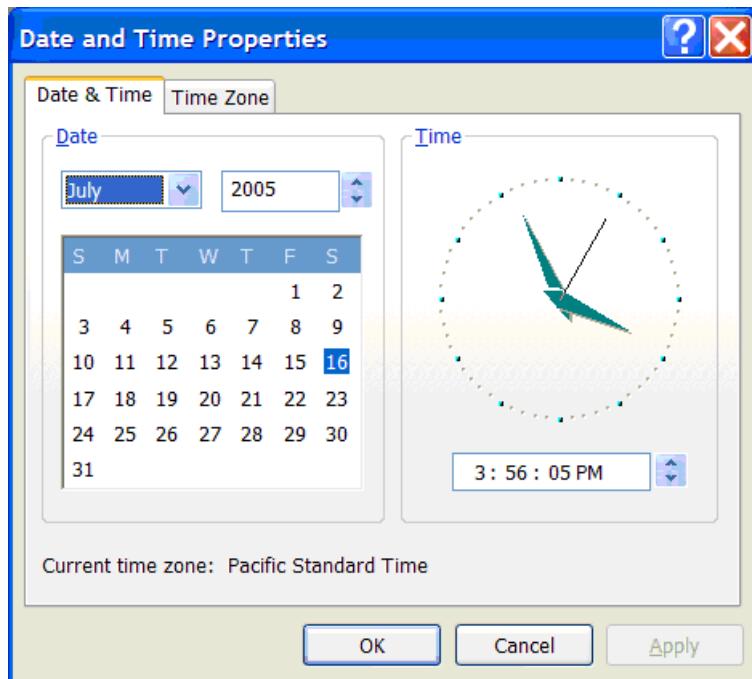
See [WPF and Win32 Interoperation](#).

## How to Use This Tutorial

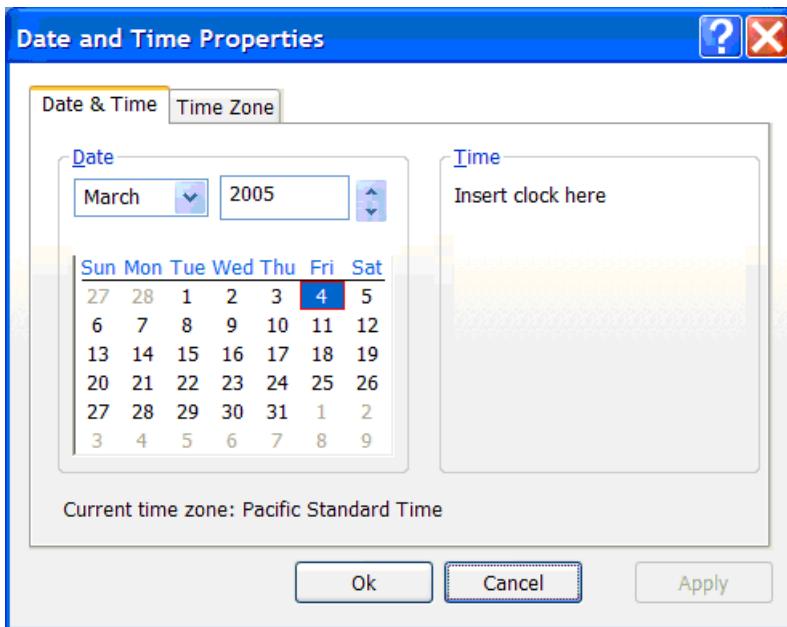
This tutorial concentrates on the important steps of producing an interoperation application. The tutorial is backed by a sample, [Win32 Clock Interoperation Sample](#), but that sample is reflective of the end product. This tutorial documents the steps as if you were starting with an existing Win32 project of your own, perhaps a pre-existing project, and you were adding a hosted WPF to your application. You can compare your end product with [Win32 Clock Interoperation Sample](#).

## A Walkthrough of Windows Presentation Framework Inside Win32 (HwndSource)

The following graphic shows the intended end product of this tutorial:



You can recreate this dialog by creating a C++ Win32 project in Visual Studio, and using the dialog editor to create the following:



(You do not need to use Visual Studio to use [HwndSource](#), and you do not need to use C++ to write Win32 programs, but this is a fairly typical way to do it, and lends itself well to a stepwise tutorial explanation).

You need to accomplish five particular substeps in order to put a WPF clock into the dialog:

1. Enable your Win32 project to call managed code (`/clr`) by changing project settings in Visual Studio.
2. Create a [WPF Page](#) in a separate DLL.
3. Put that [WPF Page](#) inside an [HwndSource](#).
4. Get an HWND for that [Page](#) using the [Handle](#) property.
5. Use Win32 to decide where to place the HWND within the larger Win32 application

## /clr

The first step is to turn this unmanaged Win32 project into one that can call managed code. You use the `/clr` compiler option, which will link to the necessary DLLs you want to use, and adjust the Main method for use with WPF.

To enable the use of managed code inside the C++ project: Right-click on win32clock project and select **Properties**. On the **General** property page (the default), change Common Language Runtime support to `/clr`.

Next, add references to DLLs necessary for WPF: PresentationCore.dll, PresentationFramework.dll, System.dll, WindowsBase.dll, UIAutomationProvider.dll, and UIAutomationTypes.dll. (Following instructions assume the operating system is installed on C: drive.)

1. Right-click win32clock project and select **References...**, and inside that dialog:
2. Right-click win32clock project and select **References....**
3. Click **Add New Reference**, click Browse tab, enter `C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationCore.dll`, and click OK.
4. Repeat for `PresentationFramework.dll`: `C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationFramework.dll`.
5. Repeat for `WindowsBase.dll`: `C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\WindowsBase.dll`.
6. Repeat for `UIAutomationTypes.dll`: `C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\UIAutomationTypes.dll`

Assemblies\Microsoft\Framework\v3.0\UIAutomationTypes.dll.

7. Repeat for UIAutomationProvider.dll: C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\UIAutomationProvider.dll.
8. Click **Add New Reference**, select System.dll, and click **OK**.
9. Click **OK** to exit the win32clock Property Pages for adding references.

Finally, add the `STAThreadAttribute` to the `_tWinMain` method for use with WPF:

```
[System::STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPTSTR    lpCmdLine,
                        int       nCmdShow)
```

This attribute tells the common language runtime (CLR) that when it initializes Component Object Model (COM), it should use a single threaded apartment model (STA), which is necessary for WPF (and Windows Forms).

## Create a Windows Presentation Framework Page

Next, you create a DLL that defines a WPF [Page](#). It's often easiest to create the WPF [Page](#) as a standalone application, and write and debug the WPF portion that way. Once done, that project can be turned into a DLL by right-clicking the project, clicking on **Properties**, going to the Application, and changing Output type to Windows Class Library.

The WPF dll project can then be combined with the Win32 project (one solution that contains two projects) – right-click on the solution, select **Add\Existing Project**.

To use that WPF dll from the Win32 project, you need to add a reference:

1. Right-click win32clock project and select **References....**
2. Click **Add New Reference**.
3. Click the **Projects** tab. Select WPFClock, click OK.
4. Click **OK** to exit the win32clock Property Pages for adding references.

## HwndSource

Next, you use [HwndSource](#) to make the WPF [Page](#) look like an HWND. You add this block of code to a C++ file:

```

namespace ManagedCode
{
    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Media;

    HWND GetHwnd(HWND parent, int x, int y, int width, int height) {
        HwndSource^ source = gcnew HwndSource(
            0, // class style
            WS_VISIBLE | WS_CHILD, // style
            0, // exstyle
            x, y, width, height,
            "hi", // NAME
            IntPtr(parent) // parent window
        );

        UIElement^ page = gcnew WPFClock::Clock();
        source->RootVisual = page;
        return (HWND) source->Handle.ToPointer();
    }
}
}

```

This is a long piece of code that could use some explanation. The first part is various clauses so that you do not need to fully qualify all the calls:

```

namespace ManagedCode
{
    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Media;

```

Then you define a function that creates the WPF content, puts an [HwndSource](#) around it, and returns the HWND:

```

HWND GetHwnd(HWND parent, int x, int y, int width, int height) {

```

First you create an [HwndSource](#), whose parameters are similar to CreateWindow:

```

HwndSource^ source = gcnew HwndSource(
    0, // class style
    WS_VISIBLE | WS_CHILD, // style
    0, // exstyle
    x, y, width, height,
    "hi", // NAME
    IntPtr(parent) // parent window
);

```

Then you create the WPF content class by calling its constructor:

```

UIElement^ page = gcnew WPFClock::Clock();

```

You then connect the page to the [HwndSource](#):

```

source->RootVisual = page;

```

And in the final line, return the HWND for the `HwndSource`:

```
return (HWND) source->Handle.ToPointer();
```

## Positioning the Hwnd

Now that you have an HWND that contains the WPF clock, you need to put that HWND inside the Win32 dialog. If you knew just where to put the HWND, you would just pass that size and location to the `GetHwnd` function you defined earlier. But you used a resource file to define the dialog, so you are not exactly sure where any of the HWNDs are positioned. You can use the Visual Studio dialog editor to put a Win32 STATIC control where you want the clock to go ("Insert clock here"), and use that to position the WPF clock.

Where you handle WM\_INITDIALOG, you use `GetDlgItem` to retrieve the HWND for the placeholder STATIC:

```
HWND placeholder = GetDlgItem(hDlg, IDC_CLOCK);
```

You then calculate the size and position of that placeholder STATIC, so you can put the WPF clock in that place:

RECT rectangle;

```
GetWindowRect(placeholder, &rectangle);
int width = rectangle.right - rectangle.left;
int height = rectangle.bottom - rectangle.top;
POINT point;
point.x = rectangle.left;
point.y = rectangle.top;
result = MapWindowPoints(NULL, hDlg, &point, 1);
```

Then you hide the placeholder STATIC:

```
ShowWindow(placeholder, SW_HIDE);
```

And create the WPF clock HWND in that location:

```
HWND clock = ManagedCode::GetHwnd(hDlg, point.x, point.y, width, height);
```

To make the tutorial interesting, and to produce a real WPF clock, you will need to create a WPF clock control at this point. You can do so mostly in markup, with just a few event handlers in code-behind. Since this tutorial is about interoperation and not about control design, complete code for the WPF clock is provided here as a code block, without discrete instructions for building it up or what each part means. Feel free to experiment with this code to change the look and feel or functionality of the control.

Here is the markup:

```
<Page x:Class="WPFClock.Clock"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      >
  <Grid>
    <Grid.Background>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="#fcfcfe" Offset="0" />
        <GradientStop Color="#f6f4f0" Offset="1.0" />
      </LinearGradientBrush>
    </Grid.Background>
  </Grid>
</Page>
```

```

<Grid Name="PodClock" HorizontalAlignment="Center" VerticalAlignment="Center">
    <Grid.Resources>
        <Storyboard x:Key="sb">
            <DoubleAnimation From="0" To="360" Duration="12:00:00" RepeatBehavior="Forever"
                Storyboard.TargetName="HourHand"
                Storyboard.TargetProperty="(Rectangle.RenderTransform).(RotateTransform.Angle)"
            />
            <DoubleAnimation From="0" To="360" Duration="01:00:00" RepeatBehavior="Forever"
                Storyboard.TargetName="MinuteHand"
                Storyboard.TargetProperty="(Rectangle.RenderTransform).(RotateTransform.Angle)"
            />
            <DoubleAnimation From="0" To="360" Duration="0:1:00" RepeatBehavior="Forever"
                Storyboard.TargetName="SecondHand"
                Storyboard.TargetProperty="(Rectangle.RenderTransform).(RotateTransform.Angle)"
            />
        </Storyboard>
    </Grid.Resources>

    <Ellipse Width="108" Height="108" StrokeThickness="3">
        <Ellipse.Stroke>
            <LinearGradientBrush>
                <GradientStop Color="LightBlue" Offset="0" />
                <GradientStop Color="DarkBlue" Offset="1" />
            </LinearGradientBrush>
        </Ellipse.Stroke>
    </Ellipse>
    <Ellipse VerticalAlignment="Center" HorizontalAlignment="Center" Width="104" Height="104"
        Fill="LightBlue" StrokeThickness="3">
        <Ellipse.Stroke>
            <LinearGradientBrush>
                <GradientStop Color="DarkBlue" Offset="0" />
                <GradientStop Color="LightBlue" Offset="1" />
            </LinearGradientBrush>
        </Ellipse.Stroke>
    </Ellipse>
    <Border BorderThickness="1" BorderBrush="Black" Background="White" Margin="20"
        HorizontalAlignment="Right" VerticalAlignment="Center">
        <TextBlock Name="MonthDay" Text="{Binding}"/>
    </Border>
    <Canvas Width="102" Height="102">
        <Ellipse Width="8" Height="8" Fill="Black" Canvas.Top="46" Canvas.Left="46" />
        <Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black" Width="4" Height="8">
            <Rectangle.RenderTransform>
                <RotateTransform CenterX="2" CenterY="46" Angle="0" />
            </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
            <Rectangle.RenderTransform>
                <RotateTransform CenterX="2" CenterY="46" Angle="30" />
            </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
            <Rectangle.RenderTransform>
                <RotateTransform CenterX="2" CenterY="46" Angle="60" />
            </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black" Width="4" Height="8">
            <Rectangle.RenderTransform>
                <RotateTransform CenterX="2" CenterY="46" Angle="90" />
            </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
            <Rectangle.RenderTransform>
                <RotateTransform CenterX="2" CenterY="46" Angle="120" />
            </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
            <Rectangle.RenderTransform>

```

```

        <RotateTransform CenterX="2" CenterY="46" Angle="150" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black" Width="4" Height="8">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46" Angle="180" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46" Angle="210" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46" Angle="240" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black" Width="4" Height="8">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46" Angle="270" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46" Angle="300" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black" Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46" Angle="330" />
    </Rectangle.RenderTransform>
</Rectangle>

<Rectangle x:Name="HourHand" Canvas.Top="21" Canvas.Left="48"
    Fill="Black" Width="4" Height="30">
    <Rectangle.RenderTransform>
        <RotateTransform x:Name="HourHand2" CenterX="2" CenterY="30" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle x:Name="MinuteHand" Canvas.Top="6" Canvas.Left="49"
    Fill="Black" Width="2" Height="45">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="1" CenterY="45" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle x:Name="SecondHand" Canvas.Top="4" Canvas.Left="49"
    Fill="Red" Width="1" Height="47">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="0.5" CenterY="47" />
    </Rectangle.RenderTransform>
</Rectangle>
</Canvas>
</Grid>
</Grid>
</Page>

```

And here is the accompanying code-behind:

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Windows.Threading;

namespace WPFClock
{
    /// <summary>
    /// Interaction logic for Clock.xaml
    /// </summary>
    public partial class Clock : Page
    {
        private DispatcherTimer _dayTimer;

        public Clock()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(Clock_Loaded);
        }

        void Clock_Loaded(object sender, RoutedEventArgs e) {
            // set the datacontext to be today's date
            DateTime now = DateTime.Now;
            DataContext = now.Day.ToString();

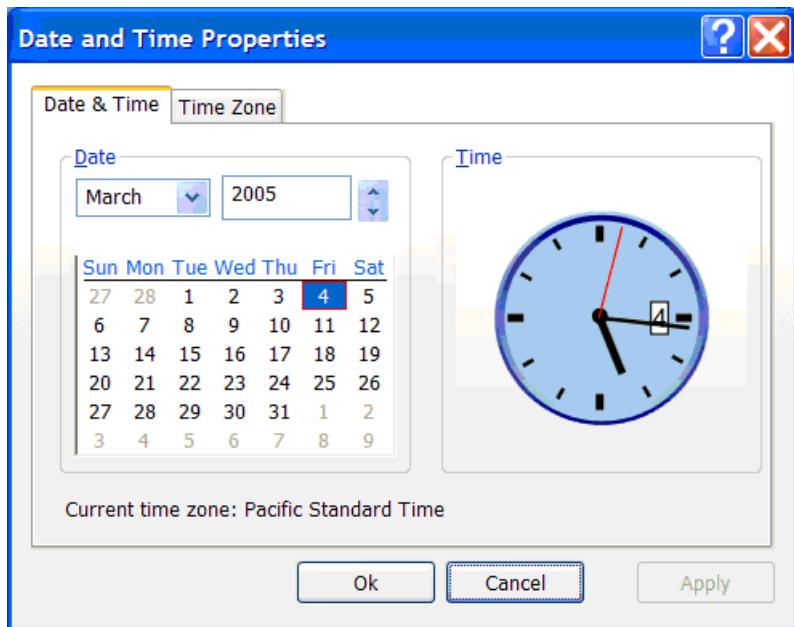
            // then set up a timer to fire at the start of tomorrow, so that we can update
            // the datacontext
            _dayTimer = new DispatcherTimer();
            _dayTimer.Interval = new TimeSpan(1, 0, 0, 0) - now.TimeOfDay;
            _dayTimer.Tick += new EventHandler(OnDayChange);
            _dayTimer.Start();

            // finally, seek the timeline, which assumes a beginning at midnight, to the appropriate
            // offset
            Storyboard sb = (Storyboard)PodClock.FindResource("sb");
            sb.Begin(PodClock, HandoffBehavior.SnapshotAndReplace, true);
            sb.Seek(PodClock, now.TimeOfDay, TimeSeekOrigin.BeginTime);
        }

        private void OnDayChange(object sender, EventArgs e)
        {
            // date has changed, update the datacontext to reflect today's date
            DateTime now = DateTime.Now;
            DataContext = now.Day.ToString();
            _dayTimer.Interval = new TimeSpan(1, 0, 0, 0);
        }
    }
}

```

The final result looks like:



To compare your end result to the code that produced this screenshot, see [Win32 Clock Interoperation Sample](#).

## See also

- [HwndSource](#)
- [WPF and Win32 Interoperation](#)
- [Win32 Clock Interoperation Sample](#)

# WPF and Direct3D9 Interoperation

8/22/2019 • 9 minutes to read • [Edit Online](#)

You can include Direct3D9 content in a Windows Presentation Foundation (WPF) application. This topic describes how to create Direct3D9 content so that it efficiently interoperates with WPF.

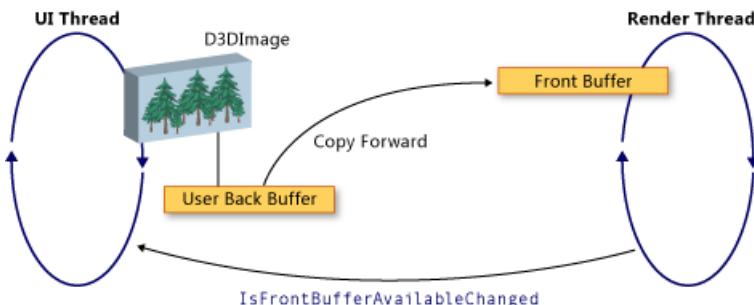
## NOTE

When using Direct3D9 content in WPF, you also need to think about performance. For more information about how to optimize for performance, see [Performance Considerations for Direct3D9 and WPF Interoperability](#).

## Display Buffers

The [D3DImage](#) class manages two display buffers, which are called the *back buffer* and the *front buffer*. The back buffer is your Direct3D9 surface. Changes to the back buffer are copied forward to the front buffer when you call the [Unlock](#) method.

The following illustration shows the relationship between the back buffer and the front buffer.



## Direct3D9 Device Creation

To render Direct3D9 content, you must create a Direct3D9 device. There are two Direct3D9 objects that you can use to create a device, [IDirect3D9](#) and [IDirect3D9Ex](#). Use these objects to create [IDirect3DDevice9](#) and [IDirect3DDevice9Ex](#) devices, respectively.

Create a device by calling one of the following methods.

- [IDirect3D9 \\* Direct3DCreate9\(UINT SDKVersion\);](#)
- [HRESULT Direct3DCreate9Ex\(UINT SDKVersion, IDirect3D9Ex \\*\\*ppD3D\);](#)

On Windows Vista or later operating system, use the [Direct3DCreate9Ex](#) method with a display that is configured to use the Windows Display Driver Model (WDDM). Use the [Direct3DCreate9](#) method on any other platform.

### Availability of the Direct3DCreate9Ex method

The d3d9.dll has the [Direct3DCreate9Ex](#) method only on Windows Vista or later operating system. If you directly link the function on Windows XP, your application fails to load. To determine whether the [Direct3DCreate9Ex](#) method is supported, load the DLL and look for the proc address. The following code shows how to test for the [Direct3DCreate9Ex](#) method. For a full code example, see [Walkthrough: Creating Direct3D9 Content for Hosting in WPF](#).

```

HRESULT
CRendererManager::EnsureD3DObjects()
{
    HRESULT hr = S_OK;

    HMODULE hD3D = NULL;
    if (!m_pD3D)
    {
        hD3D = LoadLibrary(TEXT("d3d9.dll"));
        DIRECT3DCREATE9EXFUNCTION pfnCreate9Ex = (DIRECT3DCREATE9EXFUNCTION)GetProcAddress(hD3D,
"Direct3DCreate9Ex");
        if (pfnCreate9Ex)
        {
            IFC((*pfnCreate9Ex)(D3D_SDK_VERSION, &m_pD3DEx));
            IFC(m_pD3DEx->QueryInterface(__uuidof(IDirect3D9), reinterpret_cast<void **>(&m_pD3D)));
        }
        else
        {
            m_pD3D = Direct3DCreate9(D3D_SDK_VERSION);
            if (!m_pD3D)
            {
                IFC(E_FAIL);
            }
        }
    }

    m_cAdapters = m_pD3D->GetAdapterCount();
}

Cleanup:
if (hD3D)
{
    FreeLibrary(hD3D);
}

return hr;
}

```

## HWND Creation

Creating a device requires an HWND. In general, you create a dummy HWND for Direct3D9 to use. The following code example shows how to create a dummy HWND.

```

HRESULT
CRendererManager::EnsureHWND()
{
    HRESULT hr = S_OK;

    if (!m_hwnd)
    {
        WNDCLASS wndclass;

        wndclass.style = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc = DefWindowProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hInstance = NULL;
        wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName = NULL;
        wndclass.lpszClassName = szAppName;

        if (!RegisterClass(&wndclass))
        {
            IFC(E_FAIL);
        }

        m_hwnd = CreateWindow(szAppName,
                              TEXT("D3DImageSample"),
                              WS_OVERLAPPEDWINDOW,
                              0, // Initial X
                              0, // Initial Y
                              0, // Width
                              0, // Height
                              NULL,
                              NULL,
                              NULL,
                              NULL);
    }

    Cleanup:
    return hr;
}

```

## Present Parameters

Creating a device also requires a `D3DPRESENT_PARAMETERS` struct, but only a few parameters are important. These parameters are chosen to minimize the memory footprint.

Set the `BackBufferHeight` and `BackBufferWidth` fields to 1. Setting them to 0 causes them to be set to the dimensions of the HWND.

Always set the `D3DCREATE_MULTITHREADED` and `D3DCREATE_FPU_PRESERVE` flags to prevent corrupting memory used by Direct3D9 and to prevent Direct3D9 from changing FPU settings.

The following code shows how to initialize the `D3DPRESENT_PARAMETERS` struct.

```

HRESULT
CRenderers::Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEX, HWND hwnd, UINT uAdapter)
{
    HRESULT hr = S_OK;

    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
    d3dpp.BackBufferHeight = 1;
    d3dpp.BackBufferWidth = 1;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;

    D3DCAPS9 caps;
    DWORD dwVertexProcessing;
    IFC(pD3D->GetDeviceCaps(uAdapter, D3DDEVTYPE_HAL, &caps));
    if ((caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT) == D3DDEVCAPS_HWTRANSFORMANDLIGHT)
    {
        dwVertexProcessing = D3DCREATE_HARDWARE_VERTEXPROCESSING;
    }
    else
    {
        dwVertexProcessing = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
    }

    if (pD3DEX)
    {
        IDirect3DDevice9Ex *pd3dDevice = NULL;
        IFC(pD3DEX->CreateDeviceEx(
            uAdapter,
            D3DDEVTYPE_HAL,
            hwnd,
            dwVertexProcessing | D3DCREATE_MULTITHREADED | D3DCREATE_FPU_PRESERVE,
            &d3dpp,
            NULL,
            &m_pd3dDeviceEx
        ));

        IFC(m_pd3dDeviceEx->QueryInterface(__uuidof(IDirect3DDevice9), reinterpret_cast<void**>
        (&m_pd3dDevice)));
    }
    else
    {
        assert(pD3D);

        IFC(pD3D->CreateDevice(
            uAdapter,
            D3DDEVTYPE_HAL,
            hwnd,
            dwVertexProcessing | D3DCREATE_MULTITHREADED | D3DCREATE_FPU_PRESERVE,
            &d3dpp,
            &m_pd3dDevice
        ));
    }

    Cleanup:
    return hr;
}

```

## Creating the Back Buffer Render Target

To display Direct3D9 content in a [D3DImage](#), you create a Direct3D9 surface and assign it by calling the [SetBackBuffer](#) method.

### Verifying Adapter Support

Before creating a surface, verify that all adapters support the surface properties you require. Even if you render to only one adapter, the WPF window may be displayed on any adapter in the system. You should always write Direct3D9 code that handles multi-adapter configurations, and you should check all adapters for support, because WPF might move the surface among the available adapters.

The following code example shows how to check all adapters on the system for Direct3D9 support.

```

HRESULT
CRendererManager::TestSurfaceSettings()
{
    HRESULT hr = S_OK;

    D3DFORMAT fmt = m_fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8;

    //
    // We test all adapters because we potentially use all adapters.
    // But even if this sample only rendered to the default adapter, you
    // should check all adapters because WPF may move your surface to
    // another adapter for you!
    //

    for (UINT i = 0; i < m_cAdapters; ++i)
    {
        // Can we get HW rendering?
        IFC(m_pD3D->CheckDeviceType(
            i,
            D3DDEVTYPE_HAL,
            D3DFMT_X8R8G8B8,
            fmt,
            TRUE
        ));

        // Is the format okay?
        IFC(m_pD3D->CheckDeviceFormat(
            i,
            D3DDEVTYPE_HAL,
            D3DFMT_X8R8G8B8,
            D3DUSAGE_RENDERTARGET | D3DUSAGE_DYNAMIC, // We'll use dynamic when on XP
            D3DRTYPE_SURFACE,
            fmt
        ));

        // D3DImage only allows multisampling on 9Ex devices. If we can't
        // multisample, overwrite the desired number of samples with 0.
        if (m_pD3DEx && m_uNumSamples > 1)
        {
            assert(m_uNumSamples <= 16);

            if (FAILED(m_pD3D->CheckDeviceMultiSampleType(
                i,
                D3DDEVTYPE_HAL,
                fmt,
                TRUE,
                static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),
                NULL
            )))
            {
                m_uNumSamples = 0;
            }
        }
        else
        {
            m_uNumSamples = 0;
        }
    }

    Cleanup:
    return hr;
}

```

## Creating the Surface

Before creating a surface, verify that the device capabilities support good performance on the target operating

system. For more information, see [Performance Considerations for Direct3D9 and WPF Interoperability](#).

When you have verified device capabilities, you can create the surface. The following code example shows how to create the render target.

```
HRESULT
CRenderer::CreateSurface(UINT uWidth, UINT uHeight, bool fUseAlpha, UINT m_uNumSamples)
{
    HRESULT hr = S_OK;

    SAFE_RELEASE(m_pd3dRTS);

    IFC(m_pd3dDevice->CreateRenderTarget(
        uWidth,
        uHeight,
        fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8,
        static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),
        0,
        m_pd3dDeviceEx ? FALSE : TRUE, // Lockable RT required for good XP perf
        &m_pd3dRTS,
        NULL
    ));

    IFC(m_pd3dDevice->SetRenderTarget(0, m_pd3dRTS));

    Cleanup:
    return hr;
}
```

## WDDM

On Windows Vista and later operating systems, which are configured to use the WDDM, you can create a render target texture and pass the level 0 surface to the [SetBackBuffer](#) method. This approach is not recommended on Windows XP, because you cannot create a lockable render target texture and performance will be reduced.

## Handling Device State

The [D3DImage](#) class manages two display buffers, which are called the *back buffer* and the *front buffer*. The back buffer is your Direct3D surface. Changes to the back buffer are copied forward to the front buffer when you call the [Unlock](#) method, where it is displayed on the hardware. Occasionally, the front buffer becomes unavailable. This lack of availability can be caused by screen locking, full-screen exclusive Direct3D applications, user-switching, or other system activities. When this occurs, your WPF application is notified by handling the [IsFrontBufferAvailableChanged](#) event. How your application responds to the front buffer becoming unavailable depends on whether WPF is enabled to fall back to software rendering. The [SetBackBuffer](#) method has an overload that takes a parameter that specifies whether WPF falls back to software rendering.

When you call the [SetBackBuffer\(D3DResourceType, IntPtr\)](#) overload or call the [SetBackBuffer\(D3DResourceType, IntPtr, Boolean\)](#) overload with the `enableSoftwareFallback` parameter set to `false`, the rendering system releases its reference to the back buffer when the front buffer becomes unavailable and nothing is displayed. When the front buffer is available again, the rendering system raises the [IsFrontBufferAvailableChanged](#) event to notify your WPF application. You can create an event handler for the [IsFrontBufferAvailableChanged](#) event to restart rendering again with a valid Direct3D surface. To restart rendering, you must call [SetBackBuffer](#).

When you call the [SetBackBuffer\(D3DResourceType, IntPtr, Boolean\)](#) overload with the `enableSoftwareFallback` parameter set to `true`, the rendering system retains its reference to the back buffer when the front buffer becomes unavailable, so there is no need to call [SetBackBuffer](#) when the front buffer is available again.

When software rendering is enabled, there may be situations where the user's device becomes unavailable, but

the rendering system retains a reference to the Direct3D surface. To check whether a Direct3D9 device is unavailable, call the `TestCooperativeLevel` method. To check a Direct3D9Ex devices call the `CheckDeviceState` method, because the `TestCooperativeLevel` method is deprecated and always returns success. If the user device has become unavailable, call `SetBackBuffer` to release WPF's reference to the back buffer. If you need to reset your device, call `SetBackBuffer` with the `backBuffer` parameter set to `null`, and then call `SetBackBuffer` again with `backBuffer` set to a valid Direct3D surface.

Call the `Reset` method to recover from an invalid device only if you implement multi-adapter support. Otherwise, release all Direct3D9 interfaces and re-create them completely. If the adapter layout has changed, Direct3D9 objects created before the change are not updated.

## Handling Resizing

If a `D3DImage` is displayed at a resolution other than its native size, it is scaled according to the current `BitmapScalingMode`, except that `Bilinear` is substituted for `Fant`.

If you require higher fidelity, you must create a new surface when the container of the `D3DImage` changes size.

There are three possible approaches to handle resizing.

- Participate in the layout system and create a new surface when the size changes. Do not create too many surfaces, because you may exhaust or fragment video memory.
- Wait until a resize event has not occurred for a fixed period of time to create the new surface.
- Create a `DispatcherTimer` that checks the container dimensions several times per second.

## Multi-monitor Optimization

Significantly reduced performance can result when the rendering system moves a `D3DImage` to another monitor.

On WDDM, as long as the monitors are on the same video card and you use `Direct3DCreate9Ex`, there is no reduction in performance. If the monitors are on separate video cards, performance is reduced. On Windows XP, performance is always reduced.

When the `D3DImage` moves to another monitor, you can create a new surface on the corresponding adapter to restore good performance.

To avoid the performance penalty, write code specifically for the multi-monitor case. The following list shows one way to write multi-monitor code.

1. Find a point of the `D3DImage` in screen space with the `Visual.ProjectToScreen` method.
2. Use the `MonitorFromPoint` GDI method to find the monitor that is displaying the point.
3. Use the `IDirect3D9::GetAdapterMonitor` method to find which Direct3D9 adapter the monitor is on.
4. If the adapter is not the same as the adapter with the back buffer, create a new back buffer on the new monitor and assign it to the `D3DImage` back buffer.

### NOTE

If the `D3DImage` straddles monitors, performance will be slow, except in the case of WDDM and `IDirect3D9Ex` on the same adapter. There is no way to improve performance in this situation.

The following code example shows how to find the current monitor.

```

void
CRendererManager::SetAdapter(POINT screenSpacePoint)
{
    CleanupInvalidDevices();

    //
    // After CleanupInvalidDevices, we may not have any D3D objects. Rather than
    // recreate them here, ignore the adapter update and wait for render to recreate.
    //

    if (m_pD3D && m_rgRenderers)
    {
        HMONITOR hMon = MonitorFromPoint(screenSpacePoint, MONITOR_DEFAULTTONULL);

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            if (hMon == m_pD3D->GetAdapterMonitor(i))
            {
                m_pCurrentRenderer = m_rgRenderers[i];
                break;
            }
        }
    }
}

```

Update the monitor when the [D3DImage](#) container's size or position changes, or update the monitor by using a [DispatcherTimer](#) that updates a few times per second.

## WPF Software Rendering

WPF renders synchronously on the UI thread in software in the following situations.

- Printing
- [BitmapEffect](#)
- [RenderTargetBitmap](#)

When one of these situations occurs, the rendering system calls the [CopyBackBuffer](#) method to copy the hardware buffer to software. The default implementation calls the [GetRenderTargetData](#) method with your surface. Because this call occurs outside of the Lock/Unlock pattern, it may fail. In this case, the [CopyBackBuffer](#) method returns [null](#) and no image is displayed.

You can override the [CopyBackBuffer](#) method, call the base implementation, and if it returns [null](#), you can return a placeholder [BitmapSource](#).

You can also implement your own software rendering instead of calling the base implementation.

### NOTE

If WPF is rendering completely in software, [D3DImage](#) is not shown because WPF does not have a front buffer.

## See also

- [D3DImage](#)
- [Performance Considerations for Direct3D9 and WPF Interoperability](#)
- [Walkthrough: Creating Direct3D9 Content for Hosting in WPF](#)
- [Walkthrough: Hosting Direct3D9 Content in WPF](#)

# Performance Considerations for Direct3D9 and WPF Interoperability

8/22/2019 • 4 minutes to read • [Edit Online](#)

You can host Direct3D9 content by using the [D3DImage](#) class. Hosting Direct3D9 content can affect the performance of your application. This topic describes best practices to optimize performance when hosting Direct3D9 content in a Windows Presentation Foundation (WPF) application. These best practices include how to use [D3DImage](#) and best practices when you are using Windows Vista, Windows XP, and multi-monitor displays.

## NOTE

For code examples that demonstrate these best practices, see [WPF and Direct3D9 Interoperation](#).

## Use D3DImage Sparingly

Direct3D9 content hosted in a [D3DImage](#) instance does not render as fast as in a pure Direct3D application. Copying the surface and flushing the command buffer can be costly operations. As the number of [D3DImage](#) instances increases, more flushing occurs, and performance degrades. Therefore, you should use [D3DImage](#) sparingly.

## Best Practices on Windows Vista

For best performance on Windows Vista with a display that is configured to use the Windows Display Driver Model (WDDM), create your Direct3D9 surface on an [IDirect3DDevice9Ex](#) device. This enables surface sharing. The video card must support the [D3DDEVCAPS2\\_CAN\\_STRETCHRECT\\_FROM\\_TEXTURES](#) and [D3DCAPS2\\_CANSHARERESOURCE](#) driver capabilities on Windows Vista. Any other settings cause the surface to be copied through software, which reduces performance significantly.

## NOTE

If Windows Vista has a display that is configured to use the Windows XP Display Driver Model (XDDM), the surface is always copied through software, regardless of settings. With the proper settings and video card, you will see better performance on Windows Vista when you use the WDDM because surface copies are performed in hardware.

## Best Practices on Windows XP

For best performance on Windows XP, which uses the Windows XP Display Driver Model (XDDM), create a lockable surface that behaves correctly when the [IDirect3DSurface9::GetDC](#) method is called. Internally, the [BitBlt](#) method transfers the surface across devices in hardware. The [GetDC](#) method always works on XRGB surfaces. However, if the client computer is running Windows XP with SP3 or SP2, and if the client also has the hotfix for the layered-window feature, this method only works on ARGB surfaces. The video card must support the [D3DDEVCAPS2\\_CAN\\_STRETCHRECT\\_FROM\\_TEXTURES](#) driver capability.

A 16-bit desktop display depth can significantly reduce performance. A 32-bit desktop is recommended.

If you are developing for Windows Vista and Windows XP, test the performance on Windows XP. Running out of video memory on Windows XP is a concern. In addition, [D3DImage](#) on Windows XP uses more video memory and bandwidth than Windows Vista WDDM, due to a necessary extra video memory copy. Therefore, you can

expect performance to be worse on Windows XP than on Windows Vista for the same video hardware.

#### NOTE

XDDM is available on both Windows XP and Windows Vista; however, WDDM is available only on Windows Vista.

## General Best Practices

When you create the device, use the `D3DCREATE_MULTITHREADED` creation flag. This reduces performance, but the WPF rendering system calls methods on this device from another thread. Be sure to follow the locking protocol correctly, so that no two threads access the device at the same time.

If your rendering is performed on a WPF managed thread, it is strongly recommended that you create the device with the `D3DCREATE_FPU_PRESERVE` creation flag. Without this setting, the D3D rendering can reduce the accuracy of WPF double-precision operations and introduce rendering issues.

Tiling a [D3DImage](#) is fast, unless you tile a non-pow2 surface without hardware support, or if you tile a [DrawingBrush](#) or [VisualBrush](#) that contains a [D3DImage](#).

## Best Practices for Multi-Monitor Displays

If you are using a computer that has multiple monitors, you should follow the previously described best practices. There are also some additional performance considerations for a multi-monitor configuration.

When you create the back buffer, it is created on a specific device and adapter, but WPF may display the front buffer on any adapter. Copying across adapters to update the front buffer can be very expensive. On Windows Vista that is configured to use the WDDM with multiple video cards and with an `IDirect3DDevice9Ex` device, if the front buffer is on a different adapter but still the same video card, there is no performance penalty. However, on Windows XP and the XDDM with multiple video cards, there is a significant performance penalty when the front buffer is displayed on a different adapter than the back buffer. For more information, see [WPF and Direct3D Interoperation](#).

## Performance Summary

The following table shows performance of the front buffer update as a function of operating system, pixel format, and surface lockability. The front buffer and back buffer are assumed to be on the same adapter. Depending on the adapter configuration, hardware updates are generally much faster than software updates.

SURFACE PIXEL FORMAT	WINDOWS VISTA, WDDM AND 9EX	OTHER WINDOWS VISTA CONFIGURATIONS	WINDOWS XP SP3 OR SP2 W/ HOTFIX	WINDOWS XP SP2
D3DFMT_X8R8G8B8 (not lockable)	<b>Hardware Update</b>	Software Update	Software Update	Software Update
D3DFMT_X8R8G8B8 (lockable)	<b>Hardware Update</b>	Software Update	<b>Hardware Update</b>	<b>Hardware Update</b>
D3DFMT_A8R8G8B8 (not lockable)	<b>Hardware Update</b>	Software Update	Software Update	Software Update
D3DFMT_A8R8G8B8 (lockable)	<b>Hardware Update</b>	Software Update	<b>Hardware Update</b>	Software Update

## See also

- [D3DImage](#)
- [WPF and Direct3D9 Interoperation](#)
- [Walkthrough: Creating Direct3D9 Content for Hosting in WPF](#)
- [Walkthrough: Hosting Direct3D9 Content in WPF](#)

# Walkthrough: Creating Direct3D9 Content for Hosting in WPF

9/14/2019 • 13 minutes to read • [Edit Online](#)

This walkthrough shows how to create Direct3D9 content that is suitable for hosting in a Windows Presentation Foundation (WPF) application. For more information on hosting Direct3D9 content in WPF applications, see [WPF and Direct3D9 Interoperation](#).

In this walkthrough, you perform the following tasks:

- Create a Direct3D9 project.
- Configure the Direct3D9 project for hosting in a WPF application.

When you are finished, you will have a DLL that contains Direct3D9 content for use in a WPF application.

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio 2010.
- DirectX SDK 9 or later.

## Creating the Direct3D9 Project

The first step is to create and configure the Direct3D9 project.

### To create the Direct3D9 project

1. Create a new Win32 Project in C++ named `D3DContent`.

The Win32 Application Wizard opens and displays the Welcome screen.

2. Click **Next**.

The Application Settings screen appears.

3. In the **Application type:** section, select the **DLL** option.

4. Click **Finish**.

The D3DContent project is generated.

5. In Solution Explorer, right-click the D3DContent project and select **Properties**.

The **D3DContent Property Pages** dialog box opens.

6. Select the **C/C++** node.

7. In the **Additional Include Directories** field, specify the location of the DirectX include folder. The default location for this folder is %ProgramFiles%\Microsoft DirectX SDK (version)\Include.

8. Double-click the **Linker** node to expand it.

9. In the **Additional Library Directories** field, specify the location of the DirectX libraries folder. The default location for this folder is %ProgramFiles%\Microsoft DirectX SDK (version)\Lib\x86.

10. Select the **Input** node.
11. In the **Additional Dependencies** field, add the `d3d9.lib` and `d3dx9.lib` files.
12. In Solution Explorer, add a new module definition file (.def) named `D3DContent.def` to the project.

## Creating the Direct3D9 Content

To get the best performance, your Direct3D9 content must use particular settings. The following code shows how to create a Direct3D9 surface that has the best performance characteristics. For more information, see [Performance Considerations for Direct3D9 and WPF Interoperability](#).

### To create the Direct3D9 content

1. Using Solution Explorer, add three C++ classes to the project named the following.

```
CRenderer (with virtual destructor)

CRendererManager

CTriangleRenderer
```

2. Open Renderer.h in the Code Editor and replace the automatically generated code with the following code.

```
#pragma once

class CRenderer
{
public:
    virtual ~CRenderer();

    HRESULT CheckDeviceState();
    HRESULT CreateSurface(UINT uWidth, UINT uHeight, bool fUseAlpha, UINT m_uNumSamples);

    virtual HRESULT Render() = 0;

    IDirect3DSurface9 *GetSurfaceNoRef() { return m_pd3dRTS; }

protected:
    CRenderer();

    virtual HRESULT Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND hwnd, UINT uAdapter);

    IDirect3DDevice9    *m_pd3dDevice;
    IDirect3DDevice9Ex *m_pd3dDeviceEx;

    IDirect3DSurface9 *m_pd3dRTS;

};
```

3. Open Renderer.cpp in the Code Editor and replace the automatically generated code with the following code.

```
//-----
// CRenderer
//
// An abstract base class that creates a device and a target render
// surface. Derive from this class and override Init() and Render()
// to do your own rendering. See CTriangleRenderer for an example.

//-----
```

```

#include "StdAfx.h"

//+-----
// 
// Member:
//     CRenderer ctor
//
//-----
CRenderer::CRenderer() : m_pd3dDevice(NULL), m_pd3dDeviceEx(NULL), m_pd3dRTS(NULL)
{
}

//+-----
// 
// Member:
//     CRenderer dtor
//
//-----
CRenderer::~CRenderer()
{
    SAFE_RELEASE(m_pd3dDevice);
    SAFE_RELEASE(m_pd3dDeviceEx);
    SAFE_RELEASE(m_pd3dRTS);
}

//+-----
// 
// Member:
//     CRenderer::CheckDeviceState
//
// Synopsis:
//     Returns the status of the device. 9Ex devices are a special case because
//     TestCooperativeLevel() has been deprecated in 9Ex.
//
//-----
HRESULT
CRenderer::CheckDeviceState()
{
    if (m_pd3dDeviceEx)
    {
        return m_pd3dDeviceEx->CheckDeviceState(NULL);
    }
    else if (m_pd3dDevice)
    {
        return m_pd3dDevice->TestCooperativeLevel();
    }
    else
    {
        return D3DERR_DEVICELOST;
    }
}

//+-----
// 
// Member:
//     CRenderer::CreateSurface
//
// Synopsis:
//     Creates and sets the render target
//
//-----
HRESULT
CRenderer::CreateSurface(UINT uWidth, UINT uHeight, bool fUseAlpha, UINT m_uNumSamples)
{
    HRESULT hr = S_OK;

    SAFE_RELEASE(m_pd3dRTS);
}

```

```

IFC(m_pd3dDevice->CreateRenderTarget(
    uWidth,
    uHeight,
    fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8,
    static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),
    0,
    m_pd3dDeviceEx ? FALSE : TRUE, // Lockable RT required for good XP perf
    &m_pd3dRTS,
    NULL
));

IFC(m_pd3dDevice->SetRenderTarget(0, m_pd3dRTS));

Cleanup:
    return hr;
}

//+-----+
// 
// Member:
//     CRenderer::Init
//
// Synopsis:
//     Creates the device
//
//-----+
HRESULT
CRenderer::Init(IDirect3D9 *pD3D, IDirect3DEx *pD3DEx, HWND hwnd, UINT uAdapter)
{
    HRESULT hr = S_OK;

    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
    d3dpp.BackBufferHeight = 1;
    d3dpp.BackBufferWidth = 1;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;

    D3DCAPS9 caps;
    DWORD dwVertexProcessing;
    IFC(pD3D->GetDeviceCaps(uAdapter, D3DDEVCAPS_HAL, &caps));
    if ((caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT) == D3DDEVCAPS_HWTRANSFORMANDLIGHT)
    {
        dwVertexProcessing = D3DCREATE_HARDWARE_VERTEXPROCESSING;
    }
    else
    {
        dwVertexProcessing = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
    }

    if (pD3DEx)
    {
        IDirect3DDevice9Ex *pd3dDevice = NULL;
        IFC(pD3DEx->CreateDeviceEx(
            uAdapter,
            D3DDEVTYPE_HAL,
            hwnd,
            dwVertexProcessing | D3DCREATE_MULTITHREADED | D3DCREATE_FPU_PRESERVE,
            &d3dpp,
            NULL,
            &m_pd3dDeviceEx
        ));

        IFC(m_pd3dDeviceEx->QueryInterface(__uuidof(IDirect3DDevice9), reinterpret_cast<void**>
        (&m_pd3dDevice)));
    }
    else
    {

```

```
assert(pD3D);

IFC(pD3D->CreateDevice(
    uAdapter,
    D3DDEVTYPE_HAL,
    hwnd,
    dwVertexProcessing | D3DCREATE_MULTITHREADED | D3DCREATE_FPU_PRESERVE,
    &d3dpp,
    &m_pd3dDevice
));
}

Cleanup:
    return hr;
}
```

4. Open RendererManager.h in the Code Editor and replace the automatically generated code with the following code.

```

#pragma once

class CRenderer;

class CRendererManager
{
public:
    static HRESULT Create(CRendererManager **ppManager);
    ~CRendererManager();

    HRESULT EnsureDevices();

    void SetSize(UINT uWidth, UINT uHeight);
    void SetAlpha(bool fUseAlpha);
    void SetNumDesiredSamples(UINT uNumSamples);
    void SetAdapter(POINT screenSpacePoint);

    HRESULT GetBackBufferNoRef(IDirect3DSurface9 **ppSurface);

    HRESULT Render();

private:
    CRendererManager();

    void CleanupInvalidDevices();
    HRESULT EnsureRenderers();
    HRESULT EnsureHWND();
    HRESULT EnsureD3DOBJECTS();
    HRESULT TestSurfaceSettings();
    void DestroyResources();

    IDirect3D      *m_pD3D;
    IDirect3D9Ex   *m_pD3DEx;

    UINT m_cAdapters;
    CRenderer **m_rgRenderers;
    CRenderer *m_pCurrentRenderer;

    HWND m_hwnd;

    UINT m_uWidth;
    UINT m_uHeight;
    UINT m_uNumSamples;
    bool m_fUseAlpha;
    bool m_fSurfaceSettingsChanged;
};


```

5. Open RendererManager.cpp in the Code Editor and replace the automatically generated code with the following code.

```

//-----
//  CRendererManager
//
//      Manages the list of CRenderers. Managed code pinvoke into this class
//      and this class forwards to the appropriate CRenderer.
//
//-----

#include "StdAfx.h"

const static TCHAR szAppName[] = TEXT("D3DImageSample");
typedef HRESULT (WINAPI *DIRECT3DCREATE9EXFUNCTION)(UINT SDKVersion, IDirect3D9Ex**);

```

```

//+-----
// 
//  Member:
//      CRendererManager ctor
// 
//-----
```

- CRendererManager::CRendererManager()
 :
  - `m_pD3D(NULL),`
  - `m_pD3DEX(NULL),`
  - `m_cAdapters(0),`
  - `m_hwnd(NULL),`
  - `m_pCurrentRenderer(NULL),`
  - `m_rgRenderers(NULL),`
  - `m_uWidth(1024),`
  - `m_uHeight(1024),`
  - `m_uNumSamples(0),`
  - `m_fUseAlpha(false),`
  - `m_fSurfaceSettingsChanged(true)`

```

{
```

```

}
```

```

//+-----
// 
//  Member:
//      CRendererManager dtor
// 
//-----
```

- CRendererManager::~CRendererManager()
 :
  - `DestroyResources();`
  - `if (m_hwnd)`
    - {
    - `DestroyWindow(m_hwnd);`
    - `UnregisterClass(szAppName, NULL);`
    - }

```

}
```

```

//+-----
// 
//  Member:
//      CRendererManager::Create
// 
//  Synopsis:
//      Creates the manager
// 
//-----
```

- HRESULT
 CRendererManager::Create(CRendererManager \*\*ppManager)
 :
  - `HRESULT hr = S_OK;`
  - `*ppManager = new CRendererManager();`
  - `IFCOOM(*ppManager);`
  - `Cleanup:`
    - `return hr;`

```

}
```

```

//+-----
// 
//  Member:
//      CRendererManager::EnsureRenderers
// 
//  Synopsis:
//      Makes sure the CRenderer objects exist
//-----
```

```

/*
//-----
HRESULT
CRendererManager::EnsureRenderers()
{
    HRESULT hr = S_OK;

    if (!m_rgRenderers)
    {
        IFC(EnsureHWND());

        assert(m_cAdapters);
        m_rgRenderers = new CRenderer*[m_cAdapters];
        IFCOM(m_rgRenderers);
        ZeroMemory(m_rgRenderers, m_cAdapters * sizeof(m_rgRenderers[0]));

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            IFC(CTriangleRenderer::Create(m_pD3D, m_pD3DEX, m_hwnd, i, &m_rgRenderers[i]));
        }

        // Default to the default adapter
        m_pCurrentRenderer = m_rgRenderers[0];
    }

    Cleanup:
    return hr;
}

//+-----
// Member:
//     CRendererManager::EnsureHWND
//
// Synopsis:
//     Makes sure an HWND exists if we need it
//
//-----

HRESULT
CRendererManager::EnsureHWND()
{
    HRESULT hr = S_OK;

    if (!m_hwnd)
    {
        WNDCLASS wndclass;

        wndclass.style = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfWndProc = DefWindowProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hInstance = NULL;
        wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName = NULL;
        wndclass.lpszClassName = szAppName;

        if (!RegisterClass(&wndclass))
        {
            IFC(E_FAIL);
        }

        m_hwnd = CreateWindow(szAppName,
                              TEXT("D3DImageSample"),
                              WS_OVERLAPPEDWINDOW,
                              0,                               // Initial X
                              0,                               // Initial Y
                              0,                               // Width
                              0);                             // Height
    }
}

```

```

        0,           // Height
        NULL,
        NULL,
        NULL,
        NULL);
}

Cleanup:
    return hr;
}

//+-----//
// Member:
//     CRendererManager::EnsureD3DObjects
//
// Synopsis:
//     Makes sure the D3D objects exist
//
//-----//
HRESULT
CRendererManager::EnsureD3DObjects()
{
    HRESULT hr = S_OK;

    HMODULE hD3D = NULL;
    if (!m_pD3D)
    {
        hD3D = LoadLibrary(TEXT("d3d9.dll"));
        DIRECT3DCREATE9EXFUNCTION pfnCreate9Ex = (DIRECT3DCREATE9EXFUNCTION)GetProcAddress(hD3D,
"Direct3DCreate9Ex");
        if (pfnCreate9Ex)
        {
            IFC((*pfnCreate9Ex)(D3D_SDK_VERSION, &m_pD3DEx));
            IFC(m_pD3DEx->QueryInterface(__uuidof(IDirect3D9), reinterpret_cast<void **>(&m_pD3D)));
        }
        else
        {
            m_pD3D = Direct3DCreate9(D3D_SDK_VERSION);
            if (!m_pD3D)
            {
                IFC(E_FAIL);
            }
        }
    }

    m_cAdapters = m_pD3D->GetAdapterCount();
}

Cleanup:
    if (hD3D)
    {
        FreeLibrary(hD3D);
    }

    return hr;
}

//+-----//
// Member:
//     CRendererManager::CleanupInvalidDevices
//
// Synopsis:
//     Checks to see if any devices are bad and if so, deletes all resources
//
//     We could delete resources and wait for D3DERR_DEVICENOTRESET and reset
//     the devices, but if the device is lost because of an adapter order
//     change then our existing D3D objects would have stale adapter
//     information. We'll delete everything to be safe rather than sorry.

```

```

// -----
// void
CRendererManager::CleanupInvalidDevices()
{
    for (UINT i = 0; i < m_cAdapters; ++i)
    {
        if (FAILED(m_rgRenderers[i]->CheckDeviceState()))
        {
            DestroyResources();
            break;
        }
    }
}

//+-----
// Member:
//     CRendererManager::GetBackBufferNoRef
//
// Synopsis:
//     Returns the surface of the current renderer without adding a reference
//
//     This can return NULL if we're in a bad device state.
//
// -----
HRESULT
CRendererManager::GetBackBufferNoRef(IDirect3DSurface9 **ppSurface)
{
    HRESULT hr = S_OK;

    // Make sure we at least return NULL
    *ppSurface = NULL;

    CleanupInvalidDevices();

    IFC(EnsureD3DOBJECTS());

    //
    // Even if we never render to another adapter, this sample creates devices
    // and resources on each one. This is a potential waste of video memory,
    // but it guarantees that we won't have any problems (e.g. out of video
    // memory) when switching to render on another adapter. In your own code
    // you may choose to delay creation but you'll need to handle the issues
    // that come with it.
    //

    IFC(EnsureRenderers());

    if (m_fSurfaceSettingsChanged)
    {
        if (FAILED(TestSurfaceSettings()))
        {
            IFC(E_FAIL);
        }

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            IFC(m_rgRenderers[i]->CreateSurface(m_uWidth, m_uHeight, m_fUseAlpha, m_uNumSamples));
        }
    }

    m_fSurfaceSettingsChanged = false;
}

if (m_pCurrentRenderer)
{
    *ppSurface = m_pCurrentRenderer->GetSurfaceNoRef();
}

```

```

Cleanup:
    // If we failed because of a bad device, ignore the failure for now and
    // we'll clean up and try again next time.
    if (hr == D3DERR_DEVICELOST)
    {
        hr = S_OK;
    }

    return hr;
}

//-----
// Member:
//     CRendererManager::TestSurfaceSettings
//
// Synopsis:
//     Checks to see if our current surface settings are allowed on all
//     adapters.
//
//-----

HRESULT
CRendererManager::TestSurfaceSettings()
{
    HRESULT hr = S_OK;

    D3DFORMAT fmt = m_fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8;

    //
    // We test all adapters because potentially we use all adapters.
    // But even if this sample only rendered to the default adapter, you
    // should check all adapters because WPF may move your surface to
    // another adapter for you!
    //

    for (UINT i = 0; i < m_cAdapters; ++i)
    {
        // Can we get HW rendering?
        IFC(m_pD3D->CheckDeviceType(
            i,
            D3DDEVTYPE_HAL,
            D3DFMT_X8R8G8B8,
            fmt,
            TRUE
        ));

        // Is the format okay?
        IFC(m_pD3D->CheckDeviceFormat(
            i,
            D3DDEVTYPE_HAL,
            D3DFMT_X8R8G8B8,
            D3DUSAGE_RENDERTARGET | D3DUSAGE_DYNAMIC, // We'll use dynamic when on XP
            D3DRTYPE_SURFACE,
            fmt
        ));

        // D3DImage only allows multisampling on 9Ex devices. If we can't
        // multisample, overwrite the desired number of samples with 0.
        if (m_pD3DEx && m_uNumSamples > 1)
        {
            assert(m_uNumSamples <= 16);

            if (FAILED(m_pD3D->CheckDeviceMultiSampleType(
                i,
                D3DDEVTYPE_HAL,
                fmt,
                TRUE,
                static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),
                NULL
            )))
        }
    }
}

```

```

        )))

    {
        m_uNumSamples = 0;
    }
}

else
{
    m_uNumSamples = 0;
}
}

Cleanup:
    return hr;
}

//+-----
// Member:
//     CRendererManager::DestroyResources
//
// Synopsis:
//     Delete all D3D resources
//
//-----
void
CRendererManager::DestroyResources()
{
    SAFE_RELEASE(m_pD3D);
    SAFE_RELEASE(m_pD3DEx);

    for (UINT i = 0; i < m_cAdapters; ++i)
    {
        delete m_rgRenderers[i];
    }
    delete [] m_rgRenderers;
    m_rgRenderers = NULL;

    m_pCurrentRenderer = NULL;
    m_cAdapters = 0;

    m_fSurfaceSettingsChanged = true;
}

//+-----
// Member:
//     CRendererManager::SetSize
//
// Synopsis:
//     Update the size of the surface. Next render will create a new surface.
//
//-----
void
CRendererManager::SetSize(UINT uWidth, UINT uHeight)
{
    if (uWidth != m_uWidth || uHeight != m_uHeight)
    {
        m_uWidth = uWidth;
        m_uHeight = uHeight;
        m_fSurfaceSettingsChanged = true;
    }
}

//+-----
// Member:
//     CRendererManager::SetAlpha
//
// Synopsis:

```

```

//      Update the format of the surface. Next render will create a new surface.
//
//-----
void
CRendererManager::SetAlpha(bool fUseAlpha)
{
    if (fUseAlpha != m_fUseAlpha)
    {
        m_fUseAlpha = fUseAlpha;
        m_fSurfaceSettingsChanged = true;
    }
}

//+-----
// 
// Member:
//      CRendererManager::SetNumDesiredSamples
//
// Synopsis:
//      Update the MSAA settings of the surface. Next render will create a
//      new surface.
//
//-----
void
CRendererManager::SetNumDesiredSamples(UINT uNumSamples)
{
    if (m_uNumSamples != uNumSamples)
    {
        m_uNumSamples = uNumSamples;
        m_fSurfaceSettingsChanged = true;
    }
}

//+-----
// 
// Member:
//      CRendererManager::SetAdapter
//
// Synopsis:
//      Update the current renderer. Next render will use the new renderer.
//
//-----
void
CRendererManager::SetAdapter(POINT screenSpacePoint)
{
    CleanupInvalidDevices();

    //
    // After CleanupInvalidDevices, we may not have any D3D objects. Rather than
    // recreate them here, ignore the adapter update and wait for render to recreate.
    //

    if (m_pD3D && m_rgRenderers)
    {
        HMONITOR hMon = MonitorFromPoint(screenSpacePoint, MONITOR_DEFAULTTONULL);

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            if (hMon == m_pD3D->GetAdapterMonitor(i))
            {
                m_pCurrentRenderer = m_rgRenderers[i];
                break;
            }
        }
    }
}

//+-----
// 

```

```


// Member:
//     CRendererManager::Render
//
// Synopsis:
//     Forward to the current renderer
//
//-----
HRESULT
CRendererManager::Render()
{
    return m_pCurrentRenderer ? m_pCurrentRenderer->Render() : S_OK;
}


```

6. Open TriangleRenderer.h in the Code Editor and replace the automatically generated code with the following code.

```


#pragma once

class CTriangleRenderer : public CRenderer
{
public:
    static HRESULT Create(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND hwnd, UINT uAdapter, CRenderer
**ppRenderer);
    ~CTriangleRenderer();

    HRESULT Render();

protected:
    HRESULT Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND hwnd, UINT uAdapter);

private:
    CTriangleRenderer();
    IDirect3DVertexBuffer9 *m_pd3dVB;
};


```

7. Open TriangleRenderer.cpp in the Code Editor and replace the automatically generated code with the following code.

```


//-----
// CTriangleRenderer
// Subclass of CRenderer that renders a single, spinning triangle
//-----
#include "StdAfx.h"

struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color;
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)

//+
// Member:
//     CTriangleRenderer ctor
//-----


```

```

CTriangleRenderer::CTriangleRenderer() : CRenderer(), m_pd3dVB(NULL)
{
}

//+-----
// Member:
//     CTriangleRenderer dtor
//
//-----
CTriangleRenderer::~CTriangleRenderer()
{
    SAFE_RELEASE(m_pd3dVB);
}

//+-----
// Member:
//     CTriangleRenderer::Create
//
// Synopsis:
//     Creates the renderer
//
//-----
HRESULT
CTriangleRenderer::Create(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND hwnd, UINT uAdapter, CRenderer
**ppRenderer)
{
    HRESULT hr = S_OK;

    CTriangleRenderer *pRenderer = new CTriangleRenderer();
    IFCOOM(pRenderer);

    IFC(pRenderer->Init(pD3D, pD3DEx, hwnd, uAdapter));

    *ppRenderer = pRenderer;
    pRenderer = NULL;

    Cleanup:
    delete pRenderer;

    return hr;
}

//+-----
// Member:
//     CTriangleRenderer::Init
//
// Synopsis:
//     Override of CRenderer::Init that calls base to create the device and
//     then creates the CTriangleRenderer-specific resources
//
//-----
HRESULT
CTriangleRenderer::Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND hwnd, UINT uAdapter)
{
    HRESULT hr = S_OK;
    D3DXMATRIXA16 matView, matProj;
    D3DXVECTOR3 vEyePt(0.0f, 0.0f, -5.0f);
    D3DXVECTOR3 vLookatPt(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 vUpVec(0.0f, 1.0f, 0.0f);

    // Call base to create the device and render target
    IFC(CRenderer::Init(pD3D, pD3DEx, hwnd, uAdapter));

    // Set up the VB
    CUSTOMVERTEX vertices[] =
    {

```

```

        { -1.0f, -1.0f, 0.0f, 0xffff0000, }, // x, y, z, color
        { 1.0f, -1.0f, 0.0f, 0xff00ff00, },
        { 0.0f, 1.0f, 0.0f, 0xff00ffff, },
    };

    IFC(m_pd3dDevice->CreateVertexBuffer(sizeof(vertices), 0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT,
&m_pd3dVB, NULL));

    void *pVertices;
    IFC(m_pd3dVB->Lock(0, sizeof(vertices), &pVertices, 0));
    memcpy(pVertices, vertices, sizeof(vertices));
    m_pd3dVB->Unlock();

    // Set up the camera
    D3DXMatrixLookAtLH(&matView, &vEyePt, &vLookatPt, &vUpVec);
    IFC(m_pd3dDevice->SetTransform(D3DTS_VIEW, &matView));
    D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI / 4, 1.0f, 1.0f, 100.0f);
    IFC(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj));

    // Set up the global state
    IFC(m_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE));
    IFC(m_pd3dDevice->SetRenderState(D3DRS_LIGHTING, FALSE));
    IFC(m_pd3dDevice->SetStreamSource(0, m_pd3dVB, 0, sizeof(CUSTOMVERTEX)));
    IFC(m_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX));

    Cleanup:
    return hr;
}

//+-----+
// Member:
//     CTriangleRenderer::Render
//
// Synopsis:
//     Renders the rotating triangle
//-----
HRESULT
CTriangleRenderer::Render()
{
    HRESULT hr = S_OK;
    D3DXMATRIXA16 matWorld;

    IFC(m_pd3dDevice->BeginScene());
    IFC(m_pd3dDevice->Clear(
        0,
        NULL,
        D3DCLEAR_TARGET,
        D3DCOLOR_ARGB(128, 0, 0, 128), // NOTE: Premultiplied alpha!
        1.0f,
        0
    ));

    // Set up the rotation
    UINT iTime = GetTickCount() % 1000;
    FLOAT fAngle = iTime * (2.0f * D3DX_PI) / 1000.0f;
    D3DXMatrixRotationY(&matWorld, fAngle);
    IFC(m_pd3dDevice->SetTransform(D3DTS_WORLD, &matWorld));

    IFC(m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1));

    IFC(m_pd3dDevice->EndScene());

    Cleanup:
    return hr;
}

```

8. Open stdafx.h in the Code Editor and replace the automatically generated code with the following code.

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#define WIN32_LEAN_AND_MEAN           // Exclude rarely-used stuff from Windows headers
// Windows Header Files:
#include <windows.h>

#include <d3d9.h>
#include <d3dx9.h>

#include <assert.h>

#include "RendererManager.h"
#include "Renderer.h"
#include "TriangleRenderer.h"

#define IFC(x) { hr = (x); if (FAILED(hr)) goto Cleanup; }
#define IFCOOM(x) { if ((x) == NULL) { hr = E_OUTOFMEMORY; IFC(hr); } }
#define SAFE_RELEASE(x) { if (x) { x->Release(); x = NULL; } }


```

9. Open dllmain.cpp in the Code Editor and replace the automatically generated code with the following code.

```
// dllmain.cpp : Defines the entry point for the DLL application.
#include "stdafx.h"

BOOL APIENTRY DllMain( HMODULE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
)
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

static CRendererManager *pManager = NULL;

static HRESULT EnsureRendererManager()
{
    return pManager ? S_OK : CRendererManager::Create(&pManager);
}

extern "C" HRESULT WINAPI SetSize(UINT uWidth, UINT uHeight)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetSize(uWidth, uHeight);

Cleanup:
    return hr;
}
```

```

extern "C" HRESULT WINAPI SetAlpha(BOOL fUseAlpha)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetAlpha(!fUseAlpha);

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI SetNumDesiredSamples(UINT uNumSamples)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetNumDesiredSamples(uNumSamples);

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI SetAdapter(POINT screenSpacePoint)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetAdapter(screenSpacePoint);

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI GetBackBufferNoRef(IDirect3DSurface9 **ppSurface)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    IFC(pManager->GetBackBufferNoRef(ppSurface));

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI Render()
{
    assert(pManager);

    return pManager->Render();
}

extern "C" void WINAPI Destroy()
{
    delete pManager;
    pManager = NULL;
}

```

10. Open D3DContent.def in the code editor.
11. Replace the automatically generated code with the following code.

```
LIBRARY "D3DContent"
```

```
EXPORTS
```

```
SetSize  
SetAlpha  
SetNumDesiredSamples  
SetAdapter
```

```
GetBackBufferNoRef  
Render  
Destroy
```

12. Build the project.

## Next Steps

- Host the Direct3D9 content in a WPF application. For more information, see [Walkthrough: Hosting Direct3D9 Content in WPF](#).

## See also

- [D3DImage](#)
- [Performance Considerations for Direct3D9 and WPF Interoperability](#)
- [Walkthrough: Hosting Direct3D9 Content in WPF](#)

# Walkthrough: Hosting Direct3D9 Content in WPF

11/12/2019 • 5 minutes to read • [Edit Online](#)

This walkthrough shows how to host Direct3D9 content in a Windows Presentation Foundation (WPF) application.

In this walkthrough, you perform the following tasks:

- Create a WPF project to host the Direct3D9 content.
- Import the Direct3D9 content.
- Display the Direct3D9 content by using the [D3DImage](#) class.

When you are finished, you will know how to host Direct3D9 content in a WPF application.

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio.
- DirectX SDK 9 or later.
- A DLL that contains Direct3D9 content in a WPF-compatible format. For more information, see [WPF and Direct3D9 Interoperation](#) and [Walkthrough: Creating Direct3D9 Content for Hosting in WPF](#).

## Creating the WPF Project

The first step is to create the project for the WPF application.

### To create the WPF project

Create a new WPF Application project in Visual C# named `D3DHost`. For more information, see [Walkthrough: My first WPF desktop application](#).

`MainWindow.xaml` opens in the WPF Designer.

## Importing the Direct3D9 Content

You import the Direct3D9 content from an unmanaged DLL by using the `DllImport` attribute.

### To import Direct3D9 content

1. Open `MainWindow.xaml.cs` in the Code Editor.
2. Replace the automatically generated code with the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Interop;
```

```
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Threading;
using System.Runtime.InteropServices;
using System.Security.Permissions;

namespace D3DHost
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            // Set up the initial state for the D3DImage.
            HRESULT.Check(SetSize(512, 512));
            HRESULT.Check(SetAlpha(false));
            HRESULT.Check(SetNumDesiredSamples(4));

            //
            // Optional: Subscribing to the IsFrontBufferAvailableChanged event.
            //
            // If you don't render every frame (e.g. you only render in
            // reaction to a button click), you should subscribe to the
            // IsFrontBufferAvailableChanged event to be notified when rendered content
            // is no longer being displayed. This event also notifies you when
            // the D3DImage is capable of being displayed again.

            // For example, in the button click case, if you don't render again when
            // the IsFrontBufferAvailable property is set to true, your
            // D3DImage won't display anything until the next button click.
            //
            // Because this application renders every frame, there is no need to
            // handle the IsFrontBufferAvailableChanged event.
            //
            CompositionTarget.Rendering += new EventHandler(CompositionTarget_Rendering);

            //
            // Optional: Multi-adapter optimization
            //
            // The surface is created initially on a particular adapter.
            // If the WPF window is dragged to another adapter, WPF
            // ensures that the D3DImage still shows up on the new
            // adapter.
            //
            // This process is slow on Windows XP.
            //
            // Performance is better on Vista with a 9Ex device. It's only
            // slow when the D3DImage crosses a video-card boundary.
            //
            // To work around this issue, you can move your surface when
            // the D3DImage is displayed on another adapter. To
            // determine when that is the case, transform a point on the
            // D3DImage into screen space and find out which adapter
            // contains that screen space point.
            //
            // When your D3DImage straddles two adapters, nothing
            // can be done, because one will be updating slowly.
            //
            _adapterTimer = new DispatcherTimer();
            _adapterTimer.Tick += new EventHandler(AdapterTimer_Tick);
            _adapterTimer.Interval = new TimeSpan(0, 0, 0, 0, 500);
            _adapterTimer.Start();

            //
            // Optional: Surface resizing
            //
        }
    }
}
```

```

// The D3DImage is scaled when WPF renders it at a size
// different from the natural size of the surface. If the
// D3DImage is scaled up significantly, image quality
// degrades.
//
// To avoid this, you can either create a very large
// texture initially, or you can create new surfaces as
// the size changes. Below is a very simple example of
// how to do the latter.
//
// By creating a timer at Render priority, you are guaranteed
// that new surfaces are created while the element
// is still being arranged. A 200 ms interval gives
// a good balance between image quality and performance.
// You must be careful not to create new surfaces too
// frequently. Frequently allocating a new surface may
// fragment or exhaust video memory. This issue is more
// significant on XDDM than it is on WDDM, because WDDM
// can page out video memory.
//
// Another approach is deriving from the Image class,
// participating in layout by overriding the ArrangeOverride method, and
// updating size in the overridden method. Performance will degrade
// if you resize too frequently.
//
// Blurry D3DImages can still occur due to subpixel
// alignments.
//
_sizeTimer = new DispatcherTimer(DispatcherPriority.Render);
_sizeTimer.Tick += new EventHandler(SizeTimer_Tick);
_sizeTimer.Interval = new TimeSpan(0, 0, 0, 0, 200);
_sizeTimer.Start();
}

~MainWindow()
{
    Destroy();
}

void AdapterTimer_Tick(object sender, EventArgs e)
{
    POINT p = new POINT(imgelt.PointToScreen(new Point(0, 0)));

    HRESULT.Check(SetAdapter(p));
}

void SizeTimer_Tick(object sender, EventArgs e)
{
    // The following code does not account for RenderTransforms.
    // To handle that case, you must transform up to the root and
    // check the size there.

    // Given that the D3DImage is at 96.0 DPI, its Width and Height
    // properties will always be integers. ActualWidth/Height
    // may not be integers, so they are cast to integers.
    uint actualWidth = (uint)imgelt.ActualWidth;
    uint actualHeight = (uint)imgelt.ActualHeight;
    if ((actualWidth > 0 && actualHeight > 0) &&
        (actualWidth != (uint)d3dimg.Width || actualHeight != (uint)d3dimg.Height))
    {
        HRESULT.Check(SetSize(actualWidth, actualHeight));
    }
}

void CompositionTarget_Rendering(object sender, EventArgs e)
{
    RenderingEventArgs args = (RenderingEventArgs)e;

    // It's possible for Rendering to call back twice in the same frame
}

```

```

        // so only render when we haven't already rendered in this frame.
        if (d3dimg.IsFrontBufferAvailable && _lastRender != args.RenderingTime)
        {
            IntPtr pSurface = IntPtr.Zero;
            HRESULT.Check(GetBackBufferNoRef(out pSurface));
            if (pSurface != IntPtr.Zero)
            {
                d3dimg.Lock();
                // Repeatedly calling SetBackBuffer with the same IntPtr is
                // a no-op. There is no performance penalty.
                d3dimg.SetBackBuffer(D3DResourceType.IDirect3DSurface9, pSurface);
                HRESULT.Check(Render());
                d3dimg.AddDirtyRect(new Int32Rect(0, 0, d3dimg.PixelWidth, d3dimg.PixelHeight));
                d3dimg.Unlock();

                _lastRender = args.RenderingTime;
            }
        }
    }

    DispatcherTimer _sizeTimer;
    DispatcherTimer _adapterTimer;
    TimeSpan _lastRender;

    // Import the methods exported by the unmanaged Direct3D content.

    [DllImport("D3DCode.dll")]
    static extern int GetBackBufferNoRef(out IntPtr pSurface);

    [DllImport("D3DCode.dll")]
    static extern int SetSize(uint width, uint height);

    [DllImport("D3DCode.dll")]
    static extern int SetAlpha(bool useAlpha);

    [DllImport("D3DCode.dll")]
    static extern int SetNumDesiredSamples(uint numSamples);

    [StructLayout(LayoutKind.Sequential)]
    struct POINT
    {
        public POINT(Point p)
        {
            x = (int)p.X;
            y = (int)p.Y;
        }

        public int x;
        public int y;
    }

    [DllImport("D3DCode.dll")]
    static extern int SetAdapter(POINT screenSpacePoint);

    [DllImport("D3DCode.dll")]
    static extern int Render();

    [DllImport("D3DCode.dll")]
    static extern void Destroy();
}

public static class HRESULT
{
    [SecurityPermissionAttribute(SecurityAction.Demand, Flags =
    SecurityPermissionFlag.UnmanagedCode)]
    public static void Check(int hr)
    {
        Marshal.ThrowExceptionForHR(hr);
    }
}

```

```
    }  
}
```

## Hosting the Direct3D9 Content

Finally, use the [D3DImage](#) class to host the Direct3D9 content.

### To host the Direct3D9 content

1. In MainWindow.xaml, replace the automatically generated XAML with the following XAML.

```
<Window x:Class="D3DHost.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       xmlns:i="clr-namespace:System.Windows.Interop;assembly=PresentationCore"  
       Title="MainWindow" Height="300" Width="300" Background="PaleGoldenrod">  
    <Grid>  
        <Image x:Name="imgelt">  
            <Image.Source>  
                <i:D3DImage x:Name="d3dimg" />  
            </Image.Source>  
        </Image>  
    </Grid>  
</Window>
```

2. Build the project.
3. Copy the DLL that contains the Direct3D9 content to the bin/Debug folder.
4. Press F5 to run the project.

The Direct3D content appears within the WPF application.

## See also

- [D3DImage](#)
- [Performance Considerations for Direct3D9 and WPF Interoperability](#)

# Performance

4/8/2019 • 2 minutes to read • [Edit Online](#)

Achieving optimal application performance requires forethought in application design and an understanding of best practices for Windows Presentation Foundation (WPF) application development. The topics in this section provide additional information on building high performance WPF applications.

## In This Section

[Graphics Rendering Tiers](#)

[Optimizing WPF Application Performance](#)

[Walkthrough: Caching Application Data in a WPF Application](#)

## Reference

[RenderCapability](#)

[Tier](#)

[PresentationTraceSources](#)

## See also

- [Layout](#)
- [Animation Tips and Tricks](#)

# Graphics Rendering Tiers

12/10/2019 • 6 minutes to read • [Edit Online](#)

A rendering tier defines a level of graphics hardware capability and performance for a device that runs a WPF application.

## Graphics Hardware

The features of the graphics hardware that most impact the rendering tier levels are:

- **Video RAM** The amount of video memory on the graphics hardware determines the size and number of buffers that can be used for compositing graphics.
- **Pixel Shader** A pixel shader is a graphics processing function that calculates effects on a per-pixel basis. Depending on the resolution of the displayed graphics, there could be several million pixels that need to be processed for each display frame.
- **Vertex Shader** A vertex shader is a graphics processing function that performs mathematical operations on the vertex data of the object.
- **Multitexture Support** Multitexture support refers to the ability to apply two or more distinct textures during a blending operation on a 3D graphics object. The degree of multitexture support is determined by the number of multitexture units on the graphics hardware.

## Rendering Tier Definitions

The features of the graphics hardware determine the rendering capability of a WPF application. The WPF system defines three rendering tiers:

- **Rendering Tier 0** No graphics hardware acceleration. All graphics features use software acceleration. The DirectX version level is less than version 9.0.
- **Rendering Tier 1** Some graphics features use graphics hardware acceleration. The DirectX version level is greater than or equal to version 9.0.
- **Rendering Tier 2** Most graphics features use graphics hardware acceleration. The DirectX version level is greater than or equal to version 9.0.

The [RenderCapability.Tier](#) property allows you to retrieve the rendering tier at application run time. You use the rendering tier to determine whether the device supports certain hardware-accelerated graphics features. Your application can then take different code paths at run time depending on the rendering tier supported by the device.

### Rendering Tier 0

A rendering tier value of 0 means that there is no graphics hardware acceleration available for the application on the device. At this tier level, you should assume that all graphics will be rendered by software with no hardware acceleration. This tier's functionality corresponds to a DirectX version that is less than 9.0.

### Rendering Tier 1 and Rendering Tier 2

**NOTE**

Starting in the .NET Framework 4, rendering tier 1 has been redefined to only include graphics hardware that supports DirectX 9.0 or greater. Graphics hardware that supports DirectX 7 or 8 is now defined as rendering tier 0.

A rendering tier value of 1 or 2 means that most of the graphics features of WPF will use hardware acceleration if the necessary system resources are available and have not been exhausted. This corresponds to a DirectX version that is greater than or equal to 9.0.

The following table shows the differences in graphics hardware requirements for rendering tier 1 and rendering tier 2:

FEATURE	TIER 1	TIER 2
DirectX version	Must be greater than or equal to 9.0.	Must be greater than or equal to 9.0.
Video RAM	Must be greater than or equal to 60MB.	Must be greater than or equal to 120MB.
Pixel shader	Version level must greater than or equal to 2.0.	Version level must greater than or equal to 2.0.
Vertex shader	No requirement.	Version level must greater than or equal to 2.0.
Multitexture units	No requirement.	Number of units must greater than or equal to 4.

The following features and capabilities are hardware accelerated for rendering tier 1 and rendering tier 2:

FEATURE	NOTES
2D rendering	Most 2D rendering is supported.
3D rasterization	Most 3D rasterization is supported.
3D anisotropic filtering	WPF attempts to use anisotropic filtering when rendering 3D content. Anisotropic filtering refers to enhancing the image quality of textures on surfaces that are far away and steeply angled with respect to the camera.
3D MIP mapping	WPF attempts to use MIP mapping when rendering 3D content. MIP mapping improves the quality of texture rendering when a texture occupies a smaller field of view in a <a href="#">Viewport3D</a> .
Radial gradients	While supported, avoid the use of <a href="#">RadialGradientBrush</a> on large objects.
3D lighting calculations	WPF performs per-vertex lighting, which means that a light intensity must be calculated at each vertex for each material applied to a mesh.
Text rendering	Sub-pixel font rendering uses available pixel shaders on the graphics hardware.

The following features and capabilities are hardware accelerated only for rendering tier 2:

FEATURE	NOTES
3D anti-aliasing	3D anti-aliasing is supported only on operating systems that support Windows Display Driver Model (WDDM), such as Windows Vista and Windows 7.

The following features and capabilities are **not** hardware accelerated:

FEATURE	NOTES
Printed content	All printed content is rendered using the WPF software pipeline.
Rasterized content that uses <a href="#">RenderTargetBitmap</a>	Any content rendered by using the <a href="#">Render</a> method of <a href="#">RenderTargetBitmap</a> .
Tiled content that uses <a href="#">TileBrush</a>	Any tiled content in which the <a href="#">TileMode</a> property of the <a href="#">TileBrush</a> is set to <a href="#">Tile</a> .
Surfaces that exceed the maximum texture size of the graphics hardware	For most graphics hardware, large surfaces are 2048x2048 or 4096x4096 pixels in size.
Any operation whose video RAM requirement exceeds the memory of the graphics hardware	You can monitor application video RAM usage by using the Perforator tool that is included in the <a href="#">WPF Performance Suite</a> in the Windows SDK.
Layered windows	<p>Layered windows allow WPF applications to render content to the screen in a non-rectangular window. On operating systems that support Windows Display Driver Model (WDDM), such as Windows Vista and Windows 7, layered windows are hardware accelerated. On other systems, such as Windows XP, layered windows are rendered by software with no hardware acceleration.</p> <p>You can enable layered windows in WPF by setting the following <a href="#">Window</a> properties:</p> <ul style="list-style-type: none"> <li>- <a href="#">WindowStyle = None</a></li> <li>- <a href="#">AllowsTransparency = true</a></li> <li>- <a href="#">Background = Transparent</a></li> </ul>

## Other Resources

The following resources can help you analyze the performance characteristics of your WPF application.

### Graphics Rendering Registry Settings

WPF provides four registry settings for controlling WPF rendering:

SETTING	DESCRIPTION
<b>Disable Hardware Acceleration Option</b>	Specifies whether hardware acceleration should be enabled.
<b>Maximum Multisample Value</b>	Specifies the degree of multisampling for antialiasing 3-D content.

SETTING	DESCRIPTION
<b>Required Video Driver Date Setting</b>	Specifies whether the system disables hardware acceleration for drivers released before November 2004.
<b>Use Reference Rasterizer Option</b>	Specifies whether WPF should use the reference rasterizer.

These settings can be accessed by any external configuration utility that knows how to reference the WPF registry settings. These settings can also be created or modified by accessing the values directly by using the Windows Registry Editor. For more information, see [Graphics Rendering Registry Settings](#).

### WPF Performance Profiling Tools

WPF provides a suite of performance profiling tools that allow you to analyze the run-time behavior of your application and determine the types of performance optimizations you can apply. The following table lists the performance profiling tools that are included in the Windows SDK tool, WPF Performance Suite:

TOOL	DESCRIPTION
Perforator	Use for analyzing rendering behavior.
Visual Profiler	Use for profiling the use of WPF services, such as layout and event handling, by elements in the visual tree.

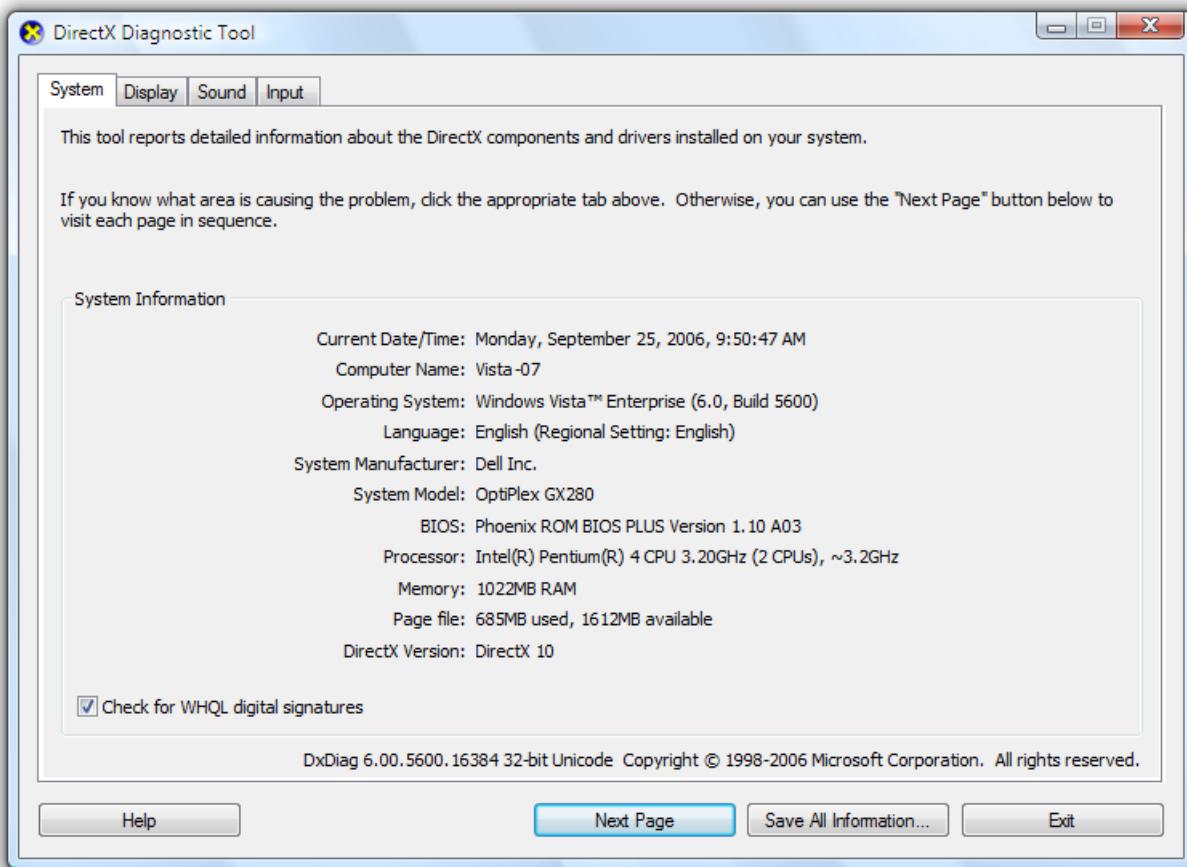
The WPF Performance Suite provides a rich, graphical view of performance data. For more information about WPF performance tools, see [WPF Performance Suite](#).

### DirectX Diagnostic Tool

The DirectX Diagnostic Tool, Dxdiag.exe, is designed to help you troubleshoot DirectX-related issues. The default installation folder for the DirectX Diagnostic Tool is:

`~\Windows\System32`

When you run the DirectX Diagnostic Tool, the main window contains a set of tabs that allow you to display and diagnose DirectX-related information. For example, the **System** tab provides system information about your computer and specifies the version of DirectX that is installed on your computer.



DirectX Diagnostic Tool main window

## See also

- [RenderCapability](#)
- [RenderOptions](#)
- [Optimizing WPF Application Performance](#)
- [WPF Performance Suite](#)
- [Graphics Rendering Registry Settings](#)
- [Animation Tips and Tricks](#)

# Optimizing WPF Application Performance

11/3/2019 • 2 minutes to read • [Edit Online](#)

This section is intended as a reference for Windows Presentation Foundation (WPF) application developers who are looking for ways to improve the performance of their applications. If you are a developer who is new to the Microsoft .NET Framework and WPF, you should first familiarize yourself with both platforms. This section assumes working knowledge of both, and is written for programmers who already know enough to get their applications up and running.

## NOTE

The performance data provided in this section are based on WPF applications running on a 2.8 GHz PC with 512 RAM and an ATI Radeon 9700 graphics card.

## In This Section

[Planning for Application Performance](#)

[Taking Advantage of Hardware](#)

[Layout and Design](#)

[2D Graphics and Imaging](#)

[Object Behavior](#)

[Application Resources](#)

[Text](#)

[Data Binding](#)

[Controls](#)

[Other Performance Recommendations](#)

[Application Startup Time](#)

## See also

- [RenderOptions](#)
- [RenderCapability](#)
- [Graphics Rendering Tiers](#)
- [WPF Graphics Rendering Overview](#)
- [Layout](#)
- [Trees in WPF](#)
- [Drawing Objects Overview](#)
- [Using DrawingVisual Objects](#)
- [Dependency Properties Overview](#)
- [Freezable Objects Overview](#)
- [XAML Resources](#)

- [Documents in WPF](#)
- [Drawing Formatted Text](#)
- [Typography in WPF](#)
- [Data Binding Overview](#)
- [Navigation Overview](#)
- [Animation Tips and Tricks](#)
- [Walkthrough: Caching Application Data in a WPF Application](#)

# Planning for Application Performance

4/8/2019 • 2 minutes to read • [Edit Online](#)

The success of achieving your performance goals depends on how well you develop your performance strategy. Planning is the first stage in developing any product. This topic describes a few very simple rules for developing a good performance strategy.

## Think in Terms of Scenarios

Scenarios can help you focus on the critical components of your application. Scenarios are generally derived from your customers, as well as competitive products. Always study your customers and find out what really makes them excited about your product, and your competitors' products. Your customers' feedback can help you to determine your application's primary scenario. For instance, if you are designing a component that will be used at startup, it is likely that the component will be called only once, when the application starts up. Startup time becomes your key scenario. Other examples of key scenarios could be the desired frame rate for animation sequences, or the maximum working set allowed for the application.

## Define Goals

Goals help you to determine whether an application is performing faster or slower. You should define goals for all of your scenarios. All performance goals that you define should be based on your customers' expectations. It may be difficult to set performance goals early on in the application development cycle, when there are still many unresolved issues. However, it is better to set an initial goal and revise it later than not to have a goal at all.

## Understand Your Platform

Always maintain the cycle of measuring, investigating, refining/correcting during your application development cycle. From the beginning to the end of the development cycle, you need to measure your application's performance in a reliable, stable environment. You should avoid variability caused by external factors. For example, when testing performance, you should disable anti-virus or any automatic update such as SMS, in order not to impact performance test results. Once you have measured your application's performance, you need to identify the changes that will result in the biggest improvements. Once you have modified your application, start the cycle again.

## Make Performance Tuning an Iterative Process

You should know the relative cost of each feature you will use. For example, the use of reflection in Microsoft .NET Framework is generally performance intensive in terms of computing resources, so you would want to use it judiciously. This does not mean to avoid the use of reflection, only that you should be careful to balance the performance requirements of your application with the performance demands of the features you use.

## Build Towards Graphical Richness

A key technique for creating a scalable approach towards achieving WPF application performance is to build towards graphical richness and complexity. Always start with using the least performance intensive resources to achieve your scenario goals. Once you achieve these goals, build towards graphic richness by using more performance intensive features, always keeping your scenario goals in mind. Remember, WPF is a very rich platform and provides very rich graphic features. Using performance intensive features without thinking can negatively impact your overall application performance.

WPF controls are inherently extensible by allowing for wide-spread customization of their appearance, while not altering their control behavior. By taking advantage of styles, data templates, and control templates, you can create and incrementally evolve a customizable user interface (UI) that adapts to your performance requirements.

## See also

- [Optimizing WPF Application Performance](#)
- [Taking Advantage of Hardware](#)
- [Layout and Design](#)
- [2D Graphics and Imaging](#)
- [Object Behavior](#)
- [Application Resources](#)
- [Text](#)
- [Data Binding](#)
- [Other Performance Recommendations](#)

# Optimizing Performance: Taking Advantage of Hardware

8/1/2019 • 3 minutes to read • [Edit Online](#)

The internal architecture of WPF has two rendering pipelines, hardware and software. This topic provides information about these rendering pipelines to help you make decisions about performance optimizations of your applications.

## Hardware Rendering Pipeline

One of the most important factors in determining WPF performance is that it is render bound—the more pixels you have to render, the greater the performance cost. However, the more rendering that can be offloaded to the graphics processing unit (GPU), the more performance benefits you can gain. The WPF application hardware rendering pipeline takes full advantage of Microsoft DirectX features on hardware that supports a minimum of Microsoft DirectX version 7.0. Further optimizations can be gained by hardware that supports Microsoft DirectX version 7.0 and PixelShader 2.0+ features.

## Software Rendering Pipeline

The WPF software rendering pipeline is entirely CPU bound. WPF takes advantage of the SSE and SSE2 instruction sets in the CPU to implement an optimized, fully-featured software rasterizer. Fallback to software is seamless any time application functionality cannot be rendered using the hardware rendering pipeline.

The biggest performance issue you will encounter when rendering in software mode is related to fill rate, which is defined as the number of pixels that you are rendering. If you are concerned about performance in software rendering mode, try to minimize the number of times a pixel is redrawn. For example, if you have an application with a blue background, which then renders a slightly transparent image over it, you will render all of the pixels in the application twice. As a result, it will take twice as long to render the application with the image than if you had only the blue background.

### Graphics Rendering Tiers

It may be very difficult to predict the hardware configuration that your application will be running on. However, you might want to consider a design that allows your application to seamlessly switch features when running on different hardware, so that it can take full advantage of each different hardware configuration.

To achieve this, WPF provides functionality to determine the graphics capability of a system at runtime. Graphics capability is determined by categorizing the video card as one of three rendering capability tiers. WPF exposes an API that allows an application to query the rendering capability tier. Your application can then take different code paths at run time depending on the rendering tier supported by the hardware.

The features of the graphics hardware that most impact the rendering tier levels are:

- **Video RAM** The amount of video memory on the graphics hardware determines the size and number of buffers that can be used for compositing graphics.
- **Pixel Shader** A pixel shader is a graphics processing function that calculates effects on a per-pixel basis. Depending on the resolution of the displayed graphics, there could be several million pixels that need to be processed for each display frame.
- **Vertex Shader** A vertex shader is a graphics processing function that performs mathematical operations on the vertex data of the object.

- **Multitexture Support** Multitexture support refers to the ability to apply two or more distinct textures during a blending operation on a 3D graphics object. The degree of multitexture support is determined by the number of multitexture units on the graphics hardware.

The pixel shader, vertex shader, and multitexture features are used to define specific DirectX version levels, which, in turn, are used to define the different rendering tiers in WPF.

The features of the graphics hardware determine the rendering capability of a WPF application. The WPF system defines three rendering tiers:

- **Rendering Tier 0** No graphics hardware acceleration. The DirectX version level is less than version 7.0.
- **Rendering Tier 1** Partial graphics hardware acceleration. The DirectX version level is greater than or equal to version 7.0, and **lesser** than version 9.0.
- **Rendering Tier 2** Most graphics features use graphics hardware acceleration. The DirectX version level is greater than or equal to version 9.0.

For more information on WPF rendering tiers, see [Graphics Rendering Tiers](#).

## See also

- [Optimizing WPF Application Performance](#)
- [Planning for Application Performance](#)
- [Layout and Design](#)
- [2D Graphics and Imaging](#)
- [Object Behavior](#)
- [Application Resources](#)
- [Text](#)
- [Data Binding](#)
- [Other Performance Recommendations](#)

# Optimizing Performance: Layout and Design

4/28/2019 • 3 minutes to read • [Edit Online](#)

The design of your WPF application can impact its performance by creating unnecessary overhead in calculating layout and validating object references. The construction of objects, particularly at run time, can affect the performance characteristics of your application.

This topic provides performance recommendations in these areas.

## Layout

The term "layout pass" describes the process of measuring and arranging the members of a [Panel](#)-derived object's collection of children, and then drawing them onscreen. The layout pass is a mathematically-intensive process—the larger the number of children in the collection, the greater the number of calculations required. For example, each time a child [UIElement](#) object in the collection changes its position, it has the potential to trigger a new pass by the layout system. Because of the close relationship between object characteristics and layout behavior, it's important to understand the type of events that can invoke the layout system. Your application will perform better by reducing as much as possible any unnecessary invocations of the layout pass.

The layout system completes two passes for each child member in a collection: a measure pass, and an arrange pass. Each child object provides its own overridden implementation of the [Measure](#) and [Arrange](#) methods in order to provide its own specific layout behavior. At its simplest, layout is a recursive system that leads to an element being sized, positioned, and drawn onscreen.

- A child [UIElement](#) object begins the layout process by first having its core properties measured.
- The object's [FrameworkElement](#) properties that are related to size, such as [Width](#), [Height](#), and [Margin](#), are evaluated.
- [Panel](#)-specific logic is applied, such as the [Dock](#) property of the [DockPanel](#), or the [Orientation](#) property of the [StackPanel](#).
- Content is arranged, or positioned, after all child objects have been measured.
- The collection of child objects is drawn to the screen.

The layout pass process is invoked again if any of the following actions occur:

- A child object is added to the collection.
- A [LayoutTransform](#) is applied to the child object.
- The [UpdateLayout](#) method is called for the child object.
- When a change occurs to the value of a dependency property that is marked with metadata affecting the measure or arrange passes.

### Use the Most Efficient Panel where Possible

The complexity of the layout process is directly based on the layout behavior of the [Panel](#)-derived elements you use. For example, a [Grid](#) or [StackPanel](#) control provides much more functionality than a [Canvas](#) control. The price for this greater increase in functionality is a greater increase in performance costs. However, if you do not require the functionality that a [Grid](#) control provides, you should use the less costly alternatives, such as a [Canvas](#) or a custom panel.

For more information, see [Panels Overview](#).

## Update Rather than Replace a RenderTransform

You may be able to update a [Transform](#) rather than replacing it as the value of a [RenderTransform](#) property. This is particularly true in scenarios that involve animation. By updating an existing [Transform](#), you avoid initiating an unnecessary layout calculation.

## Build Your Tree Top-Down

When a node is added or removed from the logical tree, property invalidations are raised on the node's parent and all its children. As a result, a top-down construction pattern should always be followed to avoid the cost of unnecessary invalidations on nodes that have already been validated. The following table shows the difference in execution speed between building a tree top-down versus bottom-up, where the tree is 150 levels deep with a single [TextBlock](#) and [DockPanel](#) at each level.

ACTION	TREE BUILDING (IN MS)	RENDER—INCLUDES TREE BUILDING (IN MS)
Bottom-up	366	454
Top-down	11	96

The following code example demonstrates how to create a tree top down.

```
private void OnBuildTreeTopDown(object sender, RoutedEventArgs e)
{
    TextBlock textBlock = new TextBlock();
    textBlock.Text = "Default";

    DockPanel parentPanel = new DockPanel();
    DockPanel childPanel;

    myCanvas.Children.Add(parentPanel);
    myCanvas.Children.Add(textBlock);

    for (int i = 0; i < 150; i++)
    {
        textBlock = new TextBlock();
        textBlock.Text = "Default";
        parentPanel.Children.Add(textBlock);

        childPanel = new DockPanel();
        parentPanel.Children.Add(childPanel);
        parentPanel = childPanel;
    }
}
```

```
Private Sub OnBuildTreeTopDown(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim textBlock As New TextBlock()
    textBlock.Text = "Default"

    Dim parentPanel As New DockPanel()
    Dim childPanel As DockPanel

    myCanvas.Children.Add(parentPanel)
    myCanvas.Children.Add(textBlock)

    For i As Integer = 0 To 149
        textBlock = New TextBlock()
        textBlock.Text = "Default"
        parentPanel.Children.Add(textBlock)

        childPanel = New DockPanel()
        parentPanel.Children.Add(childPanel)
        parentPanel = childPanel
    Next i
End Sub
```

For more information on the logical tree, see [Trees in WPF](#).

## See also

- [Optimizing WPF Application Performance](#)
- [Planning for Application Performance](#)
- [Taking Advantage of Hardware](#)
- [2D Graphics and Imaging](#)
- [Object Behavior](#)
- [Application Resources](#)
- [Text](#)
- [Data Binding](#)
- [Other Performance Recommendations](#)
- [Layout](#)

# Optimizing Performance: 2D Graphics and Imaging

8/22/2019 • 6 minutes to read • [Edit Online](#)

WPF provides a wide range of 2D graphics and imaging functionality that can be optimized for your application requirements. This topic provides information about performance optimization in those areas.

## Drawing and Shapes

WPF provides both [Drawing](#) and [Shape](#) objects to represent graphical drawing content. However, [Drawing](#) objects are simpler constructs than [Shape](#) objects and provide better performance characteristics.

A [Shape](#) allows you to draw a graphical shape to the screen. Because they are derived from the [FrameworkElement](#) class, [Shape](#) objects can be used inside panels and most controls.

WPF offers several layers of access to graphics and rendering services. At the top layer, [Shape](#) objects are easy to use and provide many useful features, such as layout and event handling. WPF provides a number of ready-to-use shape objects. All shape objects inherit from the [Shape](#) class. Available shape objects include [Ellipse](#), [Line](#), [Path](#), [Polygon](#), [Polyline](#), and [Rectangle](#).

[Drawing](#) objects, on the other hand, do not derive from the [FrameworkElement](#) class and provide a lighter-weight implementation for rendering shapes, images, and text.

There are four types of [Drawing](#) objects:

- [GeometryDrawing](#) Draws a shape.
- [ImageDrawing](#) Draws an image.
- [GlyphRunDrawing](#) Draws text.
- [DrawingGroup](#) Draws other drawings. Use a drawing group to combine other drawings into a single composite drawing.

The [GeometryDrawing](#) object is used to render geometry content. The [Geometry](#) class and the concrete classes which derive from it, such as [CombinedGeometry](#), [EllipseGeometry](#), and [PathGeometry](#), provide a means for rendering 2D graphics, as well as providing hit-testing and clipping support. Geometry objects can be used to define the region of a control, for example, or to define the clip region to apply to an image. Geometry objects can be simple regions, such as rectangles and circles, or composite regions created from two or more geometry objects. More complex geometric regions can be created by combining [PathSegment](#)-derived objects, such as [ArcSegment](#), [BezierSegment](#), and [QuadraticBezierSegment](#).

On the surface, the [Geometry](#) class and the [Shape](#) class are quite similar. Both are used in the rendering of 2D graphics and both have similar concrete classes which derive from them, for example, [EllipseGeometry](#) and [Ellipse](#). However, there are important differences between these two sets of classes. For one, the [Geometry](#) class lacks some of the functionality of the [Shape](#) class, such as the ability to draw itself. To draw a geometry object, another class such as [DrawingContext](#), [Drawing](#), or a [Path](#) (it is worth noting that a [Path](#) is a [Shape](#)) must be used to perform the drawing operation. Rendering properties such as fill, stroke, and the stroke thickness are on the class which draws the geometry object, while a shape object contains these properties. One way to think of this difference is that a geometry object defines a region, a circle for example, while a shape object defines a region, defines how that region is filled and outlined, and participates in the layout system.

Since [Shape](#) objects derive from the [FrameworkElement](#) class, using them can add significantly more memory consumption in your application. If you really do not need the [FrameworkElement](#) features for your graphical

content, consider using the lighter-weight [Drawing](#) objects.

For more information on [Drawing](#) objects, see [Drawing Objects Overview](#).

## StreamGeometry Objects

The [StreamGeometry](#) object is a lightweight alternative to [PathGeometry](#) for creating geometric shapes. Use a [StreamGeometry](#) when you need to describe a complex geometry. [StreamGeometry](#) is optimized for handling many [PathGeometry](#) objects and performs better when compared to using many individual [PathGeometry](#) objects.

The following example uses attribute syntax to create a triangular [StreamGeometry](#) in XAML.

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel>

        <Path Data="F0 M10,100 L100,100 100,50Z"
              StrokeThickness="1" Stroke="Black"/>

    </StackPanel>
</Page>
```

For more information on [StreamGeometry](#) objects, see [Create a Shape Using a StreamGeometry](#).

## DrawingVisual Objects

The [DrawingVisual](#) object is a lightweight drawing class that is used to render shapes, images, or text. This class is considered lightweight because it does not provide layout or event handling, which improves its performance. For this reason, drawings are ideal for backgrounds and clip art. For more information, see [Using DrawingVisual Objects](#).

## Images

WPF imaging provides a significant improvement over the imaging capabilities in previous versions of Windows. Imaging capabilities, such as displaying a bitmap or using an image on a common control were primarily handled by the Microsoft Windows Graphics Device Interface (GDI) or Microsoft Windows GDI+ application programming interface (API). These API provided baseline imaging functionality, but lacked features such as support for codec extensibility and high fidelity image support. WPF Imaging API have been redesigned to overcome the shortcomings of GDI and GDI+ and provide a new set of API to display and use images within your applications.

When using images, consider the following recommendations for gaining better performance:

- If your application requires you to display thumbnail images, consider creating a reduced-sized version of the image. By default, WPF loads your image and decodes it to its full size. If you only want a thumbnail version of the image, WPF unnecessary decodes the image to its full-size and then scales it down to a thumbnail size. To avoid this unnecessary overhead, you can either request WPF to decode the image to a thumbnail size, or request WPF to load a thumbnail size image.
- Always decode the image to desired size and not to the default size. As mentioned above, request WPF to decode your image to a desired size and not the default full size. You will reduce not only your application's working set, but execution speed as well.
- If possible, combine the images into a single image, such as a film strip composed of multiple images.
- For more information, see [Imaging Overview](#).

## BitmapScalingMode

When animating the scale of any bitmap, the default high-quality image re-sampling algorithm can sometimes consume sufficient system resources to cause frame rate degradation, effectively causing animations to stutter. By setting the [BitmapScalingMode](#) property of the [RenderOptions](#) object to [LowQuality](#) you can create a smoother animation when scaling a bitmap. [LowQuality](#) mode tells the WPF rendering engine to switch from a quality-optimized algorithm to a speed-optimized algorithm when processing images.

The following example shows how to set the [BitmapScalingMode](#) for an image object.

```
// Set the bitmap scaling mode for the image to render faster.  
RenderOptions.SetBitmapScalingMode(MyImage, BitmapScalingMode.LowQuality);
```

```
' Set the bitmap scaling mode for the image to render faster.  
RenderOptions.SetBitmapScalingMode(MyImage, BitmapScalingMode.LowQuality)
```

## CachingHint

By default, WPF does not cache the rendered contents of [TileBrush](#) objects, such as [DrawingBrush](#) and [VisualBrush](#). In static scenarios where neither the contents nor use of the [TileBrush](#) in the scene is changing, this makes sense, since it conserves video memory. It does not make as much sense when a [TileBrush](#) with static content is used in a non-static way—for example, when a static [DrawingBrush](#) or [VisualBrush](#) is mapped to the surface of a rotating 3D object. The default behavior of WPF is to re-render the entire content of the [DrawingBrush](#) or [VisualBrush](#) for every frame, even though the content is unchanging.

By setting the [CachingHint](#) property of the [RenderOptions](#) object to [Cache](#) you can increase performance by using cached versions of the tiled brush objects.

The [CacheInvalidationThresholdMinimum](#) and [CacheInvalidationThresholdMaximum](#) property values are relative size values that determine when the [TileBrush](#) object should be regenerated due to changes in scale. For example, by setting the [CacheInvalidationThresholdMaximum](#) property to 2.0, the cache for the [TileBrush](#) only needs to be regenerated when its size exceeds twice the size of the current cache.

The following example shows how to use the caching hint option for a [DrawingBrush](#).

```
DrawingBrush drawingBrush = new DrawingBrush();  
  
// Set the caching hint option for the brush.  
RenderOptions.SetCachingHint(drawingBrush, CachingHint.Cache);  
  
// Set the minimum and maximum relative sizes for regenerating the tiled brush.  
// The tiled brush will be regenerated and re-cached when its size is  
// 0.5x or 2x of the current cached size.  
RenderOptions.SetCacheInvalidationThresholdMinimum(drawingBrush, 0.5);  
RenderOptions.SetCacheInvalidationThresholdMaximum(drawingBrush, 2.0);
```

```
Dim drawingBrush As New DrawingBrush()  
  
' Set the caching hint option for the brush.  
RenderOptions.SetCachingHint(drawingBrush, CachingHint.Cache)  
  
' Set the minimum and maximum relative sizes for regenerating the tiled brush.  
' The tiled brush will be regenerated and re-cached when its size is  
' 0.5x or 2x of the current cached size.  
RenderOptions.SetCacheInvalidationThresholdMinimum(drawingBrush, 0.5)  
RenderOptions.SetCacheInvalidationThresholdMaximum(drawingBrush, 2.0)
```

## See also

- [Optimizing WPF Application Performance](#)
- [Planning for Application Performance](#)
- [Taking Advantage of Hardware](#)
- [Layout and Design](#)
- [Object Behavior](#)
- [Application Resources](#)
- [Text](#)
- [Data Binding](#)
- [Other Performance Recommendations](#)
- [Animation Tips and Tricks](#)

# Optimizing Performance: Object Behavior

7/23/2019 • 7 minutes to read • [Edit Online](#)

Understanding the intrinsic behavior of WPF objects will help you make the right tradeoffs between functionality and performance.

## Not Removing Event Handlers on Objects may Keep Objects Alive

The delegate that an object passes to its event is effectively a reference to that object. Therefore, event handlers can keep objects alive longer than expected. When performing clean up of an object that has registered to listen to an object's event, it is essential to remove that delegate before releasing the object. Keeping unneeded objects alive increases the application's memory usage. This is especially true when the object is the root of a logical tree or a visual tree.

WPF introduces a weak event listener pattern for events that can be useful in situations where the object lifetime relationships between source and listener are difficult to keep track of. Some existing WPF events use this pattern. If you are implementing objects with custom events, this pattern may be of use to you. For details, see [Weak Event Patterns](#).

There are several tools, such as the CLR Profiler and the Working Set Viewer, that can provide information on the memory usage of a specified process. The CLR Profiler includes a number of very useful views of the allocation profile, including a histogram of allocated types, allocation and call graphs, a time line showing garbage collections of various generations and the resulting state of the managed heap after those collections, and a call tree showing per-method allocations and assembly loads. For more information, see [NET Framework Developer Center](#).

## Dependency Properties and Objects

In general, accessing a dependency property of a [DependencyObject](#) is not slower than accessing a CLR property. While there is a small performance overhead for setting a property value, getting a value is as fast as getting the value from a CLR property. Offsetting the small performance overhead is the fact that dependency properties support robust features, such as data binding, animation, inheritance, and styling. For more information, see [Dependency Properties Overview](#).

### DependencyProperty Optimizations

You should define dependency properties in your application very carefully. If your [DependencyProperty](#) affects only render type metadata options, rather than other metadata options such as [AffectsMeasure](#), you should mark it as such by overriding its metadata. For more information about overriding or obtaining property metadata, see [Dependency Property Metadata](#).

It may be more efficient to have a property change handler invalidate the measure, arrange, and render passes manually if not all property changes actually affect measure, arrange, and render. For instance, you might decide to re-render a background only when a value is greater than a set limit. In this case, your property change handler would only invalidate render when the value exceeds the set limit.

### Making a DependencyProperty Inheritable is Not Free

By default, registered dependency properties are non-inheritable. However, you can explicitly make any property inheritable. While this is a useful feature, converting a property to be inheritable impacts performance by increasing the length of time for property invalidation.

### Use RegisterClassHandler Carefully

While calling [RegisterClassHandler](#) allows you to save your instance state, it is important to be aware that the handler is called on every instance, which can cause performance problems. Only use [RegisterClassHandler](#) when your application requires that you save your instance state.

### Set the Default Value for a DependencyProperty during Registration

When creating a [DependencyProperty](#) that requires a default value, set the value using the default metadata passed as a parameter to the [Register](#) method of the [DependencyProperty](#). Use this technique rather than setting the property value in a constructor or on each instance of an element.

### Set the PropertyMetadata Value using Register

When creating a [DependencyProperty](#), you have the option of setting the [PropertyMetadata](#) using either the [Register](#) or [OverrideMetadata](#) methods. Although your object could have a static constructor to call [OverrideMetadata](#), this is not the optimal solution and will impact performance. For best performance, set the [PropertyMetadata](#) during the call to [Register](#).

## Freezable Objects

A [Freezable](#) is a special type of object that has two states: unfrozen and frozen. Freezing objects whenever possible improves the performance of your application and reduces its working set. For more information, see [Freezable Objects Overview](#).

Each [Freezable](#) has a [Changed](#) event that is raised whenever it changes. However, change notifications are costly in terms of application performance.

Consider the following example in which each [Rectangle](#) uses the same [Brush](#) object:

```
rectangle_1.Fill = myBrush;
rectangle_2.Fill = myBrush;
rectangle_3.Fill = myBrush;
// ...
rectangle_10.Fill = myBrush;
```

```
rectangle_1.Fill = myBrush
rectangle_2.Fill = myBrush
rectangle_3.Fill = myBrush
' ...
rectangle_10.Fill = myBrush
```

By default, WPF provides an event handler for the [SolidColorBrush](#) object's [Changed](#) event in order to invalidate the [Rectangle](#) object's [Fill](#) property. In this case, each time the [SolidColorBrush](#) has to fire its [Changed](#) event it is required to invoke the callback function for each [Rectangle](#)—the accumulation of these callback function invocations impose a significant performance penalty. In addition, it is very performance intensive to add and remove handlers at this point since the application would have to traverse the entire list to do so. If your application scenario never changes the [SolidColorBrush](#), you will be paying the cost of maintaining [Changed](#) event handlers unnecessarily.

Freezing a [Freezable](#) can improve its performance, because it no longer needs to expend resources on maintaining change notifications. The table below shows the size of a simple [SolidColorBrush](#) when its [IsFrozen](#) property is set to `true`, compared to when it is not. This assumes applying one brush to the [Fill](#) property of ten [Rectangle](#) objects.

STATE	SIZE
Frozen <a href="#">SolidColorBrush</a>	212 Bytes

STATE	SIZE
Non-frozen <a href="#">SolidColorBrush</a>	972 Bytes

The following code sample demonstrates this concept:

```
Brush frozenBrush = new SolidColorBrush(Colors.Blue);
frozenBrush.Freeze();
Brush nonFrozenBrush = new SolidColorBrush(Colors.Blue);

for (int i = 0; i < 10; i++)
{
    // Create a Rectangle using a non-frozed Brush.
    Rectangle rectangleNonFrozen = new Rectangle();
    rectangleNonFrozen.Fill = nonFrozenBrush;

    // Create a Rectangle using a frozed Brush.
    Rectangle rectangleFrozen = new Rectangle();
    rectangleFrozen.Fill = frozenBrush;
}
```

```
Dim frozenBrush As Brush = New SolidColorBrush(Colors.Blue)
frozenBrush.Freeze()
Dim nonFrozenBrush As Brush = New SolidColorBrush(Colors.Blue)

For i As Integer = 0 To 9
    ' Create a Rectangle using a non-frozed Brush.
    Dim rectangleNonFrozen As New Rectangle()
    rectangleNonFrozen.Fill = nonFrozenBrush

    ' Create a Rectangle using a frozed Brush.
    Dim rectangleFrozen As New Rectangle()
    rectangleFrozen.Fill = frozenBrush
Next i
```

### Changed Handlers on Unfrozen Freezables may Keep Objects Alive

The delegate that an object passes to a [Freezable](#) object's [Changed](#) event is effectively a reference to that object. Therefore, [Changed](#) event handlers can keep objects alive longer than expected. When performing clean up of an object that has registered to listen to a [Freezable](#) object's [Changed](#) event, it is essential to remove that delegate before releasing the object.

WPF also hooks up [Changed](#) events internally. For example, all dependency properties which take [Freezable](#) as a value will listen to [Changed](#) events automatically. The [Fill](#) property, which takes a [Brush](#), illustrates this concept.

```
Brush myBrush = new SolidColorBrush(Colors.Red);
Rectangle myRectangle = new Rectangle();
myRectangle.Fill = myBrush;
```

```
Dim myBrush As Brush = New SolidColorBrush(Colors.Red)
Dim myRectangle As New Rectangle()
myRectangle.Fill = myBrush
```

On the assignment of `myBrush` to `myRectangle.Fill`, a delegate pointing back to the `Rectangle` object will be added to the [SolidColorBrush](#) object's [Changed](#) event. This means the following code does not actually make `myRect` eligible for garbage collection:

```
myRectangle = null;
```

```
myRectangle = Nothing
```

In this case `myBrush` is still keeping `myRectangle` alive and will call back to it when it fires its [Changed](#) event. Note that assigning `myBrush` to the [Fill](#) property of a new [Rectangle](#) will simply add another event handler to `myBrush`.

The recommended way to clean up these types of objects is to remove the [Brush](#) from the [Fill](#) property, which will in turn remove the [Changed](#) event handler.

```
myRectangle.Fill = null;  
myRectangle = null;
```

```
myRectangle.Fill = Nothing  
myRectangle = Nothing
```

## User Interface Virtualization

WPF also provides a variation of the [StackPanel](#) element that automatically "virtualizes" data-bound child content. In this context, the word virtualize refers to a technique by which a subset of objects are generated from a larger number of data items based upon which items are visible on-screen. It is intensive, both in terms of memory and processor, to generate a large number of UI elements when only a few may be on the screen at a given time. [VirtualizingStackPanel](#) (through functionality provided by [VirtualizingPanel](#)) calculates visible items and works with the [ItemContainerGenerator](#) from an [ItemsControl](#) (such as [ListBox](#) or [ListView](#)) to only create elements for visible items.

As a performance optimization, visual objects for these items are only generated or kept alive if they are visible on the screen. When they are no longer in the viewable area of the control, the visual objects may be removed. This is not to be confused with data virtualization, where data objects are not all present in the local collection—rather streamed in as needed.

The table below shows the elapsed time adding and rendering 5000 [TextBlock](#) elements to a [StackPanel](#) and a [VirtualizingStackPanel](#). In this scenario, the measurements represent the time between attaching a text string to the [ItemsSource](#) property of an [ItemsControl](#) object to the time when the panel elements display the text string.

HOST PANEL	RENDER TIME (MS)
StackPanel	3210
VirtualizingStackPanel	46

## See also

- [Optimizing WPF Application Performance](#)
- [Planning for Application Performance](#)
- [Taking Advantage of Hardware](#)
- [Layout and Design](#)
- [2D Graphics and Imaging](#)
- [Application Resources](#)

- [Text](#)
- [Data Binding](#)
- [Other Performance Recommendations](#)

# Optimizing Performance: Application Resources

11/3/2019 • 2 minutes to read • [Edit Online](#)

WPF allows you to share application resources so that you can support a consistent look or behavior across similar-typed elements. This topic provides a few recommendations in this area that can help you improve the performance of your applications.

For more information on resources, see [XAML Resources](#).

## Sharing resources

If your application uses custom controls and defines resources in a [ResourceDictionary](#) (or XAML Resources node), it is recommended that you either define the resources at the [Application](#) or [Window](#) object level, or define them in the default theme for the custom controls. Defining resources in a custom control's [ResourceDictionary](#) imposes a performance impact for every instance of that control. For example, if you have performance-intensive brush operations defined as part of the resource definition of a custom control and many instances of the custom control, the application's working set will increase significantly.

To illustrate this point, consider the following. Let's say you are developing a card game using WPF. For most card games, you need 52 cards with 52 different faces. You decide to implement a card custom control and you define 52 brushes (each representing a card face) in the resources of your card custom control. In your main application, you initially create 52 instances of this card custom control. Each instance of the card custom control generates 52 instances of [Brush](#) objects, which gives you a total of  $52 * 52$  [Brush](#) objects in your application. By moving the brushes out of the card custom control resources to the [Application](#) or [Window](#) object level, or defining them in the default theme for the custom control, you reduce the working set of the application, since you are now sharing the 52 brushes among 52 instances of the card control.

## Sharing a Brush without Copying

If you have multiple elements using the same [Brush](#) object, define the brush as a resource and reference it, rather than defining the brush inline in XAML. This method will create one instance and reuse it, whereas defining brushes inline in XAML creates a new instance for each element.

The following markup sample illustrates this point:

```

<StackPanel.Resources>
    <LinearGradientBrush x:Key="myBrush" StartPoint="0,0.5" EndPoint="1,0.5" Opacity="0.5">
        <LinearGradientBrush.GradientStops>
            <GradientStopCollection>
                <GradientStop Color="GoldenRod" Offset="0" />
                <GradientStop Color="White" Offset="1" />
            </GradientStopCollection>
        </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
</StackPanel.Resources>

<!-- Non-shared Brush object. -->
<Label>
    Label 1
    <Label.Background>
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5" Opacity="0.5">
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Color="GoldenRod" Offset="0" />
                    <GradientStop Color="White" Offset="1" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Label.Background>
</Label>

<!-- Shared Brush object. -->
<Label Background="{StaticResource myBrush}">Label 2</Label>
<Label Background="{StaticResource myBrush}">Label 3</Label>

```

## Use Static Resources when Possible

A static resource provides a value for any XAML property attribute by looking up a reference to an already defined resource. Lookup behavior for that resource is analogous to compile-time lookup.

A dynamic resource, on the other hand, will create a temporary expression during the initial compilation and thus defer lookup for resources until the requested resource value is actually required in order to construct an object. Lookup behavior for that resource is analogous to run-time lookup, which imposes a performance impact. Use static resources whenever possible in your application, using dynamic resources only when necessary.

The following markup sample shows the use of both types of resources:

```

<StackPanel.Resources>
    <SolidColorBrush x:Key="myBrush" Color="Teal"/>
</StackPanel.Resources>

<!-- StaticResource reference -->
<Label Foreground="{StaticResource myBrush}">Label 1</Label>

<!-- DynamicResource reference -->
<Label Foreground="{DynamicResource {x:Static SystemColors.ControlBrushKey}}">Label 2</Label>

```

## See also

- [Optimizing WPF Application Performance](#)
- [Planning for Application Performance](#)
- [Taking Advantage of Hardware](#)
- [Layout and Design](#)
- [2D Graphics and Imaging](#)

- Object Behavior
- Text
- Data Binding
- Other Performance Recommendations

# Optimizing Performance: Text

8/22/2019 • 9 minutes to read • [Edit Online](#)

WPF includes support for the presentation of text content through the use of feature-rich user interface (UI) controls. In general you can divide text rendering in three layers:

1. Using the [Glyphs](#) and [GlyphRun](#) objects directly.
2. Using the [FormattedText](#) object.
3. Using high-level controls, such as the [TextBlock](#) and [FlowDocument](#) objects.

This topic provides text rendering performance recommendations.

## Rendering Text at the Glyph Level

Windows Presentation Foundation (WPF) provides advanced text support including glyph-level markup with direct access to [Glyphs](#) for customers who want to intercept and persist text after formatting. These features provide critical support for the different text rendering requirements in each of the following scenarios.

- Screen display of fixed-format documents.
- Print scenarios.
  - Extensible Application Markup Language (XAML) as a device printer language.
  - Microsoft XPS Document Writer.
  - Previous printer drivers, output from Win32 applications to the fixed format.
  - Print spool format.
- Fixed-format document representation, including clients for previous versions of Windows and other computing devices.

### NOTE

[Glyphs](#) and [GlyphRun](#) are designed for fixed-format document presentation and print scenarios. Windows Presentation Foundation (WPF) provides several elements for general layout and user interface (UI) scenarios such as [Label](#) and [TextBlock](#). For more information on layout and UI scenarios, see the [Typography in WPF](#).

The following examples show how to define properties for a [Glyphs](#) object in Extensible Application Markup Language (XAML). The [Glyphs](#) object represents the output of a [GlyphRun](#) in XAML. The examples assume that the Arial, Courier New, and Times New Roman fonts are installed in the **C:\WINDOWS\Fonts** folder on the local computer.

```

<!-- The example shows how to use a Glyphs object. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>

    <StackPanel Background="PowderBlue">

        <Glyphs
            FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
            FontRenderingEmSize = "100"
            StyleSimulations   = "BoldSimulation"
            UnicodeString      = "Hello World!"
            Fill               = "Black"
            OriginX            = "100"
            OriginY            = "200"
        />

    </StackPanel>
</Page>

```

## Using DrawGlyphRun

If you have custom control and you want to render glyphs, use the [DrawGlyphRun](#) method.

WPF also provides lower-level services for custom text formatting through the use of the [FormattedText](#) object. The most efficient way of rendering text in Windows Presentation Foundation (WPF) is by generating text content at the glyph level using [Glyphs](#) and [GlyphRun](#). However, the cost of this efficiency is the loss of easy to use rich text formatting, which are built-in features of Windows Presentation Foundation (WPF) controls, such as [TextBlock](#) and [FlowDocument](#).

## FormattedText Object

The [FormattedText](#) object allows you to draw multi-line text, in which each character in the text can be individually formatted. For more information, see [Drawing Formatted Text](#).

To create formatted text, call the [FormattedText](#) constructor to create a [FormattedText](#) object. Once you have created the initial formatted text string, you can apply a range of formatting styles. If your application wants to implement its own layout, then the [FormattedText](#) object is better choice than using a control, such as [TextBlock](#). For more information on the [FormattedText](#) object, see [Drawing Formatted Text](#).

The [FormattedText](#) object provides low-level text formatting capability. You can apply multiple formatting styles to one or more characters. For example, you could call both the [SetFontStyle](#) and [SetForegroundBrush](#) methods to change the formatting of the first five characters in the text.

The following code example creates a [FormattedText](#) object and renders it.

```
protected override void OnRender(DrawingContext drawingContext)
{
    string testString = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor";

    // Create the initial formatted text string.
    FormattedText formattedText = new FormattedText(
        testString,
        CultureInfo.GetCultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface("Verdana"),
        32,
        Brushes.Black);

    // Set a maximum width and height. If the text overflows these values, an ellipsis "..." appears.
    formattedText.MaxTextWidth = 300;
    formattedText.MaxTextHeight = 240;

    // Use a larger font size beginning at the first (zero-based) character and continuing for 5 characters.
    // The font size is calculated in terms of points -- not as device-independent pixels.
    formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5);

    // Use a Bold font weight beginning at the 6th character and continuing for 11 characters.
    formattedText.SetFontWeight(FontWeights.Bold, 6, 11);

    // Use a linear gradient brush beginning at the 6th character and continuing for 11 characters.
    formattedText.SetForegroundBrush(
        new LinearGradientBrush(
            Colors.Orange,
            Colors.Teal,
            90.0),
        6, 11);

    // Use an Italic font style beginning at the 28th character and continuing for 28 characters.
    formattedTextSetFontStyle(FontStyles.Italic, 28, 28);

    // Draw the formatted text string to the DrawingContext of the control.
    drawingContext.DrawText(formattedText, new Point(10, 0));
}
```

```

Protected Overrides Sub OnRender(ByVal drawingContext As DrawingContext)
    Dim testString As String = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor"

    ' Create the initial formatted text string.
    Dim formattedText As New FormattedText(testString, CultureInfo.GetCultureInfo("en-us"),
    FlowDirection.LeftToRight, New Typeface("Verdana"), 32, Brushes.Black)

    ' Set a maximum width and height. If the text overflows these values, an ellipsis "..." appears.
    formattedText.MaxTextWidth = 300
    formattedText.MaxTextHeight = 240

    ' Use a larger font size beginning at the first (zero-based) character and continuing for 5 characters.
    ' The font size is calculated in terms of points -- not as device-independent pixels.
    formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5)

    ' Use a Bold font weight beginning at the 6th character and continuing for 11 characters.
    formattedText.SetFontWeight(FontWeights.Bold, 6, 11)

    ' Use a linear gradient brush beginning at the 6th character and continuing for 11 characters.
    formattedText.SetForegroundBrush(New LinearGradientBrush(Colors.Orange, Colors.Teal, 90.0), 6, 11)

    ' Use an Italic font style beginning at the 28th character and continuing for 28 characters.
    formattedTextSetFontStyle(FontStyles.Italic, 28, 28)

    ' Draw the formatted text string to the DrawingContext of the control.
    drawingContext.DrawText(formattedText, New Point(10, 0))
End Sub

```

## FlowDocument, TextBlock, and Label Controls

WPF includes multiple controls for drawing text to the screen. Each control is targeted to a different scenario and has its own list of features and limitations.

### **FlowDocument Impacts Performance More than TextBlock or Label**

In general, the [TextBlock](#) element should be used when limited text support is required, such as a brief sentence in a user interface (UI). [Label](#) can be used when minimal text support is required. The [FlowDocument](#) element is a container for re-flowable documents that support rich presentation of content, and therefore, has a greater performance impact than using the [TextBlock](#) or [Label](#) controls.

For more information on [FlowDocument](#), see [Flow Document Overview](#).

### **Avoid Using TextBlock in FlowDocument**

The [TextBlock](#) element is derived from [UIElement](#). The [Run](#) element is derived from [TextElement](#), which is less costly to use than a [UIElement](#)-derived object. When possible, use [Run](#) rather than [TextBlock](#) for displaying text content in a [FlowDocument](#).

The following markup sample illustrates two ways of setting text content within a [FlowDocument](#):

```

<FlowDocument>

    <!-- Text content within a Run (more efficient). -->
    <Paragraph>
        <Run>Line one</Run>
    </Paragraph>

    <!-- Text content within a TextBlock (less efficient). -->
    <Paragraph>
        <TextBlock>Line two</TextBlock>
    </Paragraph>

</FlowDocument>

```

## Avoid Using Run to Set Text Properties

In general, using a [Run](#) within a [TextBlock](#) is more performance intensive than not using an explicit [Run](#) object at all. If you are using a [Run](#) in order to set text properties, set those properties directly on the [TextBlock](#) instead.

The following markup sample illustrates these two ways of setting a text property, in this case, the [FontWeight](#) property:

```

<!-- Run is used to set text properties. -->
<TextBlock>
    <Run FontWeight="Bold">Hello, world</Run>
</TextBlock>

<!-- TextBlock is used to set text properties, which is more efficient. -->
<TextBlock FontWeight="Bold">
    Hello, world
</TextBlock>

```

The following table shows the cost of displaying 1000 [TextBlock](#) objects with and without an explicit [Run](#).

TEXTBLOCK TYPE	CREATION TIME (MS)	RENDER TIME (MS)
Run setting text properties	146	540
TextBlock setting text properties	43	453

## Avoid Databinding to the Label.Content Property

Imagine a scenario where you have a [Label](#) object that is updated frequently from a [String](#) source. When data binding the [Label](#) element's [Content](#) property to the [String](#) source object, you may experience poor performance. Each time the source [String](#) is updated, the old [String](#) object is discarded and a new [String](#) is recreated—because a [String](#) object is immutable, it cannot be modified. This, in turn, causes the [ContentPresenter](#) of the [Label](#) object to discard its old content and regenerate the new content to display the new [String](#).

The solution to this problem is simple. If the [Label](#) is not set to a custom [ContentTemplate](#) value, replace the [Label](#) with a [TextBlock](#) and data bind its [Text](#) property to the source string.

DATA BOUND PROPERTY	UPDATE TIME (MS)
Label.Content	835
TextBlock.Text	242

# Hyperlink

The [Hyperlink](#) object is an inline-level flow content element that allows you to host hyperlinks within the flow content.

## Combine Hyperlinks in One TextBlock Object

You can optimize the use of multiple [Hyperlink](#) elements by grouping them together within the same [TextBlock](#). This helps to minimize the number of objects you create in your application. For example, you may want to display multiple hyperlinks, such as the following:

MSN Home | My MSN

The following markup example shows multiple [TextBlock](#) elements used to display the hyperlinks:

```
<!-- Hyperlinks in separate TextBlocks. -->
<TextBlock>
    <Hyperlink TextDecorations="None" NavigateUri="http://www.msn.com">MSN Home</Hyperlink>
</TextBlock>

<TextBlock Text=" | "/>

<TextBlock>
    <Hyperlink TextDecorations="None" NavigateUri="http://my.msn.com">My MSN</Hyperlink>
</TextBlock>
```

The following markup example shows a more efficient way of displaying the hyperlinks, this time, using a single [TextBlock](#):

```
<!-- Hyperlinks combined in the same TextBlock. -->
<TextBlock>
    <Hyperlink TextDecorations="None" NavigateUri="http://www.msn.com">MSN Home</Hyperlink>

    <Run Text=" | " />

    <Hyperlink TextDecorations="None" NavigateUri="http://my.msn.com">My MSN</Hyperlink>
</TextBlock>
```

## Showing Underlines on Hyperlinks Only on MouseEnter Events

A [TextDecoration](#) object is a visual ornamentation that you can add to text; however, it can be performance intensive to instantiate. If you make extensive use of [Hyperlink](#) elements, consider showing an underline only when triggering an event, such as the [MouseEnter](#) event. For more information, see [Specify Whether a Hyperlink is Underlined](#).

The following image shows how the [MouseEnter](#) event triggers the underlined hyperlink:



The following markup sample shows a [Hyperlink](#) defined with and without an underline:

```

<!-- Hyperlink with default underline. -->
<Hyperlink NavigateUri="http://www.msn.com">
    MSN Home
</Hyperlink>

<Run Text=" | " />

<!-- Hyperlink with no underline. -->
<Hyperlink Name="myHyperlink" TextDecorations="None"
    MouseEnter="OnMouseEnter"
    MouseLeave="OnMouseLeave"
    NavigateUri="http://www.msn.com">
    My MSN
</Hyperlink>

```

The following table shows the performance cost of displaying 1000 [Hyperlink](#) elements with and without an underline.

HYPERLINK	CREATION TIME (MS)	RENDER TIME (MS)
With underline	289	1130
Without underline	299	776

## Text Formatting Features

WPF provides rich text formatting services, such as automatic hyphenations. These services may impact application performance and should only be used when needed.

### Avoid Unnecessary Use of Hyphenation

Automatic hyphenation finds hyphen breakpoints for lines of text, and allows additional break positions for lines in [TextBlock](#) and [FlowDocument](#) objects. By default, the automatic hyphenation feature is disabled in these objects. You can enable this feature by setting the object's `IsHyphenationEnabled` property to `true`. However, enabling this feature causes WPF to initiate Component Object Model (COM) interoperability, which can impact application performance. It is recommended that you do not use automatic hyphenation unless you need it.

### Use Figures Carefully

A [Figure](#) element represents a portion of flow content that can be absolutely-positioned within a page of content. In some cases, a [Figure](#) may cause an entire page to automatically reformat if its position collides with content that has already been laid-out. You can minimize the possibility of unnecessary reformatting by either grouping [Figure](#) elements next to each other, or declaring them near the top of content in a fixed page size scenario.

### Optimal Paragraph

The optimal paragraph feature of the [FlowDocument](#) object lays out paragraphs so that white space is distributed as evenly as possible. By default, the optimal paragraph feature is disabled. You can enable this feature by setting the object's `IsOptimalParagraphEnabled` property to `true`. However, enabling this feature impacts application performance. It is recommended that you do not use the optimal paragraph feature unless you need it.

## See also

- [Optimizing WPF Application Performance](#)
- [Planning for Application Performance](#)
- [Taking Advantage of Hardware](#)

- Layout and Design
- 2D Graphics and Imaging
- Object Behavior
- Application Resources
- Data Binding
- Other Performance Recommendations

# Optimizing Performance: Data Binding

11/7/2019 • 5 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) data binding provides a simple and consistent way for applications to present and interact with data. Elements can be bound to data from a variety of data sources in the form of CLR objects and XML.

This topic provides data binding performance recommendations.

## How Data Binding References are Resolved

Before discussing data binding performance issues, it is worthwhile to explore how the Windows Presentation Foundation (WPF) data binding engine resolves object references for binding.

The source of a Windows Presentation Foundation (WPF) data binding can be any CLR object. You can bind to properties, sub-properties, or indexers of a CLR object. The binding references are resolved by using either Microsoft .NET Framework reflection or an [ICustomTypeDescriptor](#). Here are three methods for resolving object references for binding.

The first method involves using reflection. In this case, the  [PropertyInfo](#) object is used to discover the attributes of the property and provides access to property metadata. When using the [ICustomTypeDescriptor](#) interface, the data binding engine uses this interface to access the property values. The [ICustomTypeDescriptor](#) interface is especially useful in cases where the object does not have a static set of properties.

Property change notifications can be provided either by implementing the [INotifyPropertyChanged](#) interface or by using the change notifications associated with the [PropertyDescriptor](#). However, the preferred strategy for implementing property change notifications is to use [INotifyPropertyChanged](#).

If the source object is a CLR object and the source property is a CLR property, the Windows Presentation Foundation (WPF) data binding engine has to first use reflection on the source object to get the [PropertyDescriptor](#), and then query for a [PropertyDescriptor](#). This sequence of reflection operations is potentially very time-consuming from a performance perspective.

The second method for resolving object references involves a CLR source object that implements the [INotifyPropertyChanged](#) interface, and a source property that is a CLR property. In this case, the data binding engine uses reflection directly on the source type and gets the required property. This is still not the optimal method, but it will cost less in working set requirements than the first method.

The third method for resolving object references involves a source object that is a [DependencyObject](#) and a source property that is a [DependencyProperty](#). In this case, the data binding engine does not need to use reflection. Instead, the property engine and the data binding engine together resolve the property reference independently. This is the optimal method for resolving object references used for data binding.

The table below compares the speed of data binding the [Text](#) property of one thousand [TextBlock](#) elements using these three methods.

BINDING THE TEXT PROPERTY OF A TEXTBLOCK	BINDING TIME (MS)	RENDER TIME -- INCLUDES BINDING (MS)
To a property of a CLR object	115	314

BINDING THE TEXT PROPERTY OF A TEXTBLOCK	BINDING TIME (MS)	RENDER TIME -- INCLUDES BINDING (MS)
To a property of a CLR object which implements <a href="#">INotifyPropertyChanged</a>	115	305
To a <a href="#">DependencyProperty</a> of a <a href="#">DependencyObject</a> .	90	263

## Binding to Large CLR Objects

There is a significant performance impact when you data bind to a single CLR object with thousands of properties. You can minimize this impact by dividing the single object into multiple CLR objects with fewer properties. The table shows the binding and rendering times for data binding to a single large CLR object versus multiple smaller objects.

DATA BINDING 1000 TEXTBLOCK OBJECTS	BINDING TIME (MS)	RENDER TIME -- INCLUDES BINDING (MS)
To a CLR object with 1000 properties	950	1200
To 1000 CLR objects with one property	115	314

## Binding to an ItemsSource

Consider a scenario in which you have a CLR [List<T>](#) object that holds a list of employees that you want to display in a [ListBox](#). To create a correspondence between these two objects, you would bind your employee list to the [ItemsSource](#) property of the [ListBox](#). However, suppose you have a new employee joining your group. You might think that in order to insert this new person into your bound [ListBox](#) values, you would simply add this person to your employee list and expect this change to be recognized by the data binding engine automatically. That assumption would prove false; in actuality, the change will not be reflected in the [ListBox](#) automatically. This is because the CLR [List<T>](#) object does not automatically raise a collection changed event. In order to get the [ListBox](#) to pick up the changes, you would have to recreate your list of employees and re-attach it to the [ItemsSource](#) property of the [ListBox](#). While this solution works, it introduces a huge performance impact. Each time you reassign the [ItemsSource](#) of [ListBox](#) to a new object, the [ListBox](#) first throws away its previous items and regenerates its entire list. The performance impact is magnified if your [ListBox](#) maps to a complex [DataTemplate](#).

A very efficient solution to this problem is to make your employee list an [ObservableCollection<T>](#). An [ObservableCollection<T>](#) object raises a change notification which the data binding engine can receive. The event adds or removes an item from an [ItemsControl](#) without the need to regenerate the entire list.

The table below shows the time it takes to update the [ListBox](#) (with UI virtualization turned off) when one item is added. The number in the first row represents the elapsed time when the CLR [List<T>](#) object is bound to [ListBox](#) element's [ItemsSource](#). The number in the second row represents the elapsed time when an [ObservableCollection<T>](#) is bound to the [ListBox](#) element's [ItemsSource](#). Note the significant time savings using the [ObservableCollection<T>](#) data binding strategy.

DATA BINDING THE ITEMSOURCE	UPDATE TIME FOR 1 ITEM (MS)
To a CLR <a href="#">List&lt;T&gt;</a> object	1656
To an <a href="#">ObservableCollection&lt;T&gt;</a>	20

## Bind `IList` to `ItemsControl` not `IEnumerable`

If you have a choice between binding an `IList<T>` or an `IEnumerable` to an `ItemsControl` object, choose the `IList<T>` object. Binding `IEnumerable` to an `ItemsControl` forces WPF to create a wrapper `IList<T>` object, which means your performance is impacted by the unnecessary overhead of a second object.

## Do not Convert CLR objects to XML Just for Data Binding.

WPF allows you to data bind to XML content; however, data binding to XML content is slower than data binding to CLR objects. Do not convert CLR object data to XML if the only purpose is for data binding.

## See also

- [Optimizing WPF Application Performance](#)
- [Planning for Application Performance](#)
- [Taking Advantage of Hardware](#)
- [Layout and Design](#)
- [2D Graphics and Imaging](#)
- [Object Behavior](#)
- [Application Resources](#)
- [Text](#)
- [Other Performance Recommendations](#)
- [Data Binding Overview](#)
- [Walkthrough: Caching Application Data in a WPF Application](#)

# Optimizing performance: Controls

11/3/2019 • 4 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) includes many of the common user-interface (UI) components that are used in most Windows applications. This topic contains techniques for improving the performance of your UI.

## Displaying large data sets

WPF controls such as the [ListView](#) and [ComboBox](#) are used to display lists of items in an application. If the list to display is large, the application's performance can be affected. This is because the standard layout system creates a layout container for each item associated with the list control, and computes its layout size and position. Typically, you do not have to display all the items at the same time; instead you display a subset, and the user scrolls through the list. In this case, it makes sense to use UI *virtualization*, which means the item container generation and associated layout computation for an item is deferred until the item is visible.

UI Virtualization is an important aspect of list controls. UI virtualization should not be confused with data virtualization. UI virtualization stores only visible items in memory but in a data-binding scenario stores the entire data structure in memory. In contrast, data virtualization stores only the data items that are visible on the screen in memory.

By default, UI virtualization is enabled for the [ListView](#) and [ListBox](#) controls when their list items are bound to data. [TreeView](#) virtualization can be enabled by setting the [VirtualizingStackPanel.IsVirtualizing](#) attached property to `true`. If you want to enable UI virtualization for custom controls that derive from [ItemsControl](#) or existing item controls that use the [StackPanel](#) class, such as [ComboBox](#), you can set the [ItemsPanel](#) to [VirtualizingStackPanel](#) and set [IsVirtualizing](#) to `true`. Unfortunately, you can disable UI virtualization for these controls without realizing it. The following is a list of conditions that disable UI virtualization.

- Item containers are added directly to the [ItemsControl](#). For example, if an application explicitly adds [ListBoxItem](#) objects to a [ListBox](#), the [ListBox](#) does not virtualize the [ListBoxItem](#) objects.
- Item containers in the [ItemsControl](#) are of different types. For example, a [Menu](#) that uses [Separator](#) objects cannot implement item recycling because the [Menu](#) contains objects of type [Separator](#) and [MenuItem](#).
- Setting [CanContentScroll](#) to `false`.
- Setting [IsVirtualizing](#) to `false`.

An important consideration when you virtualize item containers is whether you have additional state information associated with an item container that belongs with the item. In this case, you must save the additional state. For example, you might have an item contained in an [Expander](#) control and the [IsExpanded](#) state is bound to the item's container, and not to the item itself. When the container is reused for a new item, the current value of [IsExpanded](#) is used for the new item. In addition, the old item loses the correct [IsExpanded](#) value.

Currently, no WPF controls offer built-in support for data virtualization.

## Container recycling

An optimization to UI virtualization added in the .NET Framework 3.5 SP1 for controls that inherit from [ItemsControl](#) is *container recycling*, which can also improve scrolling performance. When an [ItemsControl](#) that uses UI virtualization is populated, it creates an item container for each item that scrolls into view and destroys the item container for each item that scrolls out of view. *Container recycling* enables the control to reuse the existing item containers for different data items, so that item containers are not constantly created and destroyed as the

user scrolls the [ItemsControl](#). You can choose to enable item recycling by setting the [VirtualizationMode](#) attached property to [Recycling](#).

Any [ItemsControl](#) that supports virtualization can use container recycling. For an example of how to enable container recycling on a [ListBox](#), see [Improve the Scrolling Performance of a ListBox](#).

## Supporting bidirectional virtualization

[VirtualizingStackPanel](#) offers built-in support for UI virtualization in one direction, either horizontally or vertically. If you want to use bidirectional virtualization for your controls, you must implement a custom panel that extends the [VirtualizingStackPanel](#) class. The [VirtualizingStackPanel](#) class exposes virtual methods such as [OnViewportSizeChanged](#), [LineUp](#), [PageUp](#), and [MouseWheelUp](#). These virtual methods enable you to detect a change in the visible part of a list and handle it accordingly.

## Optimizing templates

The visual tree contains all the visual elements in an application. In addition to the objects directly created, it also contains objects due to template expansion. For example, when you create a [Button](#), you also get [ClassicBorderDecorator](#) and [ContentPresenter](#) objects in the visual tree. If you haven't optimized your control templates, you may be creating a lot of extra unnecessary objects in the visual tree. For more information on the visual tree, see [WPF Graphics Rendering Overview](#).

## Deferred scrolling

By default, when the user drags the thumb on a scrollbar, the content view continuously updates. If scrolling is slow in your control, consider using deferred scrolling. In deferred scrolling, the content is updated only when the user releases the thumb.

To implement deferred scrolling, set the [IsDeferredScrollingEnabled](#) property to `true`.

[IsDeferredScrollingEnabled](#) is an attached property and can be set on [ScrollViewer](#) and any control that has a [ScrollViewer](#) in its control template.

## Controls that implement performance features

The following table lists the common controls for displaying data and their support of performance features. See the previous sections for information on how to enable these features.

CONTROL	VIRTUALIZATION	CONTAINER RECYCLING	DEFERRED SCROLLING
<a href="#">ComboBox</a>	Can be enabled	Can be enabled	Can be enabled
<a href="#">ContextMenu</a>	Can be enabled	Can be enabled	Can be enabled
<a href="#">DocumentViewer</a>	Not available	Not available	Can be enabled
<a href="#">ListBox</a>	Default	Can be enabled	Can be enabled
<a href="#">ListView</a>	Default	Can be enabled	Can be enabled
<a href="#">TreeView</a>	Can be enabled	Can be enabled	Can be enabled
<a href="#">ToolBar</a>	Not available	Not available	Can be enabled

**NOTE**

For an example of how to enable virtualization and container recycling on a [TreeView](#), see [Improve the Performance of a TreeView](#).

## See also

- [Layout](#)
- [Layout and Design](#)
- [Data Binding](#)
- [Controls](#)
- [Styling and Templating](#)
- [Walkthrough: Caching Application Data in a WPF Application](#)

# Optimizing Performance: Other Recommendations

11/12/2019 • 3 minutes to read • [Edit Online](#)

This topic provides performance recommendations in addition to the ones covered by the topics in the [Optimizing WPF Application Performance](#) section.

This topic contains the following sections:

- [Opacity on Brushes Versus Opacity on Elements](#)
- [Navigation to Object](#)
- [Hit Testing on Large 3D Surfaces](#)
- [CompositionTarget.Rendering Event](#)
- [Avoid Using ScrollBarVisibility=Auto](#)
- [Configure Font Cache Service to Reduce Start-up Time](#)

## Opacity on Brushes Versus Opacity on Elements

When you use a [Brush](#) to set the [Fill](#) or [Stroke](#) of an element, it is better to set the [Brush.Opacity](#) value rather than the setting the element's [Opacity](#) property. Modifying an element's [Opacity](#) property can cause WPF to create a temporary surface.

## Navigation to Object

The [NavigationWindow](#) object derives from [Window](#) and extends it with content navigation support, primarily by aggregating [NavigationService](#) and the journal. You can update the client area of [NavigationWindow](#) by specifying either a uniform resource identifier (URI) or an object. The following sample shows both methods:

```
private void buttonGoToUri(object sender, RoutedEventArgs args)
{
    navWindow.Source = new Uri("NewPage.xaml", UriKind.RelativeOrAbsolute);
}

private void buttonGoNewObject(object sender, RoutedEventArgs args)
{
    NewPage nextPage = new NewPage();
    nextPage.InitializeComponent();
    navWindow.Content = nextPage;
}
```

```
Private Sub buttonGoToUri(ByVal sender As Object, ByVal args As RoutedEventArgs)
    navWindow.Source = New Uri("NewPage.xaml", UriKind.RelativeOrAbsolute)
End Sub

Private Sub buttonGoNewObject(ByVal sender As Object, ByVal args As RoutedEventArgs)
    Dim nextPage As New NewPage()
    nextPage.InitializeComponent()
    navWindow.Content = nextPage
End Sub
```

Each [NavigationWindow](#) object has a journal that records the user's navigation history in that window. One of

the purposes of the journal is to allow users to retrace their steps.

When you navigate using a uniform resource identifier (URI), the journal stores only the uniform resource identifier (URI) reference. This means that each time you revisit the page, it is dynamically reconstructed, which may be time consuming depending on the complexity of the page. In this case, the journal storage cost is low, but the time to reconstitute the page is potentially high.

When you navigate using an object, the journal stores the entire visual tree of the object. This means that each time you revisit the page, it renders immediately without having to be reconstructed. In this case, the journal storage cost is high, but the time to reconstitute the page is low.

When you use the [NavigationWindow](#) object, you will need to keep in mind how the journaling support impacts your application's performance. For more information, see [Navigation Overview](#).

## Hit Testing on Large 3D Surfaces

Hit testing on large 3D surfaces is a very performance intensive operation in terms of CPU consumption. This is especially true when the 3D surface is animating. If you do not require hit testing on these surfaces, then disable hit testing. Objects that are derived from [UIElement](#) can disable hit testing by setting the [IsHitTestVisible](#) property to `false`.

## CompositionTarget.Rendering Event

The [CompositionTarget.Rendering](#) event causes WPF to continuously animate. If you use this event, detach it at every opportunity.

## Avoid Using ScrollBarVisibility=Auto

Whenever possible, avoid using the [ScrollBarVisibility.Auto](#) value for the [HorizontalScrollBarVisibility](#) and [VerticalScrollBarVisibility](#) properties. These properties are defined for [RichTextBox](#), [ScrollViewer](#), and [TextBox](#) objects, and as an attached property for the [ListBox](#) object. Instead, set [ScrollBarVisibility](#) to [Disabled](#), [Hidden](#), or [Visible](#).

The [Auto](#) value is intended for cases when space is limited and scrollbars should only be displayed when necessary. For example, it may be useful to use this [ScrollBarVisibility](#) value with a [ListBox](#) of 30 items as opposed to a [TextBox](#) with hundreds of lines of text.

## Configure Font Cache Service to Reduce Start-up Time

The WPF Font Cache service shares font data between WPF applications. The first WPF application you run starts this service if the service is not already running. If you are using Windows Vista, you can set the "Windows Presentation Foundation (WPF) Font Cache 3.0.0.0" service from "Manual" (the default) to "Automatic (Delayed Start)" to reduce the initial start-up time of WPF applications.

## See also

- [Planning for Application Performance](#)
- [Taking Advantage of Hardware](#)
- [Layout and Design](#)
- [2D Graphics and Imaging](#)
- [Object Behavior](#)
- [Application Resources](#)
- [Text](#)
- [Data Binding](#)

- Animation Tips and Tricks

# Application Startup Time

7/1/2019 • 7 minutes to read • [Edit Online](#)

The amount of time that is required for a WPF application to start can vary greatly. This topic describes various techniques for reducing the perceived and actual startup time for a Windows Presentation Foundation (WPF) application.

## Understanding Cold Startup and Warm Startup

Cold startup occurs when your application starts for the first time after a system reboot, or when you start your application, close it, and then start it again after a long period of time. When an application starts, if the required pages (code, static data, registry, etc) are not present in the Windows memory manager's standby list, page faults occur. Disk access is required to bring the pages into memory.

Warm startup occurs when most of the pages for the main common language runtime (CLR) components are already loaded in memory, which saves expensive disk access time. That is why a managed application starts faster when it runs a second time.

## Implement a Splash Screen

In cases where there is a significant, unavoidable delay between starting an application and displaying the first UI, optimize the perceived startup time by using a *splash screen*. This approach displays an image almost immediately after the user starts the application. When the application is ready to display its first UI, the splash screen fades. Starting in the .NET Framework 3.5 SP1, you can use the [SplashScreen](#) class to implement a splash screen. For more information, see [Add a Splash Screen to a WPF Application](#).

You can also implement your own splash screen by using native Win32 graphics. Display your implementation before the [Run](#) method is called.

## Analyze the Startup Code

Determine the reason for a slow cold startup. Disk I/O may be responsible, but this is not always the case. In general, you should minimize the use of external resources, such as network, Web services, or disk.

Before you test, verify that no other running applications or services use managed code or WPF code.

Start your WPF application immediately after a reboot, and determine how long it takes to display. If all subsequent launches of your application (warm startup) are much faster, your cold startup issue is most likely caused by I/O.

If your application's cold startup issue is not related to I/O, it is likely that your application performs some lengthy initialization or computation, waits for some event to complete, or requires a lot of JIT compilation at startup. The following sections describe some of these situations in more detail.

## Optimize Module Loading

Use tools such as Process Explorer (Processexp.exe) and Tlist.exe to determine which modules your application loads. The command `Tlist <pid>` shows all the modules that are loaded by a process.

For example, if you are not connecting to the Web and you see that System.Web.dll is loaded, then there is a module in your application that references this assembly. Check to make sure that the reference is necessary.

If your application has multiple modules, merge them into a single module. This approach requires less CLR assembly-loading overhead. Fewer assemblies also mean that the CLR maintains less state.

## Defer Initialization Operations

Consider postponing initialization code until after the main application window is rendered.

Be aware that initialization may be performed inside a class constructor, and if the initialization code references other classes, it can cause a cascading effect in which many class constructors are executed.

## Avoid Application Configuration

Consider avoiding application configuration. For example, if an application has simple configuration requirements and has strict startup time goals, registry entries or a simple INI file may be a faster startup alternative.

## Utilize the GAC

If an assembly is not installed in the Global Assembly Cache (GAC), there are delays caused by hash verification of strong-named assemblies and by Ngen image validation if a native image for that assembly is available on the computer. Strong-name verification is skipped for all assemblies installed in the GAC. For more information, see [Gacutil.exe \(Global Assembly Cache Tool\)](#).

## Use Ngen.exe

Consider using the Native Image Generator (Ngen.exe) on your application. Using Ngen.exe means trading CPU consumption for more disk access because the native image generated by Ngen.exe is likely to be larger than the MSIL image.

To improve the warm startup time, you should always use Ngen.exe on your application, because this avoids the CPU cost of JIT compilation of the application code.

In some cold startup scenarios, using Ngen.exe can also be helpful. This is because the JIT compiler (mscorjit.dll) does not have to be loaded.

Having both Ngen and JIT modules can have the worst effect. This is because mscojit.dll must be loaded, and when the JIT compiler works on your code, many pages in the Ngen images must be accessed when the JIT compiler reads the assemblies' metadata.

### Ngen and ClickOnce

The way you plan to deploy your application can also make a difference in load time. ClickOnce application deployment does not support Ngen. If you decide to use Ngen.exe for your application, you will have to use another deployment mechanism, such as Windows Installer.

For more information, see [Ngen.exe \(Native Image Generator\)](#).

### Rebasing and DLL Address Collisions

If you use Ngen.exe, be aware that rebasing can occur when the native images are loaded in memory. If a DLL is not loaded at its preferred base address because that address range is already allocated, the Windows loader will load it at another address, which can be a time-consuming operation.

You can use the Virtual Address Dump (Vadump.exe) tool to check if there are modules in which all the pages are private. If this is the case, the module may have been rebased to a different address. Therefore, its pages cannot be shared.

For more information about how to set the base address, see [Ngen.exe \(Native Image Generator\)](#).

## Optimize Authenticode

Authenticode verification adds to the startup time. Authenticode-signed assemblies have to be verified with the certification authority (CA). This verification can be time consuming, because it can require connecting to the network several times to download current certificate revocation lists. It also makes sure that there is a full chain of valid certificates on the path to a trusted root. This can translate to several seconds of delay while the assembly is being loaded.

Consider installing the CA certificate on the client computer, or avoid using Authenticode when it is possible. If you know that your application does not need the publisher evidence, you do not have to pay the cost of signature verification.

Starting in .NET Framework 3.5, there is a configuration option that allows the Authenticode verification to be bypassed. To do this, add the following setting to the app.exe.config file:

```
<configuration>
  <runtime>
    <generatePublisherEvidence enabled="false"/>
  </runtime>
</configuration>
```

For more information, see [<generatePublisherEvidence> Element](#).

## Compare Performance on Windows Vista

The memory manager in Windows Vista has a technology called SuperFetch. SuperFetch analyzes memory usage patterns over time to determine the optimal memory content for a specific user. It works continuously to maintain that content at all times.

This approach differs from the pre-fetch technique used in Windows XP, which preloads data into memory without analyzing usage patterns. Over time, if the user uses your WPF application frequently on Windows Vista, the cold startup time of your application may improve.

## Use AppDomains Efficiently

If possible, load assemblies into a domain-neutral code area to make sure that the native image, if one exists, is used in all AppDomains created in the application.

For the best performance, enforce efficient cross-domain communication by reducing cross-domain calls. When possible, use calls without arguments or with primitive type arguments.

## Use the NeutralResourcesLanguage Attribute

Use the [NeutralResourcesLanguageAttribute](#) to specify the neutral culture for the [ResourceManager](#). This approach avoids unsuccessful assembly lookups.

## Use the BinaryFormatter Class for Serialization

If you must use serialization, use the [BinaryFormatter](#) class instead of the [XmlSerializer](#) class. The [BinaryFormatter](#) class is implemented in the Base Class Library (BCL) in the mscorlib.dll assembly. The [XmlSerializer](#) is implemented in the System.Xml.dll assembly, which might be an additional DLL to load.

If you must use the [XmlSerializer](#) class, you can achieve better performance if you pre-generate the serialization assembly.

## Configure ClickOnce to Check for Updates After Startup

If your application uses ClickOnce, avoid network access on startup by configuring ClickOnce to check the deployment site for updates after the application starts.

If you use the XAML browser application (XBAP) model, keep in mind that ClickOnce checks the deployment site for updates even if the XBAP is already in the ClickOnce cache. For more information, see [ClickOnce Security and Deployment](#).

## Configure the PresentationFontCache Service to Start Automatically

The first WPF application to run after a reboot is the PresentationFontCache service. The service caches the system fonts, improves font access, and improves overall performance. There is an overhead in starting the service, and in some controlled environments, consider configuring the service to start automatically when the system reboots.

## Set Data Binding Programmatically

Instead of using XAML to set the [DataContext](#) declaratively for the main window, consider setting it programmatically in the [OnActivated](#) method.

## See also

- [SplashScreen](#)
- [AppDomain](#)
- [NeutralResourcesLanguageAttribute](#)
- [ResourceManager](#)
- [Add a Splash Screen to a WPF Application](#)
- [Ngen.exe \(Native Image Generator\)](#)
- [<generatePublisherEvidence> Element](#)

# Walkthrough: Caching Application Data in a WPF Application

12/3/2019 • 9 minutes to read • [Edit Online](#)

Caching enables you to store data in memory for rapid access. When the data is accessed again, applications can get the data from the cache instead of retrieving it from the original source. This can improve performance and scalability. In addition, caching makes data available when the data source is temporarily unavailable.

The .NET Framework provides classes that enable you to use caching in .NET Framework applications. These classes are located in the [System.Runtime.Caching](#) namespace.

## NOTE

The [System.Runtime.Caching](#) namespace is new in the .NET Framework 4. This namespace makes caching is available to all .NET Framework applications. In previous versions of the .NET Framework, caching was available only in the [System.Web](#) namespace and therefore required a dependency on ASP.NET classes.

This walkthrough shows you how to use the caching functionality that is available in the .NET Framework as part of a Windows Presentation Foundation (WPF) application. In the walkthrough, you cache the contents of a text file.

Tasks illustrated in this walkthrough include the following:

- Creating a WPF application project.
- Adding a reference to the .NET Framework 4.
- Initializing a cache.
- Adding a cache entry that contains the contents of a text file.
- Providing an eviction policy for the cache entry.
- Monitoring the path of the cached file and notifying the cache instance about changes to the monitored item.

## Prerequisites

In order to complete this walkthrough, you will need:

- Visual Studio 2010.
- A text file that contains a small amount of text. (You will display the contents of the text file in a message box.) The code illustrated in the walkthrough assumes that you are working with the following file:

c:\cache\cacheText.txt

However, you can use any text file and make small changes to the code in this walkthrough.

## Creating a WPF Application Project

You will start by creating a WPF application project.

**To create a WPF application**

1. Start Visual Studio.
  2. In the **File** menu, click **New**, and then click **New Project**.
- The **New Project** dialog box is displayed.
3. Under **Installed Templates**, select the programming language you want to use (**Visual Basic** or **Visual C#**).
  4. In the **New Project** dialog box, select **WPF Application**.

**NOTE**

If you do not see the **WPF Application** template, make sure that you are targeting a version of the .NET Framework that supports WPF. In the **New Project** dialog box, select .NET Framework 4 from the list.

5. In the **Name** text box, enter a name for your project. For example, you can enter **WPF Caching**.
6. Select the **Create directory for solution** check box.
7. Click **OK**.

The WPF Designer opens in **Design** view and displays the MainWindow.xaml file. Visual Studio creates the **My Project** folder, the Application.xaml file, and the MainWindow.xaml file.

## Targeting the .NET Framework and Adding a Reference to the Caching Assemblies

By default, WPF applications target the .NET Framework 4 Client Profile. To use the [System.Runtime.Caching](#) namespace in a WPF application, the application must target the .NET Framework 4 (not the .NET Framework 4 Client Profile) and must include a reference to the namespace.

Therefore, the next step is to change the .NET Framework target and add a reference to the [System.Runtime.Caching](#) namespace.

**NOTE**

The procedure for changing the .NET Framework target is different in a Visual Basic project and in a Visual C# project.

### To change the target .NET Framework in Visual Basic

1. In **Solutions Explorer**, right-click the project name, and then click **Properties**.

The properties window for the application is displayed.

2. Click the **Compile** tab.
3. At the bottom of the window, click **Advanced Compile Options....**

The **Advanced Compiler Settings** dialog box is displayed.

4. In the **Target framework (all configurations)** list, select .NET Framework 4. (Do not select .NET Framework 4 Client Profile.)
  5. Click **OK**.
- The **Target Framework Change** dialog box is displayed.
6. In the **Target Framework Change** dialog box, click **Yes**.

The project is closed and is then reopened.

7. Add a reference to the caching assembly by following these steps:

- a. In **Solution Explorer**, right-click the name of the project and then click **Add Reference**.
- b. Select the **.NET** tab, select `System.Runtime.Caching`, and then click **OK**.

**To change the target .NET Framework in a Visual C# project**

1. In **Solution Explorer**, right-click the project name and then click **Properties**.

The properties window for the application is displayed.

2. Click the **Application** tab.
3. In the **Target framework** list, select .NET Framework 4. (Do not select **.NET Framework 4 Client Profile**.)
4. Add a reference to the caching assembly by following these steps:
  - a. Right-click the **References** folder and then click **Add Reference**.
  - b. Select the **.NET** tab, select `System.Runtime.Caching`, and then click **OK**.

## Adding a Button to the WPF Window

Next, you will add a button control and create an event handler for the button's `Click` event. Later you will add code to so when you click the button, the contents of the text file are cached and displayed.

**To add a button control**

1. In **Solution Explorer**, double-click the `MainWindow.xaml` file to open it.
2. From the **Toolbox**, under **Common WPF Controls**, drag a `Button` control to the `MainWindow` window.
3. In the **Properties** window, set the `Content` property of the `Button` control to **Get Cache**.

## Initializing the Cache and Caching an Entry

Next, you will add the code to perform the following tasks:

- Create an instance of the cache class—that is, you will instantiate a new `MemoryCache` object.
- Specify that the cache uses a `HostFileChangeMonitor` object to monitor changes in the text file.
- Read the text file and cache its contents as a cache entry.
- Display the contents of the cached text file.

**To create the cache object**

1. Double-click the button you just added in order to create an event handler in the `MainWindow.xaml.cs` or `MainWindow.Xaml.vb` file.
2. At the top of the file (before the class declaration), add the following `Imports` (Visual Basic) or `using` (C#) statements:

```
using System.Runtime.Caching;
using System.IO;
```

```
Imports System.Runtime.Caching  
Imports System.IO
```

3. In the event handler, add the following code to instantiate the cache object:

```
ObjectCache cache = MemoryCache.Default;
```

```
Dim cache As ObjectCache = MemoryCache.Default
```

The [ObjectCache](#) class is a built-in class that provides an in-memory object cache.

4. Add the following code to read the contents of a cache entry named `filecontents`:

```
Dim fileContents As String = TryCast(cache("filecontents"), String)
```

```
string fileContents = cache["filecontents"] as string;
```

5. Add the following code to check whether the cache entry named `filecontents` exists:

```
If fileContents Is Nothing Then  
End If
```

```
if (fileContents == null)  
{  
}
```

If the specified cache entry does not exist, you must read the text file and add it as a cache entry to the cache.

6. In the `if/then` block, add the following code to create a new [CacheItemPolicy](#) object that specifies that the cache entry expires after 10 seconds.

```
Dim policy As New CacheItemPolicy()  
policy.AbsoluteExpiration = DateTimeOffset.Now.AddSeconds(10.0)
```

```
CacheItemPolicy policy = new CacheItemPolicy();  
policy.AbsoluteExpiration = DateTimeOffset.Now.AddSeconds(10.0);
```

If no eviction or expiration information is provided, the default is [InfiniteAbsoluteExpiration](#), which means the cache entries never expire based only on an absolute time. Instead, cache entries expire only when there is memory pressure. As a best practice, you should always explicitly provide either an absolute or a sliding expiration.

7. Inside the `if/then` block and following the code you added in the previous step, add the following code to create a collection for the file paths that you want to monitor, and to add the path of the text file to the collection:

```
Dim filePaths As New List(Of String)()
filePaths.Add("c:\cache\cacheText.txt")
```

```
List<string> filePaths = new List<string>();
filePaths.Add("c:\\cache\\cacheText.txt");
```

**NOTE**

If the text file you want to use is not `c:\cache\cacheText.txt`, specify the path where the text file is that you want to use.

- Following the code that you added in the previous step, add the following code to add a new [HostFileChangeMonitor](#) object to the collection of change monitors for the cache entry:

```
policy.ChangeMonitors.Add(New HostFileChangeMonitor(filePaths))
```

```
policy.ChangeMonitors.Add(new HostFileChangeMonitor(filePaths));
```

The [HostFileChangeMonitor](#) object monitors the text file's path and notifies the cache if changes occur. In this example, the cache entry will expire if the content of the file changes.

- Following the code that you added in the previous step, add the following code to read the contents of the text file:

```
fileContents = File.ReadAllText("c:\cache\cacheText.txt") & vbCrLf & DateTime.Now.ToString()
```

```
fileContents = File.ReadAllText("c:\\cache\\cacheText.txt") + "\n" + DateTime.Now;
```

The date and time timestamp is added so that you will be able to see when the cache entry expires.

- Following the code that you added in the previous step, add the following code to insert the contents of the file into the cache object as a [CacheItem](#) instance:

```
cache.Set("filecontents", fileContents, policy)
```

```
cache.Set("filecontents", fileContents, policy);
```

You specify information about how the cache entry should be evicted by passing the [CacheItemPolicy](#) object that you created earlier as a parameter.

- After the `if/then` block, add the following code to display the cached file content in a message box:

```
MessageBox.Show(fileContents)
```

```
MessageBox.Show(fileContents);
```

12. In the **Build** menu, click **Build WPF Caching** to build your project.

## Testing Caching in the WPF Application

You can now test the application.

### To test caching in the WPF application

1. Press CTRL+F5 to run the application.

The `MainWindow` window is displayed.

2. Click **Get Cache**.

The cached content from the text file is displayed in a message box. Notice the timestamp on the file.

3. Close the message box and then click **Get Cache** again.

The timestamp is unchanged. This indicates the cached content is displayed.

4. Wait 10 seconds or more and then click **Get Cache** again.

This time a new timestamp is displayed. This indicates that the policy let the cache entry expire and that new cached content is displayed.

5. In a text editor, open the text file that you created. Do not make any changes yet.

6. Close the message box and then click **Get Cache** again.

Notice the timestamp again.

7. Make a change to the text file and then save the file.

8. Close the message box and then click **Get Cache** again.

This message box contains the updated content from the text file and a new timestamp. This indicates that the host-file change monitor evicted the cache entry immediately when you changed the file, even though the absolute timeout period had not expired.

#### NOTE

You can increase the eviction time to 20 seconds or more to allow more time for you to make a change in the file.

## Code Example

After you have completed this walkthrough, the code for the project you created will resemble the following example.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Runtime.Caching;
using System.IO;

namespace WPFCaching
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, RoutedEventArgs e)
        {

            ObjectCache cache = MemoryCache.Default;
            string fileContents = cache["filecontents"] as string;

            if (fileContents == null)
            {
                CacheItemPolicy policy = new CacheItemPolicy();
                policy.AbsoluteExpiration =
                    DateTimeOffset.Now.AddSeconds(10.0);

                List<string> filePaths = new List<string>();
                filePaths.Add("c:\\cache\\cacheText.txt");

                policy.ChangeMonitors.Add(new
                    HostFileChangeMonitor(filePaths));

                // Fetch the file contents.
                fileContents = File.ReadAllText("c:\\cache\\cacheText.txt") + "\n" + DateTime.Now.ToString();

                cache.Set("filecontents", fileContents, policy);
            }
            MessageBox.Show(fileContents);
        }
    }
}

```

```
Imports System.Runtime.Caching
Imports System.IO

Class MainWindow

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.Windows.RoutedEventArgs)
Handles Button1.Click
        Dim cache As ObjectCache = MemoryCache.Default
        Dim fileContents As String = TryCast(cache("filecontents"), _
            String)

        If fileContents Is Nothing Then
            Dim policy As New CacheItemPolicy()
            policy.AbsoluteExpiration = _
                DateTimeOffset.Now.AddSeconds(10.0)
            Dim filePaths As New List(Of String)()
            filePaths.Add("c:\cache\cacheText.txt")
            policy.ChangeMonitors.Add(New _
                HostFileChangeMonitor(filePaths))

            ' Fetch the file contents.
            fileContents = File.ReadAllText("c:\cache\cacheText.txt") & vbCrLf & DateTime.Now.ToString()
            cache.Set("filecontents", fileContents, policy)
        End If
        MessageBox.Show(fileContents)
    End Sub
End Class
```

## See also

- [MemoryCache](#)
- [ObjectCache](#)
- [System.Runtime.Caching](#)
- [Caching in .NET Framework Applications](#)

# Threading Model

11/12/2019 • 24 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) is designed to save developers from the difficulties of threading. As a result, the majority of WPF developers won't have to write an interface that uses more than one thread. Because multithreaded programs are complex and difficult to debug, they should be avoided when single-threaded solutions exist.

No matter how well architected, however, no UI framework will ever be able to provide a single-threaded solution for every sort of problem. WPF comes close, but there are still situations where multiple threads improve user interface (UI) responsiveness or application performance. After discussing some background material, this paper explores some of these situations and then concludes with a discussion of some lower-level details.

## NOTE

This topic discusses threading by using the `BeginInvoke` method for asynchronous calls. You can also make asynchronous calls by calling the `InvokeAsync` method, which take an `Action` or `Func<TResult>` as a parameter. The `InvokeAsync` method returns a `DispatcherOperation` or `DispatcherOperation<TResult>`, which has a `Task` property. You can use the `await` keyword with either the `DispatcherOperation` or the associated `Task`. If you need to wait synchronously for the `Task` that is returned by a `DispatcherOperation` or `DispatcherOperation<TResult>`, call the `DispatcherOperationWait` extension method. Calling `Task.Wait` will result in a deadlock. For more information about using a `Task` to perform asynchronous operations, see Task Parallelism. The `Invoke` method also has overloads that take an `Action` or `Func<TResult>` as a parameter. You can use the `Invoke` method to make synchronous calls by passing in a delegate, `Action` or `Func<TResult>`.

## Overview and the Dispatcher

Typically, WPF applications start with two threads: one for handling rendering and another for managing the UI. The rendering thread effectively runs hidden in the background while the UI thread receives input, handles events, paints the screen, and runs application code. Most applications use a single UI thread, although in some situations it is best to use several. We'll discuss this with an example later.

The UI thread queues work items inside an object called a `Dispatcher`. The `Dispatcher` selects work items on a priority basis and runs each one to completion. Every UI thread must have at least one `Dispatcher`, and each `Dispatcher` can execute work items in exactly one thread.

The trick to building responsive, user-friendly applications is to maximize the `Dispatcher` throughput by keeping the work items small. This way items never get stale sitting in the `Dispatcher` queue waiting for processing. Any perceivable delay between input and response can frustrate a user.

How then are WPF applications supposed to handle big operations? What if your code involves a large calculation or needs to query a database on some remote server? Usually, the answer is to handle the big operation in a separate thread, leaving the UI thread free to tend to items in the `Dispatcher` queue. When the big operation is complete, it can report its result back to the UI thread for display.

Historically, Windows allows UI elements to be accessed only by the thread that created them. This means that a background thread in charge of some long-running task cannot update a text box when it is finished. Windows does this to ensure the integrity of UI components. A list box could look strange if its contents were updated by a background thread during painting.

WPF has a built-in mutual exclusion mechanism that enforces this coordination. Most classes in WPF derive

from [DispatcherObject](#). At construction, a [DispatcherObject](#) stores a reference to the [Dispatcher](#) linked to the currently running thread. In effect, the [DispatcherObject](#) associates with the thread that creates it. During program execution, a [DispatcherObject](#) can call its public [VerifyAccess](#) method. [VerifyAccess](#) examines the [Dispatcher](#) associated with the current thread and compares it to the [Dispatcher](#) reference stored during construction. If they don't match, [VerifyAccess](#) throws an exception. [VerifyAccess](#) is intended to be called at the beginning of every method belonging to a [DispatcherObject](#).

If only one thread can modify the UI, how do background threads interact with the user? A background thread can ask the UI thread to perform an operation on its behalf. It does this by registering a work item with the [Dispatcher](#) of the UI thread. The [Dispatcher](#) class provides two methods for registering work items: [Invoke](#) and [BeginInvoke](#). Both methods schedule a delegate for execution. [Invoke](#) is a synchronous call – that is, it doesn't return until the UI thread actually finishes executing the delegate. [BeginInvoke](#) is asynchronous and returns immediately.

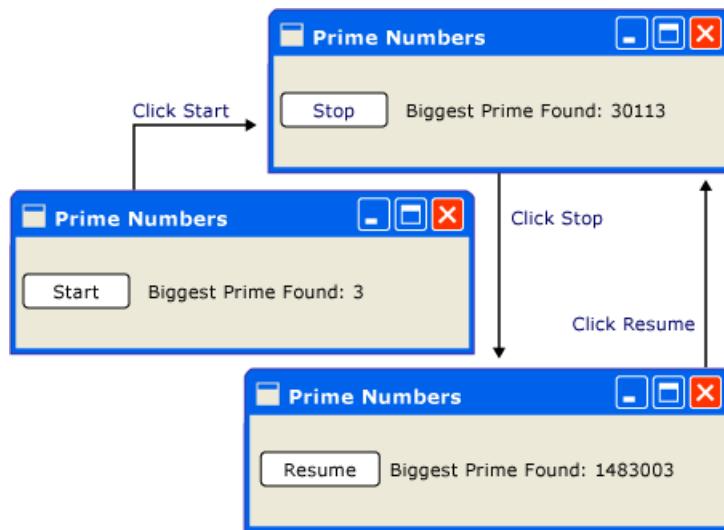
The [Dispatcher](#) orders the elements in its queue by priority. There are ten levels that may be specified when adding an element to the [Dispatcher](#) queue. These priorities are maintained in the [DispatcherPriority](#) enumeration. Detailed information about [DispatcherPriority](#) levels can be found in the Windows SDK documentation.

## Threads in Action: The Samples

### A Single-Threaded Application with a Long-Running Calculation

Most graphical user interfaces (GUIs) spend a large portion of their time idle while waiting for events that are generated in response to user interactions. With careful programming this idle time can be used constructively, without affecting the responsiveness of the UI. The WPF threading model doesn't allow input to interrupt an operation happening in the UI thread. This means you must be sure to return to the [Dispatcher](#) periodically to process pending input events before they get stale.

Consider the following example:



This simple application counts upwards from three, searching for prime numbers. When the user clicks the **Start** button, the search begins. When the program finds a prime, it updates the user interface with its discovery. At any point, the user can stop the search.

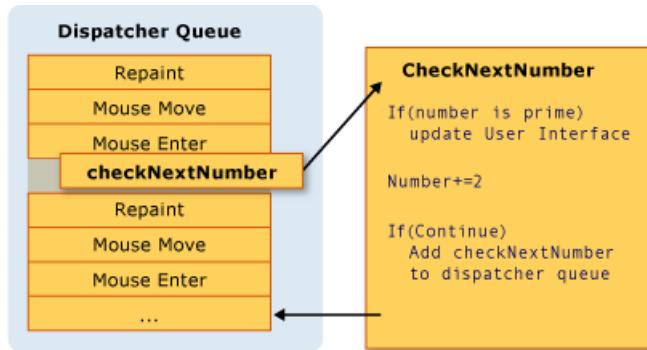
Although simple enough, the prime number search could go on forever, which presents some difficulties. If we handled the entire search inside of the click event handler of the button, we would never give the UI thread a chance to handle other events. The UI would be unable to respond to input or process messages. It would never repaint and never respond to button clicks.

We could conduct the prime number search in a separate thread, but then we would need to deal with synchronization issues. With a single-threaded approach, we can directly update the label that lists the largest

prime found.

If we break up the task of calculation into manageable chunks, we can periodically return to the **Dispatcher** and process events. We can give WPF an opportunity to repaint and process input.

The best way to split processing time between calculation and event handling is to manage calculation from the **Dispatcher**. By using the [BeginInvoke](#) method, we can schedule prime number checks in the same queue that UI events are drawn from. In our example, we schedule only a single prime number check at a time. After the prime number check is complete, we schedule the next check immediately. This check proceeds only after pending UI events have been handled.



Microsoft Word accomplishes spell checking using this mechanism. Spell checking is done in the background using the idle time of the UI thread. Let's take a look at the code.

The following example shows the XAML that creates the user interface.

```
<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Prime Numbers" Width="260" Height="75"
    >
<StackPanel Orientation="Horizontal" VerticalAlignment="Center" >
    <Button Content="Start"
        Click="StartOrStop"
        Name="startStopButton"
        Margin="5,0,5,0"
        />
    <TextBlock Margin="10,5,0,0">Biggest Prime Found:</TextBlock>
    <TextBlock Name="bigPrime" Margin="4,5,0,0">3</TextBlock>
</StackPanel>
</Window>
```

The following example shows the code-behind.

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

namespace SDKSamples
{
    public partial class Window1 : Window
    {
        public delegate void NextPrimeDelegate();

        //Current number to check
        private long num = 3;

        private bool continueCalculating = false;
```

```

public Window1() : base()
{
    InitializeComponent();
}

private void StartOrStop(object sender, EventArgs e)
{
    if (continueCalculating)
    {
        continueCalculating = false;
        startStopButton.Content = "Resume";
    }
    else
    {
        continueCalculating = true;
        startStopButton.Content = "Stop";
        startStopButton.Dispatcher.BeginInvoke(
            DispatcherPriority.Normal,
            new NextPrimeDelegate(CheckNextNumber));
    }
}

public void CheckNextNumber()
{
    // Reset flag.
    NotAPrime = false;

    for (long i = 3; i <= Math.Sqrt(num); i++)
    {
        if (num % i == 0)
        {
            // Set not a prime flag to true.
            NotAPrime = true;
            break;
        }
    }

    // If a prime number.
    if (!NotAPrime)
    {
        bigPrime.Text = num.ToString();
    }

    num += 2;
    if (continueCalculating)
    {
        startStopButton.Dispatcher.BeginInvoke(
            System.Windows.Threading.DispatcherPriority.SystemIdle,
            new NextPrimeDelegate(this.CheckNextNumber));
    }
}

private bool NotAPrime = false;
}
}

```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Threading
Imports System.Threading

Namespace SDKSamples
    Partial Public Class MainWindow
        Inherits Window
        Public Delegate Sub NextPrimeDelegate()

        'Current number to check
        Private num As Long = 3

        Private continueCalculating As Boolean = False

        Public Sub New()
            MyBase.New()
            InitializeComponent()
        End Sub

        Private Sub StartOrStop(ByVal sender As Object, ByVal e As EventArgs)
            If continueCalculating Then
                continueCalculating = False
                startStopButton.Content = "Resume"
            Else
                continueCalculating = True
                startStopButton.Content = "Stop"
                startStopButton.Dispatcher.BeginInvoke(DispatcherPriority.Normal, New
NextPrimeDelegate(AddressOf CheckNextNumber))
            End If
        End Sub

        Public Sub CheckNextNumber()
            ' Reset flag.
            NotAPrime = False

            For i As Long = 3 To Math.Sqrt(num)
                If num Mod i = 0 Then
                    ' Set not a prime flag to true.
                    NotAPrime = True
                    Exit For
                End If
            Next

            ' If a prime number.
            If Not NotAPrime Then
                bigPrime.Text = num.ToString()
            End If

            num += 2
            If continueCalculating Then

startStopButton.Dispatcher.BeginInvoke(System.Windows.Threading.DispatcherPriority.SystemIdle, New
NextPrimeDelegate(AddressOf Me.CheckNextNumber))
            End If
        End Sub

        Private NotAPrime As Boolean = False
    End Class
End Namespace

```

The following example shows the event handler for the [Button](#).

```

private void StartOrStop(object sender, EventArgs e)
{
    if (continueCalculating)
    {
        continueCalculating = false;
        startStopButton.Content = "Resume";
    }
    else
    {
        continueCalculating = true;
        startStopButton.Content = "Stop";
        startStopButton.Dispatcher.BeginInvoke(
            DispatcherPriority.Normal,
            new NextPrimeDelegate(CheckNextNumber));
    }
}

```

```

Private Sub StartOrStop(ByVal sender As Object, ByVal e As EventArgs)
    If continueCalculating Then
        continueCalculating = False
        startStopButton.Content = "Resume"
    Else
        continueCalculating = True
        startStopButton.Content = "Stop"
        startStopButton.Dispatcher.BeginInvoke(DispatcherPriority.Normal, New NextPrimeDelegate(AddressOf
CheckNextNumber))
    End If
End Sub

```

Besides updating the text on the [Button](#), this handler is responsible for scheduling the first prime number check by adding a delegate to the [Dispatcher](#) queue. Sometime after this event handler has completed its work, the [Dispatcher](#) will select this delegate for execution.

As we mentioned earlier, [BeginInvoke](#) is the [Dispatcher](#) member used to schedule a delegate for execution. In this case, we choose the [SystemIdle](#) priority. The [Dispatcher](#) will execute this delegate only when there are no important events to process. UI responsiveness is more important than number checking. We also pass a new delegate representing the number-checking routine.

```

public void CheckNextNumber()
{
    // Reset flag.
    NotAPrime = false;

    for (long i = 3; i <= Math.Sqrt(num); i++)
    {
        if (num % i == 0)
        {
            // Set not a prime flag to true.
            NotAPrime = true;
            break;
        }
    }

    // If a prime number.
    if (!NotAPrime)
    {
        bigPrime.Text = num.ToString();
    }

    num += 2;
    if (continueCalculating)
    {
        startStopButton.Dispatcher.BeginInvoke(
            System.Windows.Threading.DispatcherPriority.SystemIdle,
            new NextPrimeDelegate(this.CheckNextNumber));
    }
}

private bool NotAPrime = false;

```

```

Public Sub CheckNextNumber()
    ' Reset flag.
    NotAPrime = False

    For i As Long = 3 To Math.Sqrt(num)
        If num Mod i = 0 Then
            ' Set not a prime flag to true.
            NotAPrime = True
            Exit For
        End If
    Next

    ' If a prime number.
    If Not NotAPrime Then
        bigPrime.Text = num.ToString()
    End If

    num += 2
    If continueCalculating Then
        startStopButton.Dispatcher.BeginInvoke(System.Windows.Threading.DispatcherPriority.SystemIdle, New
NextPrimeDelegate(AddressOf Me.CheckNextNumber))
    End If
End Sub

Private NotAPrime As Boolean = False

```

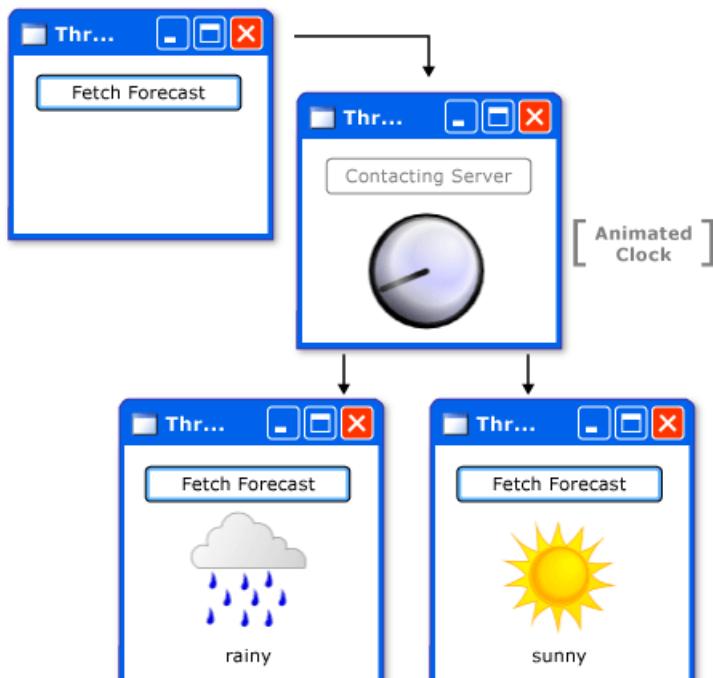
This method checks if the next odd number is prime. If it is prime, the method directly updates the `bigPrime` [TextBlock](#) to reflect its discovery. We can do this because the calculation is occurring in the same thread that was used to create the component. Had we chosen to use a separate thread for the calculation, we would have to use a more complicated synchronization mechanism and execute the update in the UI thread. We'll demonstrate this situation next.

For the complete source code for this sample, see the [Single-Threaded Application with Long-Running Calculation Sample](#)

### Handling a Blocking Operation with a Background Thread

Handling blocking operations in a graphical application can be difficult. We don't want to call blocking methods from event handlers because the application will appear to freeze up. We can use a separate thread to handle these operations, but when we're done, we have to synchronize with the UI thread because we can't directly modify the GUI from our worker thread. We can use [Invoke](#) or [BeginInvoke](#) to insert delegates into the [Dispatcher](#) of the UI thread. Eventually, these delegates will be executed with permission to modify UI elements.

In this example, we mimic a remote procedure call that retrieves a weather forecast. We use a separate worker thread to execute this call, and we schedule an update method in the [Dispatcher](#) of the UI thread when we're finished.



```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Threading;
using System.Threading;

namespace SDKSamples
{
    public partial class Window1 : Window
    {
        // Delegates to be used in placing jobs onto the Dispatcher.
        private delegate void NoArgDelegate();
        private delegate void OneArgDelegate(String arg);

        // Storyboards for the animations.
        private Storyboard showClockFaceStoryboard;
        private Storyboard hideClockFaceStoryboard;
        private Storyboard showWeatherImageStoryboard;
        private Storyboard hideWeatherImageStoryboard;

        public Window1(): base()
        {
            InitializeComponent();
        }
    }
}
```

```

}

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Load the storyboard resources.
    showClockFaceStoryboard =
        (Storyboard)this.Resources["ShowClockFaceStoryboard"];
    hideClockFaceStoryboard =
        (Storyboard)this.Resources["HideClockFaceStoryboard"];
    showWeatherImageStoryboard =
        (Storyboard)this.Resources["ShowWeatherImageStoryboard"];
    hideWeatherImageStoryboard =
        (Storyboard)this.Resources["HideWeatherImageStoryboard"];
}

private void ForecastButtonHandler(object sender, RoutedEventArgs e)
{
    // Change the status image and start the rotation animation.
    fetchButton.IsEnabled = false;
    fetchButton.Content = "Contacting Server";
    weatherText.Text = "";
    hideWeatherImageStoryboard.Begin(this);

    // Start fetching the weather forecast asynchronously.
    NoArgDelegate fetcher = new NoArgDelegate(
        this.FetchWeatherFromServer);

    fetcher.BeginInvoke(null, null);
}

private void FetchWeatherFromServer()
{
    // Simulate the delay from network access.
    Thread.Sleep(4000);

    // Tried and true method for weather forecasting - random numbers.
    Random rand = new Random();
    String weather;

    if (rand.Next(2) == 0)
    {
        weather = "rainy";
    }
    else
    {
        weather = "sunny";
    }

    // Schedule the update function in the UI thread.
    tomorrowsWeather.Dispatcher.BeginInvoke(
        System.Windows.Threading.DispatcherPriority.Normal,
        new OneArgDelegate(UpdateUserInterface),
        weather);
}

private void UpdateUserInterface(String weather)
{
    //Set the weather image
    if (weather == "sunny")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "SunnyImageSource"];
    }
    else if (weather == "rainy")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "RainingImageSource"];
    }
}

```

```

//Stop clock animation
showClockFaceStoryboard.Stop(this);
hideClockFaceStoryboard.Begin(this);

//Update UI text
fetchButton.IsEnabled = true;
fetchButton.Content = "Fetch Forecast";
weatherText.Text = weather;
}

private void HideClockFaceStoryboard_Completed(object sender,
EventArgs args)
{
    showWeatherImageStoryboard.Begin(this);
}

private void HideWeatherImageStoryboard_Completed(object sender,
EventArgs args)
{
    showClockFaceStoryboard.Begin(this, true);
}
}
}

```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Media
Imports System.Windows.Media.Animation
Imports System.Windows.Media.Imaging
Imports System.Windows.Shapes
Imports System.Windows.Threading
Imports System.Threading

Namespace SDKSamples
    Partial Public Class Window1
        Inherits Window
        ' Delegates to be used in placing jobs onto the Dispatcher.
        Private Delegate Sub NoArgDelegate()
        Private Delegate Sub OneArgDelegate(ByVal arg As String)

        ' Storyboards for the animations.
        Private showClockFaceStoryboard As Storyboard
        Private hideClockFaceStoryboard As Storyboard
        Private showWeatherImageStoryboard As Storyboard
        Private hideWeatherImageStoryboard As Storyboard

        Public Sub New()
            MyBase.New()
            InitializeComponent()
        End Sub

        Private Sub Window_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
            ' Load the storyboard resources.
            showClockFaceStoryboard = CType(Me.Resources("ShowClockFaceStoryboard"), Storyboard)
            hideClockFaceStoryboard = CType(Me.Resources("HideClockFaceStoryboard"), Storyboard)
            showWeatherImageStoryboard = CType(Me.Resources("ShowWeatherImageStoryboard"), Storyboard)
            hideWeatherImageStoryboard = CType(Me.Resources("HideWeatherImageStoryboard"), Storyboard)
        End Sub

        Private Sub ForecastButtonHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
            ' Change the status image and start the rotation animation.
            fetchButton.IsEnabled = False
            fetchButton.Content = "Contacting Server"
            weatherText.Text = ""
            hideWeatherImageStoryboard.Begin(Me)
        End Sub
    End Class

```

```

' Start fetching the weather forecast asynchronously.
Dim fetcher As New NoArgDelegate(AddressOf Me.FetchWeatherFromServer)

fetcher.BeginInvoke(Nothing, Nothing)
End Sub

Private Sub FetchWeatherFromServer()
    ' Simulate the delay from network access.
    Thread.Sleep(4000)

    ' Tried and true method for weather forecasting - random numbers.
    Dim rand As New Random()
    Dim weather As String

    If rand.Next(2) = 0 Then
        weather = "rainy"
    Else
        weather = "sunny"
    End If

    ' Schedule the update function in the UI thread.
    tomorrowWeather.Dispatcher.BeginInvoke(System.Windows.Threading.DispatcherPriority.Normal, New
OneArgDelegate(AddressOf UpdateUserInterface), weather)
End Sub

Private Sub UpdateUserInterface(ByVal weather As String)
    'Set the weather image
    If weather = "sunny" Then
        weatherIndicatorImage.Source = CType(Me.Resources("SunnyImageSource"), ImageSource)
    ElseIf weather = "rainy" Then
        weatherIndicatorImage.Source = CType(Me.Resources("RainingImageSource"), ImageSource)
    End If

    'Stop clock animation
    showClockFaceStoryboard.Stop(Me)
    hideClockFaceStoryboard.Begin(Me)

    'Update UI text
    fetchButton.IsEnabled = True
    fetchButton.Content = "Fetch Forecast"
    weatherText.Text = weather
End Sub

Private Sub HideClockFaceStoryboard_Completed(ByVal sender As Object, ByVal args As EventArgs)
    showWeatherImageStoryboard.Begin(Me)
End Sub

Private Sub HideWeatherImageStoryboard_Completed(ByVal sender As Object, ByVal args As EventArgs)
    showClockFaceStoryboard.Begin(Me, True)
End Sub
End Class
End Namespace

```

The following are some of the details to be noted.

- Creating the Button Handler

```

private void ForecastButtonHandler(object sender, RoutedEventArgs e)
{
    // Change the status image and start the rotation animation.
    fetchButton.IsEnabled = false;
    fetchButton.Content = "Contacting Server";
    weatherText.Text = "";
    hideWeatherImageStoryboard.Begin(this);

    // Start fetching the weather forecast asynchronously.
    NoArgDelegate fetcher = new NoArgDelegate(
        this.FetchWeatherFromServer);

    fetcher.BeginInvoke(null, null);
}

```

```

Private Sub ForecastButtonHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Change the status image and start the rotation animation.
    fetchButton.IsEnabled = False
    fetchButton.Content = "Contacting Server"
    weatherText.Text = ""
    hideWeatherImageStoryboard.Begin(Me)

    ' Start fetching the weather forecast asynchronously.
    Dim fetcher As New NoArgDelegate(AddressOf Me.FetchWeatherFromServer)

    fetcher.BeginInvoke(Nothing, Nothing)
End Sub

```

When the button is clicked, we display the clock drawing and start animating it. We disable the button. We invoke the `FetchWeatherFromServer` method in a new thread, and then we return, allowing the `Dispatcher` to process events while we wait to collect the weather forecast.

- Fetching the Weather

```

private void FetchWeatherFromServer()
{
    // Simulate the delay from network access.
    Thread.Sleep(4000);

    // Tried and true method for weather forecasting - random numbers.
    Random rand = new Random();
    String weather;

    if (rand.Next(2) == 0)
    {
        weather = "rainy";
    }
    else
    {
        weather = "sunny";
    }

    // Schedule the update function in the UI thread.
    tomorrowsWeather.Dispatcher.BeginInvoke(
        System.Windows.Threading.DispatcherPriority.Normal,
        new OneArgDelegate(UpdateUserInterface),
        weather);
}

```

```

Private Sub FetchWeatherFromServer()
    ' Simulate the delay from network access.
    Thread.Sleep(4000)

    ' Tried and true method for weather forecasting - random numbers.
    Dim rand As New Random()
    Dim weather As String

    If rand.Next(2) = 0 Then
        weather = "rainy"
    Else
        weather = "sunny"
    End If

    ' Schedule the update function in the UI thread.
    tomorrowWeather.Dispatcher.BeginInvoke(System.Windows.Threading.DispatcherPriority.Normal, New
    OneArgDelegate(AddressOf UpdateUserInterface), weather)
End Sub

```

To keep things simple, we don't actually have any networking code in this example. Instead, we simulate the delay of network access by putting our new thread to sleep for four seconds. In this time, the original UI thread is still running and responding to events. To show this, we've left an animation running, and the minimize and maximize buttons also continue to work.

When the delay is finished, and we've randomly selected our weather forecast, it's time to report back to the UI thread. We do this by scheduling a call to `UpdateUserInterface` in the UI thread using that thread's `Dispatcher`. We pass a string describing the weather to this scheduled method call.

- Updating the UI

```

private void UpdateUserInterface(String weather)
{
    //Set the weather image
    if (weather == "sunny")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "SunnyImageSource"];
    }
    else if (weather == "rainy")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "RainingImageSource"];
    }

    //Stop clock animation
    showClockFaceStoryboard.Stop(this);
    hideClockFaceStoryboard.Begin(this);

    //Update UI text
    fetchButton.IsEnabled = true;
    fetchButton.Content = "Fetch Forecast";
    weatherText.Text = weather;
}

```

```

Private Sub UpdateUserInterface(ByVal weather As String)
    'Set the weather image
    If weather = "sunny" Then
        weatherIndicatorImage.Source = CType(Me.Resources("SunnyImageSource"), ImageSource)
    ElseIf weather = "rainy" Then
        weatherIndicatorImage.Source = CType(Me.Resources("RainingImageSource"), ImageSource)
    End If

    'Stop clock animation
    showClockFaceStoryboard.Stop(Me)
    hideClockFaceStoryboard.Begin(Me)

    'Update UI text
    fetchButton.IsEnabled = True
    fetchButton.Content = "Fetch Forecast"
    weatherText.Text = weather
End Sub

```

When the [Dispatcher](#) in the UI thread has time, it executes the scheduled call to `UpdateUserInterface`. This method stops the clock animation and chooses an image to describe the weather. It displays this image and restores the "fetch forecast" button.

### Multiple Windows, Multiple Threads

Some WPF applications require multiple top-level windows. It is perfectly acceptable for one Thread/[Dispatcher](#) combination to manage multiple windows, but sometimes several threads do a better job. This is especially true if there is any chance that one of the windows will monopolize the thread.

Windows Explorer works in this fashion. Each new Explorer window belongs to the original process, but it is created under the control of an independent thread.

By using a [WPFFrame](#) control, we can display Web pages. We can easily create a simple Internet Explorer substitute. We start with an important feature: the ability to open a new explorer window. When the user clicks the "new window" button, we launch a copy of our window in a separate thread. This way, long-running or blocking operations in one of the windows won't lock all the other windows.

In reality, the Web browser model has its own complicated threading model. We've chosen it because it should be familiar to most readers.

The following example shows the code.

```
<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MultiBrowse"
    Height="600"
    Width="800"
    Loaded="OnLoaded"
    >
<StackPanel Name="Stack" Orientation="Vertical">
    <StackPanel Orientation="Horizontal">
        <Button Content="New Window"
            Click="NewWindowHandler" />
        <TextBox Name="newLocation"
            Width="500" />
        <Button Content="GO!" 
            Click="Browse" />
    </StackPanel>

    <Frame Name="placeHolder"
        Width="800"
        Height="550"></Frame>
</StackPanel>
</Window>
```

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Threading;
using System.Threading;

namespace SDKSamples
{
    public partial class Window1 : Window
    {

        public Window1() : base()
        {
            InitializeComponent();
        }

        private void OnLoaded(object sender, RoutedEventArgs e)
        {
            placeHolder.Source = new Uri("http://www.msn.com");
        }

        private void Browse(object sender, RoutedEventArgs e)
        {
            placeHolder.Source = new Uri(newLocation.Text);
        }

        private void NewWindowHandler(object sender, RoutedEventArgs e)
        {
            Thread newWindowThread = new Thread(new ThreadStart(ThreadStartingPoint));
            newWindowThread.SetApartmentState(ApartmentState.STA);
            newWindowThread.IsBackground = true;
            newWindowThread.Start();
        }

        private void ThreadStartingPoint()
        {
            Window1 tempWindow = new Window1();
            tempWindow.Show();
            System.Windows.Threading.Dispatcher.Run();
        }
    }
}
```

```

Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Data
Imports System.Windows.Threading
Imports System.Threading

Namespace SDKSamples
    Partial Public Class Window1
        Inherits Window

        Public Sub New()
            MyBase.New()
            InitializeComponent()
        End Sub

        Private Sub OnLoaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
            placeHolder.Source = New Uri("http://www.msn.com")
        End Sub

        Private Sub Browse(ByVal sender As Object, ByVal e As RoutedEventArgs)
            placeHolder.Source = New Uri(newLocation.Text)
        End Sub

        Private Sub NewWindowHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
            Dim newWindowThread As New Thread(New ThreadStart(AddressOf ThreadStartingPoint))
            newWindowThread.SetApartmentState(ApartmentState.STA)
            newWindowThread.IsBackground = True
            newWindowThread.Start()
        End Sub

        Private Sub ThreadStartingPoint()
            Dim tempWindow As New Window1()
            tempWindow.Show()
            System.Windows.Threading.Dispatcher.Run()
        End Sub
    End Class
End Namespace

```

The following threading segments of this code are the most interesting to us in this context:

```

private void NewWindowHandler(object sender, RoutedEventArgs e)
{
    Thread newWindowThread = new Thread(new ThreadStart(ThreadStartingPoint));
    newWindowThread.SetApartmentState(ApartmentState.STA);
    newWindowThread.IsBackground = true;
    newWindowThread.Start();
}

```

```

Private Sub NewWindowHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim newWindowThread As New Thread(New ThreadStart(AddressOf ThreadStartingPoint))
    newWindowThread.SetApartmentState(ApartmentState.STA)
    newWindowThread.IsBackground = True
    newWindowThread.Start()
End Sub

```

This method is called when the "new window" button is clicked. It creates a new thread and starts it asynchronously.

```
private void ThreadStartingPoint()
{
    Window1 tempWindow = new Window1();
    tempWindow.Show();
    System.Windows.Threading.Dispatcher.Run();
}
```

```
Private Sub ThreadStartingPoint()
    Dim tempWindow As New Window1()
    tempWindow.Show()
    System.Windows.Threading.Dispatcher.Run()
End Sub
```

This method is the starting point for the new thread. We create a new window under the control of this thread. WPF automatically creates a new [Dispatcher](#) to manage the new thread. All we have to do to make the window functional is to start the [Dispatcher](#).

## Technical Details and Stumbling Points

### Writing Components Using Threading

The Microsoft .NET Framework Developer's Guide describes a pattern for how a component can expose asynchronous behavior to its clients (see [Event-based Asynchronous Pattern Overview](#)). For instance, suppose we wanted to package the `FetchWeatherFromServer` method into a reusable, nongraphical component. Following the standard Microsoft .NET Framework pattern, this would look something like the following.

```
public class WeatherComponent : Component
{
    //gets weather: Synchronous
    public string GetWeather()
    {
        string weather = "";

        //predict the weather

        return weather;
    }

    //get weather: Asynchronous
    public void GetWeatherAsync()
    {
        //get the weather
    }

    public event GetWeatherCompletedEventHandler GetWeatherCompleted;
}

public class GetWeatherCompletedEventArgs : AsyncCompletedEventArgs
{
    public GetWeatherCompletedEventArgs(Exception error, bool canceled,
        object userState, string weather)
        :
        base(error, canceled, userState)
    {
        _weather = weather;
    }

    public string Weather
    {
        get { return _weather; }
    }
    private string _weather;
}

public delegate void GetWeatherCompletedEventHandler(object sender,
    GetWeatherCompletedEventArgs e);
```

```

Public Class WeatherComponent
    Inherits Component
    'gets weather: Synchronous
    Public Function GetWeather() As String
        Dim weather As String = ""

        'predict the weather

        Return weather
    End Function

    'get weather: Asynchronous
    Public Sub GetWeatherAsync()
        'get the weather
    End Sub

    Public Event GetWeatherCompleted As GetWeatherCompletedEventHandler
End Class

Public Class GetWeatherCompletedEventArgs
    Inherits AsyncCompletedEventArgs
    Public Sub New(ByVal [error] As Exception, ByVal canceled As Boolean, ByVal userState As Object, ByVal weather As String)
        MyBase.New([error], canceled, userState)
        _weather = weather
    End Sub

    Public ReadOnly Property Weather() As String
        Get
            Return _weather
        End Get
    End Property
    Private _weather As String
End Class

Public Delegate Sub GetWeatherCompletedEventHandler(ByVal sender As Object, ByVal e As GetWeatherCompletedEventArgs)

```

`GetWeatherAsync` would use one of the techniques described earlier, such as creating a background thread, to do the work asynchronously, not blocking the calling thread.

One of the most important parts of this pattern is calling the `MethodNameCompleted` method on the same thread that called the `MethodNameAsync` method to begin with. You could do this using WPF fairly easily, by storing `CurrentDispatcher`—but then the nongraphical component could only be used in WPF applications, not in Windows Forms or ASP.NET programs.

The `DispatcherSynchronizationContext` class addresses this need—think of it as a simplified version of `Dispatcher` that works with other UI frameworks as well.

```
public class WeatherComponent2 : Component
{
    public string GetWeather()
    {
        return fetchWeatherFromServer();
    }

    private DispatcherSynchronizationContext requestingContext = null;

    public void GetWeatherAsync()
    {
        if (requestingContext != null)
            throw new InvalidOperationException("This component can only handle 1 async request at a time");

        requestingContext = (DispatcherSynchronizationContext)DispatcherSynchronizationContext.Current;

        NoArgDelegate fetcher = new NoArgDelegate(this.fetchWeatherFromServer);

        // Launch thread
        fetcher.BeginInvoke(null, null);
    }

    private void RaiseEvent(GetWeatherCompletedEventArgs e)
    {
        if (GetWeatherCompleted != null)
            GetWeatherCompleted(this, e);
    }

    private string fetchWeatherFromServer()
    {
        // do stuff
        string weather = "";

        GetWeatherCompletedEventArgs e =
            new GetWeatherCompletedEventArgs(null, false, null, weather);

        SendOrPostCallback callback = new SendOrPostCallback(DoEvent);
        requestingContext.Post(callback, e);
        requestingContext = null;

        return e.Weather;
    }

    private void DoEvent(object e)
    {
        //do stuff
    }

    public event GetWeatherCompletedEventHandler GetWeatherCompleted;
    public delegate string NoArgDelegate();
}
```

```

Public Class WeatherComponent2
    Inherits Component
    Public Function GetWeather() As String
        Return fetchWeatherFromServer()
    End Function

    Private requestingContext As DispatcherSynchronizationContext = Nothing

    Public Sub GetWeatherAsync()
        If requestingContext IsNot Nothing Then
            Throw New InvalidOperationException("This component can only handle 1 async request at a time")
        End If

        requestingContext = CType(DispatcherSynchronizationContext.Current,
DispatcherSynchronizationContext)

        Dim fetcher As New NoArgDelegate(AddressOf Me.fetchWeatherFromServer)

        ' Launch thread
        fetcher.BeginInvoke(Nothing, Nothing)
    End Sub

    Private Sub [RaiseEvent](ByVal e As GetWeatherCompletedEventArgs)
        RaiseEvent GetWeatherCompleted(Me, e)
    End Sub

    Private Function fetchWeatherFromServer() As String
        ' do stuff
        Dim weather As String = ""

        Dim e As New GetWeatherCompletedEventArgs(Nothing, False, Nothing, weather)

        Dim callback As New SendOrPostCallback(AddressOf DoEvent)
        requestingContext.Post(callback, e)
        requestingContext = Nothing

        Return e.Weather
    End Function

    Private Sub DoEvent(ByVal e As Object)
        'do stuff
    End Sub

    Public Event GetWeatherCompleted As GetWeatherCompletedEventHandler
    Public Delegate Function NoArgDelegate() As String
End Class

```

## Nested Pumping

Sometimes it is not feasible to completely lock up the UI thread. Let's consider the [Show](#) method of the [MessageBox](#) class. [Show](#) doesn't return until the user clicks the OK button. It does, however, create a window that must have a message loop in order to be interactive. While we are waiting for the user to click OK, the original application window does not respond to user input. It does, however, continue to process paint messages. The original window redraws itself when covered and revealed.



Some thread must be in charge of the message box window. WPF could create a new thread just for the message box window, but this thread would be unable to paint the disabled elements in the original window (remember the earlier discussion of mutual exclusion). Instead, WPF uses a nested message processing system. The [Dispatcher](#) class includes a special method called [PushFrame](#), which stores an application's current execution point then begins a new message loop. When the nested message loop finishes, execution resumes after the original [PushFrame](#) call.

In this case, [PushFrame](#) maintains the program context at the call to [MessageBox.Show](#), and it starts a new message loop to repaint the background window and handle input to the message box window. When the user clicks OK and clears the pop-up window, the nested loop exits and control resumes after the call to [Show](#).

### Stale Routed Events

The routed event system in WPF notifies entire trees when events are raised.

```
<Canvas MouseLeftButtonDown="handler1"
    Width="100"
    Height="100"
    >
    <Ellipse Width="50"
        Height="50"
        Fill="Blue"
        Canvas.Left="30"
        Canvas.Top="50"
        MouseLeftButtonDown="handler2"
    />
</Canvas>
```

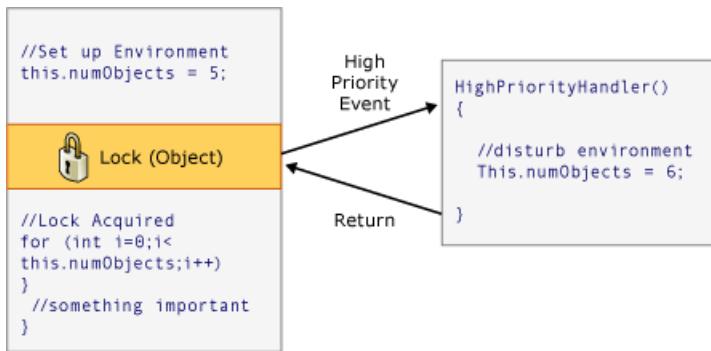
When the left mouse button is pressed over the ellipse, `handler2` is executed. After `handler2` finishes, the event is passed along to the [Canvas](#) object, which uses `handler1` to process it. This happens only if `handler2` does not explicitly mark the event object as handled.

It's possible that `handler2` will take a great deal of time processing this event. `handler2` might use [PushFrame](#) to begin a nested message loop that doesn't return for hours. If `handler2` does not mark the event as handled when this message loop is complete, the event is passed up the tree even though it is very old.

### Reentrancy and Locking

The locking mechanism of the common language runtime (CLR) doesn't behave exactly as one might imagine; one might expect a thread to cease operation completely when requesting a lock. In actuality, the thread continues to receive and process high-priority messages. This helps prevent deadlocks and make interfaces minimally responsive, but it introduces the possibility for subtle bugs. The vast majority of the time you don't need to know anything about this, but under rare circumstances (usually involving Win32 window messages or COM STA components) this can be worth knowing.

Most interfaces are not built with thread safety in mind because developers work under the assumption that a UI is never accessed by more than one thread. In this case, that single thread may make environmental changes at unexpected times, causing those ill effects that the [DispatcherObject](#) mutual exclusion mechanism is supposed to solve. Consider the following pseudocode:



Most of the time that's the right thing, but there are times in WPF where such unexpected reentrancy can really cause problems. So, at certain key times, WPF calls [DisableProcessing](#), which changes the lock instruction for that thread to use the WPF reentrancy-free lock, instead of the usual CLR lock.

So why did the CLR team choose this behavior? It had to do with COM STA objects and the finalization thread. When an object is garbage collected, its `Finalize` method is run on the dedicated finalizer thread, not the UI thread. And therein lies the problem, because a COM STA object that was created on the UI thread can only be disposed on the UI thread. The CLR does the equivalent of a `BeginInvoke` (in this case using Win32's `SendMessage`). But if the UI thread is busy, the finalizer thread is stalled and the COM STA object can't be disposed, which creates a serious memory leak. So the CLR team made the tough call to make locks work the way they do.

The task for WPF is to avoid unexpected reentrancy without reintroducing the memory leak, which is why we don't block reentrancy everywhere.

## See also

- [Single-Threaded Application with Long-Running Calculation Sample](#)

# WPF Unmanaged API Reference

4/8/2019 • 2 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) libraries expose a number of unmanaged functions that are intended for internal use only. They should not be called from user code.

## In This Section

[Activate Function](#)

[CreateDispatchSTAForwarder Function](#)

[Deactivate Function](#)

[ForwardTranslateAccelerator Function](#)

[LoadFromHistory Function](#)

[ProcessUnhandledException Function](#)

[SaveToHistory Function](#)

[SetFakeActiveWindow Function](#)

## See also

- [Advanced](#)

# Activate Function (WPF Unmanaged API Reference)

3/6/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for windows management.

## Syntax

```
void Activate(
    const ActivateParameters* pParameters,
    __deref_out_ecount(1) LPUNKNOWN* ppInner,
);
```

## Parameters

`pParameters`

A pointer to the window's activation parameters.

`ppInner`

A pointer to the address of a single-element buffer that contains a pointer to an [IOleDocument](#) object.

## Requirements

**Platforms:** See [.NET Framework System Requirements](#).

**DLL:**

In the .NET Framework 3.0 and 3.5: PresentationHostDLL.dll

In the .NET Framework 4 and later: PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 3.0

## See also

- [WPF Unmanaged API Reference](#)

# CreateIDispatchSTAForwarder Function (WPF Unmanaged API Reference)

4/8/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for thread and windows management.

## Syntax

```
HRESULT CreateIDispatchSTAForwarder(
    __in IDispatch *pDispatchDelegate,
    __deref_out IDispatch **ppForwarder
)
```

## Parameters

### Property Value/Return Value

#### pDispatchDelegate

A pointer to an [IDispatch](#) interface.

#### ppForwarder

A pointer to the address of an [IDispatch](#) interface.

## Requirements

**Platforms:** See [.NET Framework System Requirements](#).

#### DLL:

In the .NET Framework 3.0 and 3.5: PresentationHostDLL.dll

In the .NET Framework 4 and later: PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 3.0

## See also

- [WPF Unmanaged API Reference](#)

# Deactivate Function (WPF Unmanaged API Reference)

4/8/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for windows management.

## Syntax

```
void Deactivate()
```

## Requirements

**Platforms:** See [.NET Framework System Requirements](#).

**DLL:**

In the .NET Framework 3.0 and 3.5: PresentationHostDLL.dll

In the .NET Framework 4 and later: PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 3.0

## See also

- [WPF Unmanaged API Reference](#)

# ForwardTranslateAccelerator Function (WPF Unmanaged API Reference)

4/8/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for windows management.

## Syntax

```
HRESULT ForwardTranslateAccelerator(
    MSG* pMsg,
    VARIANT_BOOL appUnhandled
)
```

## Parameters

pMsg

A pointer to a message.

appUnhandled

`true` when the app has already been given a chance to handle the input message, but has not handled it; otherwise, `false`.

## Requirements

**Platforms:** See [.NET Framework System Requirements](#).

**DLL:**

In the .NET Framework 3.0 and 3.5: PresentationHostDLL.dll

In the .NET Framework 4 and later: PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 3.0

## See also

- [WPF Unmanaged API Reference](#)

# LoadFromHistory Function (WPF Unmanaged API Reference)

4/8/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for windows management.

## Syntax

```
HRESULT LoadFromHistory_export(
    IStream* pHistoryStream,
    IBindCtx* pBindCtx
)
```

## Parameters

### pHistoryStream

A pointer to a stream of history information.

### pBindCtx

A pointer to a bind context.

## Requirements

**Platforms:** See [.NET Framework System Requirements](#).

### DLL:

In the .NET Framework 3.0 and 3.5: PresentationHostDLL.dll

In the .NET Framework 4 and later: PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 3.0

## See also

- [WPF Unmanaged API Reference](#)

# ProcessUnhandledException Function (WPF Unmanaged API Reference)

4/8/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for exception handling.

## Syntax

```
void __stdcall ProcessUnhandledException(
    __in_ecount(1) BSTR errorMsg
)
```

## Parameters

### errorMsg

The error message.

## Requirements

**Platforms:** See [.NET Framework System Requirements](#).

### DLL:

In the .NET Framework 3.0 and 3.5: PresentationHostDLL.dll

In the .NET Framework 4 and later: PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 3.0

## See also

- [WPF Unmanaged API Reference](#)

# SaveToHistory Function (WPF Unmanaged API Reference)

4/8/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for windows management.

## Syntax

```
HRESULT SaveToHistory(  
    __in_ecount(1) IStream* pHistoryStream  
)
```

## Parameters

### pHistoryStream

A pointer to the history stream.

## Requirements

### Requirements

**Platforms:** See [.NET Framework System Requirements](#).

#### DLL:

In the .NET Framework 3.0 and 3.5: PresentationHostDLL.dll

In the .NET Framework 4 and later: PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 3.0

## See also

- [WPF Unmanaged API Reference](#)

# SetFakeActiveWindow Function (WPF Unmanaged API Reference)

4/8/2019 • 2 minutes to read • [Edit Online](#)

This API supports the Windows Presentation Foundation (WPF) infrastructure and is not intended to be used directly from your code.

Used by the Windows Presentation Foundation (WPF) infrastructure for windows management.

## Syntax

```
void __stdcall SetFakeActiveWindow(  
    HWND hwnd  
)
```

## Parameters

hwnd

A window handle.

## Requirements

**Platforms:** See [.NET Framework System Requirements](#).

**DLL:** PresentationHost\_v0400.dll

**.NET Framework Version:** Available since 4

## See also

- [WPF Unmanaged API Reference](#)