

# LẬP TRÌNH WPF

# Giới thiệu WPF

- ▶ WPF (Windows Presentation Foundation) là hệ thống API mới hỗ trợ việc xây dựng giao diện đồ họa trên nền Windows.
- ▶ Được xem như thế hệ kế tiếp của WinForms, WPF tăng cường khả năng lập trình giao diện bằng cách cung cấp các API cho phép tận dụng những lợi thế về đa phương tiện hiện đại.
- ▶ Là một bộ phận của .NET Framework 3.0 trở về sau

# Giới thiệu WPF

- ▶ WPF được xây dựng nhằm vào ba mục tiêu cơ bản:
- ▶ Cung cấp một nền tảng thống nhất để xây dựng giao diện người dùng;
- ▶ Cho phép người lập trình và người thiết kế giao diện làm việc cùng nhau một cách dễ dàng;
- ▶ Cung cấp một công nghệ chung để xây dựng giao diện người dùng trên cả Windows và trình duyệt Web.

# Giới thiệu WPF

	Windows Forms	PDF	Windows Forms/ GDI+	Windows Media Player	Direct3D	WPF
Giao diện đồ họa (form, controls)	x					x
On-screen văn bản	x					x
Fixed-format văn bản		x				x
Hình ảnh			x			x
Video, âm thanh				x		x
2D			x			x
3D					x	x

# Các thành phần của WPF

- ▶ WPF tổ chức các chức năng theo một nhóm namespace cùng trực thuộc namespace System.Windows.
- ▶ Một ứng dụng WPF điển hình bao giờ cũng gồm một tập các trang XAML và phần code tương ứng được viết bằng C# hoặc Visual Basic.
- ▶ Tất cả các ứng dụng đều kế thừa từ lớp chuẩn Application của WPF. Lớp này cung cấp những dịch vụ chung cho mọi ứng dụng, chẳng hạn như các biến lưu trữ trạng thái của ứng dụng, các phương thức chuẩn để kích hoạt hay kết thúc ứng dụng.

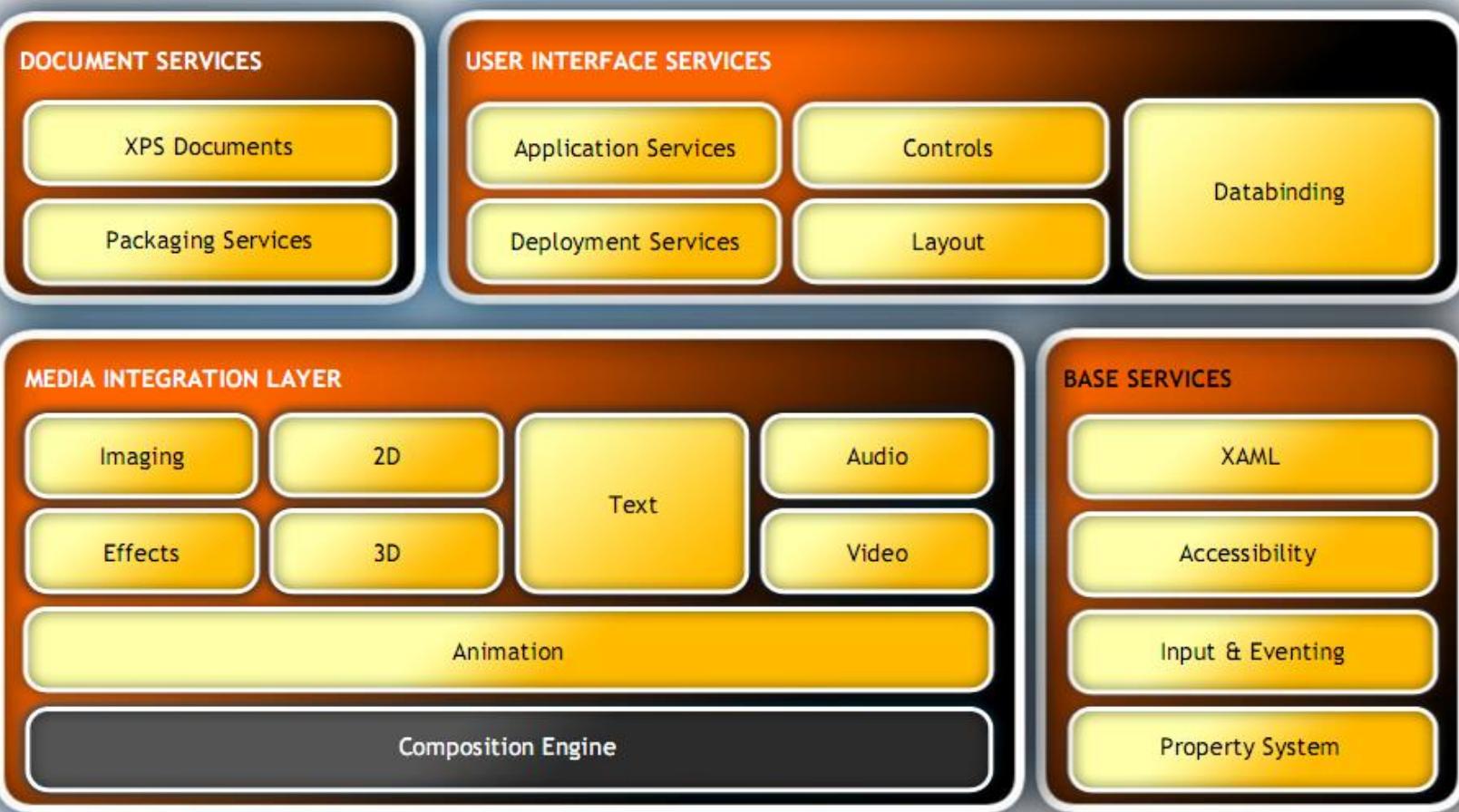
# Các thành phần của WPF

- ▶ Mặc dù WPF cung cấp một nền tảng thống nhất để tạo giao diện người dùng, những công nghệ mà WPF chưa đựng có thể phân chia thành những thành phần độc lập
- ▶ WPF được mở rộng với các tập tính năng phát triển ứng dụng bao gồm XAML, các control, cơ chế móc nối dữ liệu, layout, đồ họa 2 chiều, ba chiều, hoạt họa, style, khuôn dạng mẫu, văn bản, media, text và in ấn. WPF nằm trong .NET Framework, nên ngoài ra, ứng dụng WPF có thể kết hợp các thành phần khác có trong thư viện lớp của .NET Framework.

# Các thành phần của WPF

XPS Viewer

Windows Presentation Foundation



# Giới thiệu XAML

- ▶ XAML – Extensible Application Markup Language là một ngôn ngữ đánh dấu với cú pháp tương tự XML dùng để thể hiện các đối tượng trong .NET.
- ▶ Vai trò chính của XAML là dùng để xây dựng giao diện người dùng WPF. Nói cách khác, XAML documents sẽ định nghĩa cách sắp xếp, thể hiện các control, buttons trong cửa sổ của một chương trình WPF.

# Giới thiệu XAML

Ưu điểm của XAML:

- ▶ XAML có mã ngắn, rõ ràng dễ đọc.
- ▶ Tách mã thiết kế và logic.
- ▶ Việc tách XAML và giao diện người dùng logic cho phép tách biệt rõ ràng vai trò của nhà thiết kế và nhà phát triển.

# Cú pháp của XAML

- ▶ Mọi thành phần (elements) trong XAML document đều là một thể hiện của một lớp nào đó trong .NET. Tên của các elements này hoàn toàn giống với tên của các class. Ví dụ như thẻ `<Button>` trong WPF sẽ tạo ra một đối tượng thuộc lớp Button.
- ▶ Cũng như các tài liệu theo chuẩn XML khác, có thể gom (nest) một thẻ đặt bên trong một thẻ khác.
- ▶ Có thể thiết lập các property của mỗi class thông qua các attribute (thuộc tính).

# Cú pháp của XAML

Có 3 loại top-level element:

- ▶ Window
- ▶ Page (tương tự như Window nhưng dùng để chuyển đổi giữa các ứng dụng)
- ▶ Application (định nghĩa các application resources và thiết lập khởi động)

Ví dụ:

```
<Window x:Class="HelloWord.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

# XAML Namespaces

- ▶ XML namespace được khai báo như cách khai báo attributes.
- ▶ Các attribute này có thể được đặt bên trong bất cứ thẻ bắt đầu nào. Tuy nhiên, quy tắc là tất cả các namespace được sử dụng trong tài liệu nên được khai báo trong thẻ đầu tiên.
- ▶ Một khi đã khai báo namespace rồi, có thể sử dụng nó bất kì đâu trong tài liệu.

Ví dụ:

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

# XAML Namespaces

- ▶ <http://schemas.microsoft.com/winfx/2006/xaml/presentation> là core namespace của WPF. Nó chứa tất cả các class trong WPF, bao gồm các controls dùng để xây dựng giao diện. Nếu namespace này được khai báo không có prefix, như vậy, nó trở thành default namespace cho toàn bộ tài liệu. Nói cách khác, mọi element đều được tự động đặt trong namespace này, ngoại trừ một vài khai báo đặc biệt khác.
- ▶ <http://schemas.microsoft.com/winfx/2006/xaml/> là XAML namespace. Nó bao gồm các tính năng khác nhau của XAML cho phép can thiệp vào cách biên dịch tài liệu. Namespace này được khai báo với prefix x. Nghĩa là có thể áp dụng nó bằng cách đặt namespace prefix trước tên của thẻ (`<x:ElementName>`).

# The Code-Behind Class

Class attribute giúp cho XAML biết rằng cần phải phát triển một class mới với tên nhất định. Class mới này được kế thừa từ class có tên như tên của XML element.

Ví dụ:

```
<Window x:Class="WindowsApplication1.Window1">
```

Ở đây, chương trình sẽ tạo ra một class mới có tên là Window1 và class này kế thừa từ class Window

# Naming Elements

Trong code-behind class, nếu muốn đọc hoặc thay đổi properties hoặc những công việc khác liên quan đến xử lý sự kiện. Để làm được việc này, các control phải được đặt tên thông qua thuộc tính Name.

Ví dụ:

```
<Window x:Class= "WindowApplication1.Window1"  
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presen  
tation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
Title="Window1" Height="350" Width="525">  
    <Grid x:name= "grid1">  
        </Grid>  
</Window>
```

Bây giờ, có thể thao tác với grid trong class Window1 thông qua tên grid1:

# Naming Elements

Có thể đặt tên cho element theo kiểu XAML Name property (sử dụng prefix x:) hoặc sử dụng Name property của riêng element đó (bỏ prefix đi). Cả hai cách này đều cho kết quả tương tự trong đa số các trường hợp.

Ví dụ:

```
<Grid Name="grid1">  
</Grid>
```

# Property

Các property của những đối tượng có thể được xem như những attribute trong các thẻ của XAML.

- ▶ Đối với các property có kiểu đơn giản, có thể gán trực tiếp bằng cách dùng attribute với giá trị nằm trong cặp nháy kép.

Ví dụ: Title="Hello XAML" Height="250" Width="250"

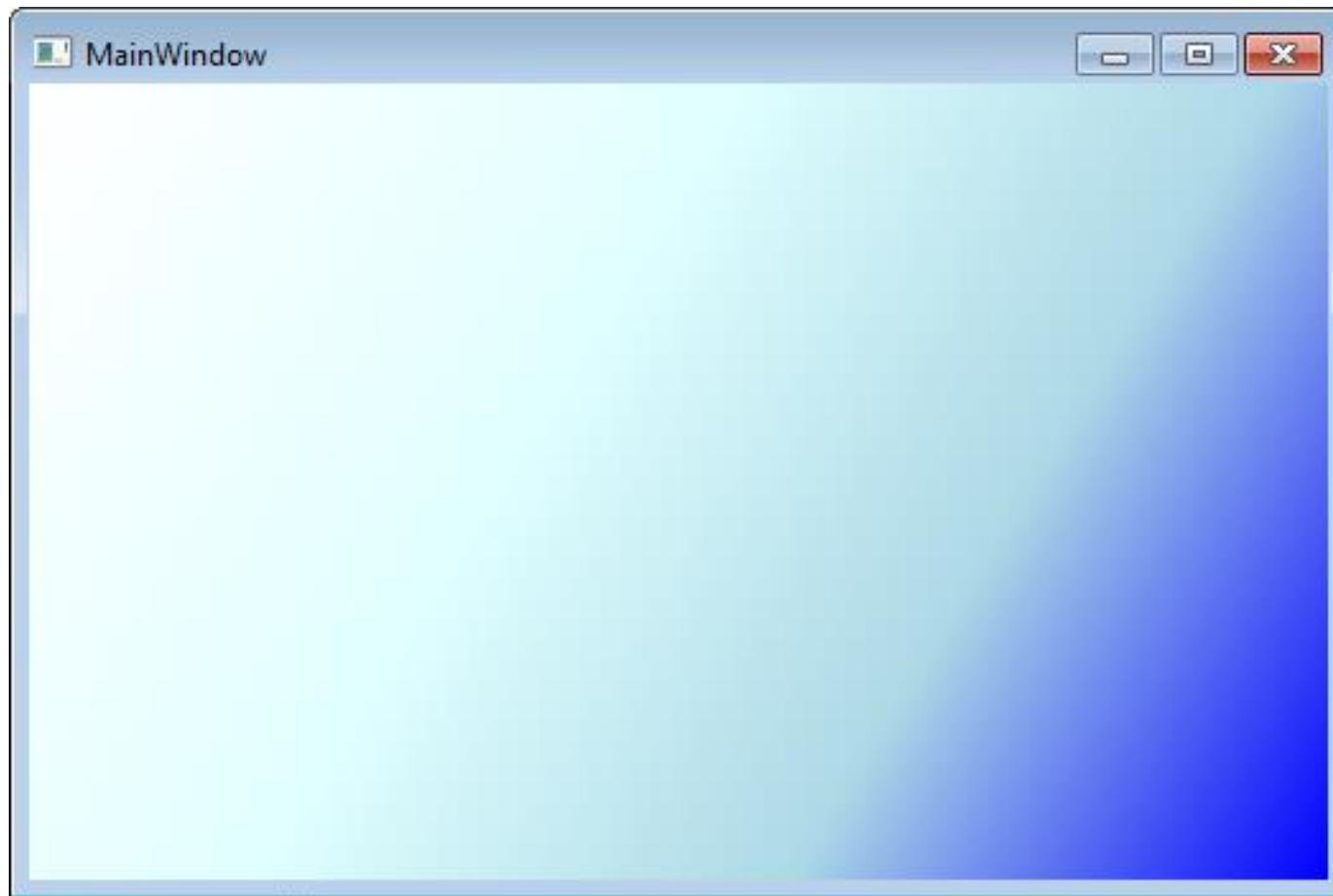
- ▶ Đối Với các property phức tạp cần phải sử dụng các thẻ để định nghĩa, các thẻ này phải nằm bên trong thẻ khai báo property của đối tượng.

# Property

Ví dụ: tô màu nền gradient cho Window bằng cách đặt thuộc tính Background của Window với một LinearGradientBrush. LinearGradientBrush lại chứa property GradientStops là một collection các điểm dừng để xác định màu.

```
<Window.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1, 1">
        <LinearGradientBrush.GradientStops>
            <GradientStopCollection>
                <GradientStop Offset="0" Color="White" />
                <GradientStop Offset="0.2" Color="Azure" />
                <GradientStop Offset="0.4" Color="LightCyan" />
                <GradientStop Offset="0.7" Color="LightBlue" />
                <GradientStop Offset="1" Color="Blue" />
            </GradientStopCollection>
        </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
</Window.Background>
```

# Property



# CLR Property và Dependency Property

CLR property là các property thường sử dụng sử dụng trong .NET được hiện thực bằng cách sử dụng một private field và một “wrapper” (get, set) để lưu trữ và truy xuất giá trị.

Trong khi đó, dependency property lưu trữ các giá trị trong một dictionary gồm các dòng dữ liệu dạng key/value, với key là tên của property và value là giá trị lưu trữ.

# Dependency Property

Lợi ích của dependency property là không phải tốn bộ nhớ lưu trữ các private field mà đa số chúng chỉ mang giá trị mặc định. Dependency property chỉ chỉ lưu trữ các giá trị bị thay đổi.

Một số giá trị mặc định mặc định của các kiểu dữ liệu khi dependency property:

- ▶ Kiểu tham chiếu (bao gồm string): null
- ▶ Kiểu structure: dựa vào default constructor
- ▶ Kiểu số: zero

# Dependency Property

Trong trường hợp phần tử hiện tại không chứa giá trị của property cần lấy, dependency property sẽ duyệt lên các phần tử ở mức cao hơn để tìm giá trị (ví dụ Button > Grid > Window).

Ví dụ như khi gán FontSize cho Window, các thành phần con của nó cũng được áp dụng giá trị FontSize này, tính năng này được gọi là Value Inheritance.

# Dependency Property

Các dependency property có thể được truy xuất trực tiếp thông qua tên của chúng trong tài liệu XAML. Trong code-behind, chúng được truy xuất thông qua hai phương thức GetValue() và SetValue().

## Trong XAML:

Với kiểu dữ liệu đơn giản, Có thể gán trực tiếp như các attribute của thẻ, như property Width sau:

```
<Button Width="100">Button1</Button>
```

Cũng có thể dùng cú pháp khác, sử dụng các thẻ lồng nhau:

```
<Button Content="Button1">  
    <Button.Width>100</Button.Width>  
</Button>
```

# Dependency Property

## Trong Code-Behind

Bởi vì các thành phần WPF đã tạo sẵn các wrapper (get/set), có thể sử dụng như các CLR property thông thường:

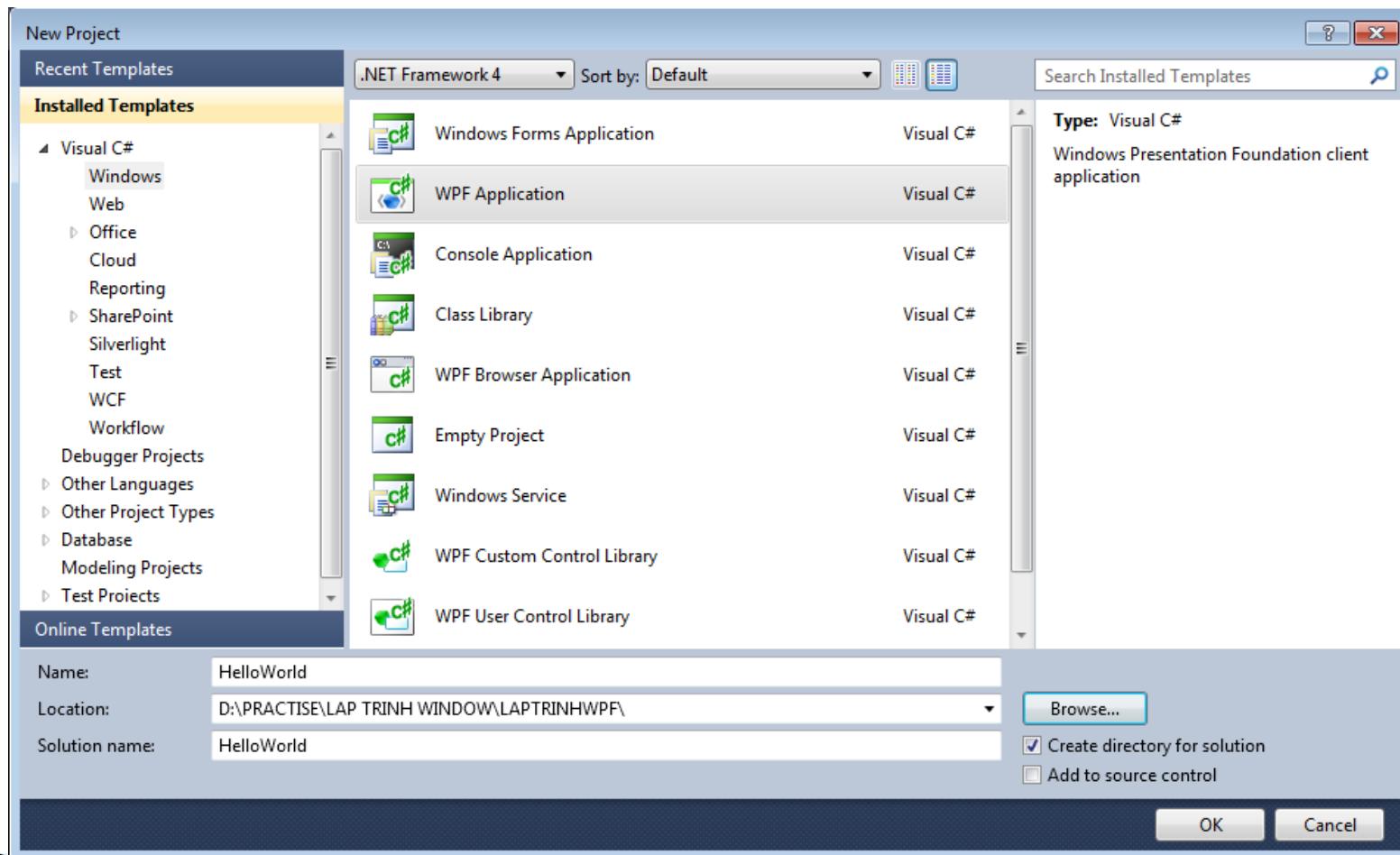
```
double width = button1.Width; button1.Width = 100;
```

Cũng có thể sử dụng hai phương thức GetValue(DependencyProperty) và SetValue(DependencyProperty, value)

```
double width = (double)button1.GetValue(Button.WidthProperty);  
button1.SetValue(Button.WidthProperty, 100d);
```

# Hello World

## Tạo một Project WPF mới



# Hello World

Mở file MainWindow.xaml thêm đoạn mã đặc tả XAML:

```
<Window x:Class="HelloWorld.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300">
    <Button Name="button" Margin="10" Click="button_Click">
        <Button.LayoutTransform>
            <ScaleTransform ScaleX="3" ScaleY="3">
                </ScaleTransform>
        </Button.LayoutTransform>
        Hello World
    </Button>
</Window>
```

# Hello World

Trong file MainWindow.xaml.cs thêm câu lệnh:

```
private void button_Click(object sender, RoutedEventArgs e)
{
    button.Content = "Welcome";
}
```



# Layout

Cách bố trí (Layout) là một vấn đề quan trọng và không thể thiếu trong việc thiết kế giao diện.

Các layout container này được kế thừa từ lớp System.Windows.Controls.Panel và có cách sắp xếp và bố trí các control bên trong nó khác nhau.

Một số layout thông dụng nhất bao gồm:

- ▶ StackPanel
- ▶ WrapPanel
- ▶ DockPanel
- ▶ Canvas
- ▶ Grid

# StackPanel

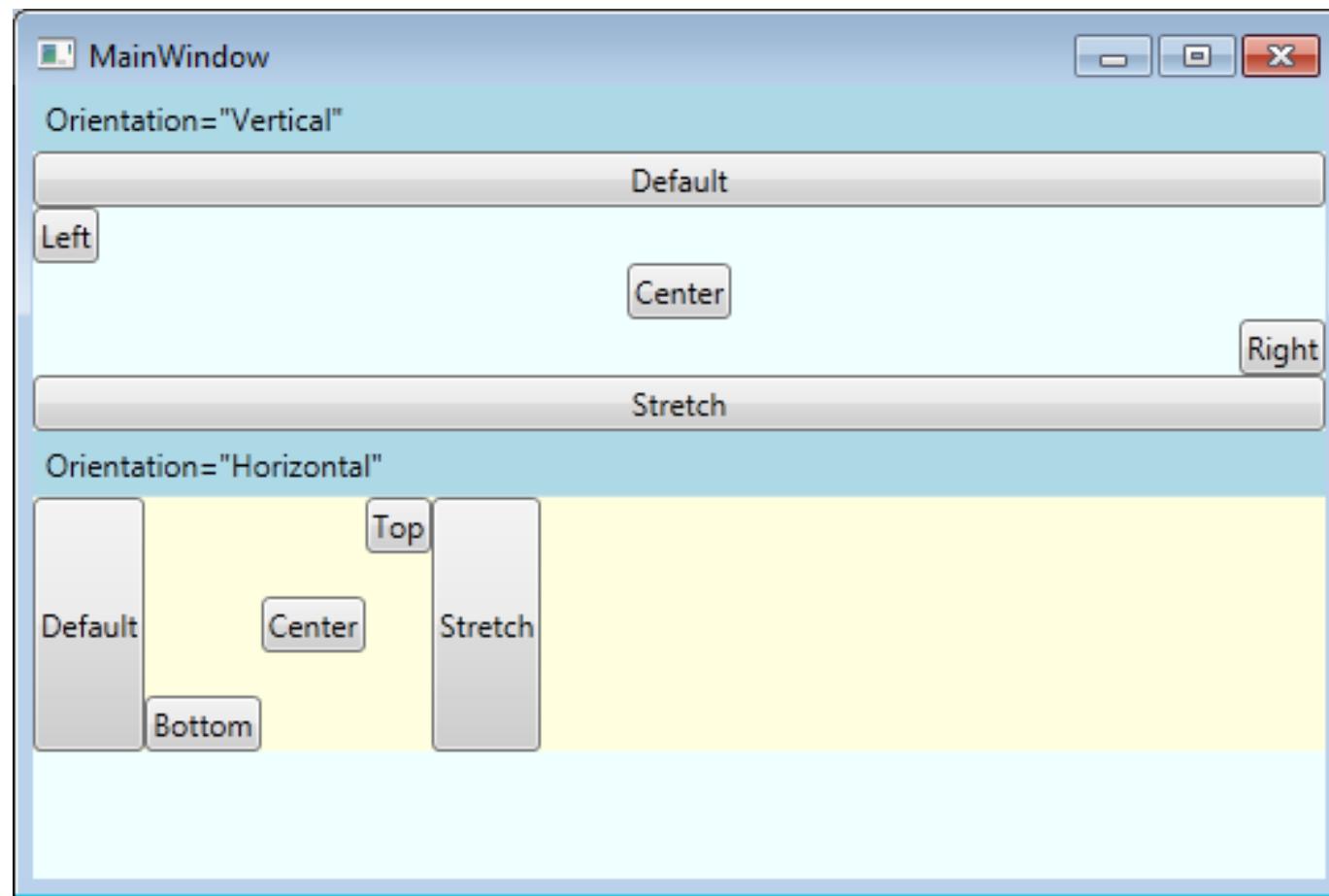
- ▶ Control này sẽ sắp xếp các control con của nó theo dòng hoặc cột tùy theo giá trị của thuộc tính Orientation là Vertical hay Horizontal. Các control con sẽ được sắp xếp với vị trí liên tiếp và không chồng lê nhau.
- ▶ Giá trị mặc định của thuộc tính Orientation là Vertical, các control sẽ được sắp xếp theo chiều dọc từ trên xuống dưới. Ngược lại là Horizontal, chúng sẽ được sắp xếp từ trái sang phải.

Ví dụ: Hai StackPanel lồng nhau, StackPanel bên ngoài sẽ co giãn khít với kích thước của Window và xếp các control theo chiều dọc và StackPanel bên trong xếp theo chiều ngang:

# StackPanel

```
<StackPanel Orientation="Vertical" Background="Azure">
    <Label Background="LightBlue">Orientation="Vertical"</Label>
    <Button>Default</Button>
    <Button HorizontalAlignment="Left">Left</Button>
    <Button HorizontalAlignment="Center">Center</Button>
    <Button HorizontalAlignment="Right">Right</Button>
    <Button HorizontalAlignment="Stretch">Stretch</Button>
    <Label Background="LightBlue">Orientation="Horizontal"</Label>
    <StackPanel Orientation="Horizontal" Background="LightYellow" Height="100">
        <Button>Default</Button>
        <Button VerticalAlignment="Bottom">Bottom</Button>
        <Button VerticalAlignment="Center">Center</Button>
        <Button VerticalAlignment="Top">Top</Button>
        <Button VerticalAlignment="Stretch">Stretch</Button>
    </StackPanel>
</StackPanel>
```

# StackPanel



# WrapPanel

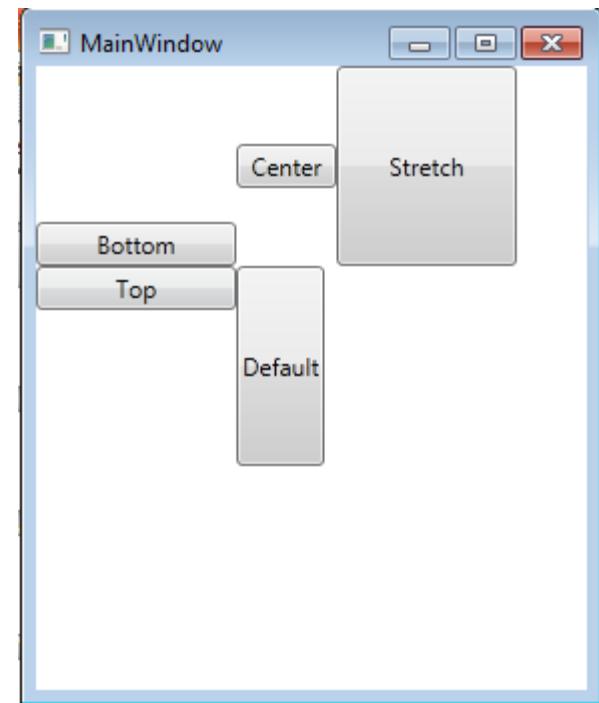
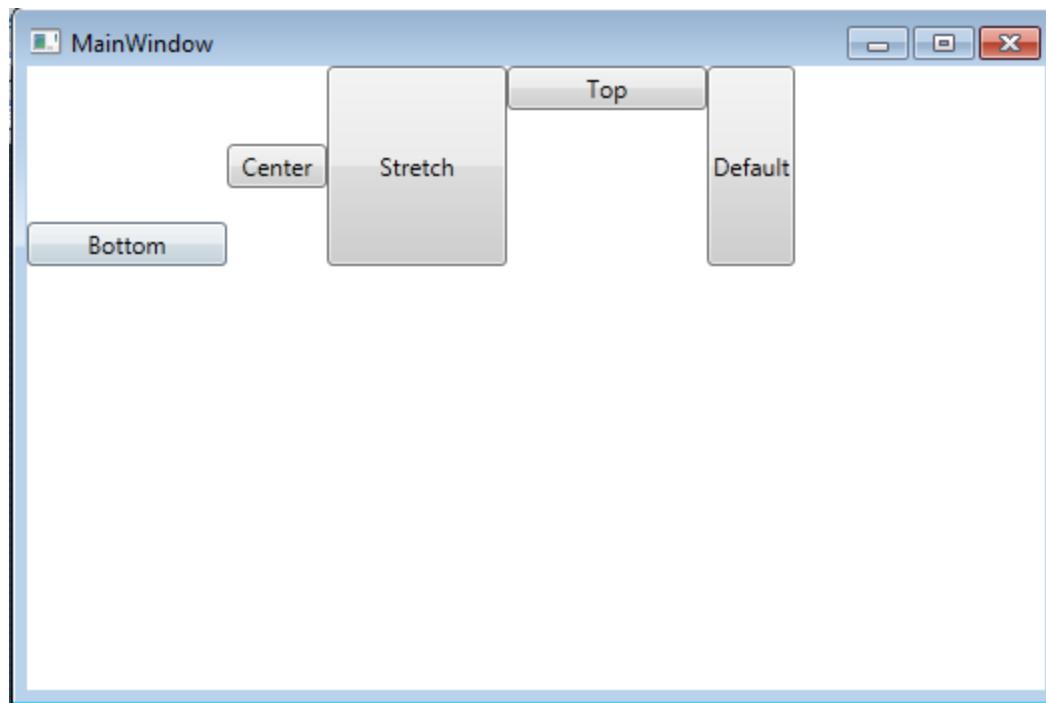
Cũng sắp xếp các control lần lượt theo hàng hoặc cột, nhưng đặc điểm chính của WrapPanel là sẽ tự động cho các control sang hàng/cột mới nếu như kích thước của hàng/cột còn lại không đủ chứa control.

Để thay đổi chiều sắp xếp các control con, sử dụng thuộc tính Orientation với hai giá trị là Horizontal và Vertical; giá trị mặc định là Horizontal.

# WrapPanel

```
<WrapPanel Orientation="Horizontal">
    <Button VerticalAlignment="Bottom" Width="100">Bottom</Button>
    <Button VerticalAlignment="Center" Width="50">Center</Button>
    <Button VerticalAlignment="Stretch" Width="90"
        Height="100">Stretch</Button>
    <Button VerticalAlignment="Top" Width="100" >Top</Button>
    <Button Height="100">Default</Button>
</WrapPanel>
```

# WrapPanel



# DockPanel

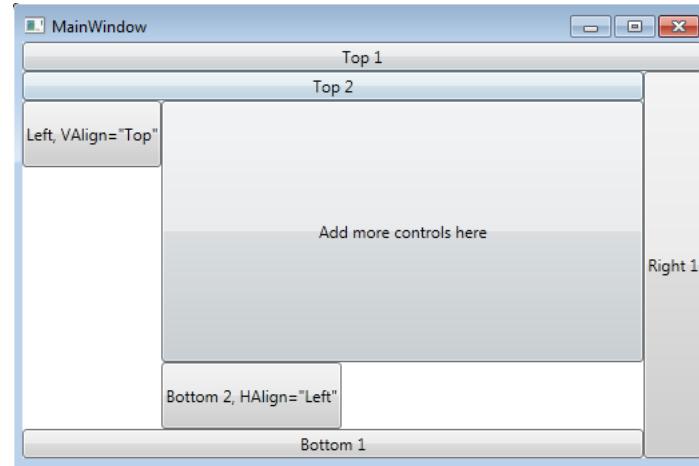
Khi sử dụng container này, các control con sẽ được gắn thêm một attached property là DockPanel.Dock cho phép gán các giá trị: Left, Right, Top, Bottom. Các giá trị tương ứng với mỗi phần không gian trong DockPanel, phần còn lại ở giữa sẽ được dùng để chứa các control khác.

Thuộc tính LastChildFill có giá trị mặc định true nhằm xác định các control thêm vào sau sẽ lấp đầy khoảng trống còn lại. Nếu muốn giữ lại khoảng trống này, hãy gán giá trị của nó trở thành false.

# DockPanel

```
<DockPanel LastChildFill="True">
    <Button DockPanel.Dock="Top">Top 1</Button>
    <Button DockPanel.Dock="Right">Right 1</Button>
    <Button DockPanel.Dock="Top">Top 2</Button>
    <Button DockPanel.Dock="Bottom">Bottom 1</Button>
    <Button DockPanel.Dock="Left" VerticalAlignment="Top"
        Height="50">Left, VAlign="Top"</Button>
    <Button DockPanel.Dock="Bottom" HorizontalAlignment="Left"
        Height="50">Bottom 2, HAlign="Left"</Button>
    <Button>Add more controls here</Button>
</DockPanel>
```

# DockPanel



# Canvas

Canvas cho phép bố trí các control bằng cách xác định vị trí cố định của chúng. Sử dụng cách này khiến cho giao diện trở nên thiếu linh hoạt và gây ra nhiều hạn chế, như khi độ phân giải hoặc kích thước của sổ thay đổi.

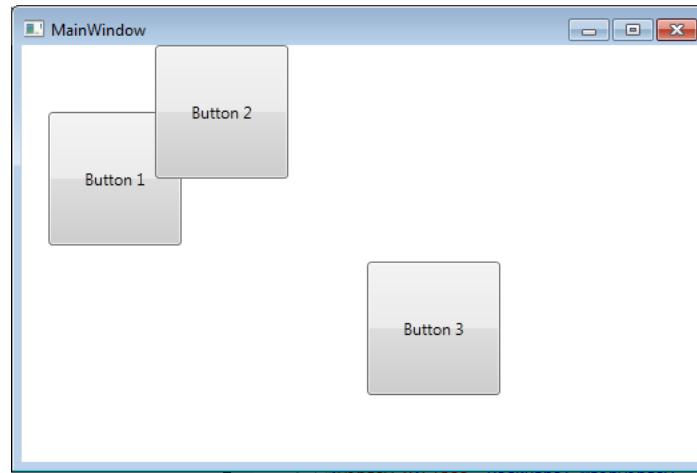
Trong Canvas, gán tọa độ cho các control con thông qua các thuộc tính Canvas.Left, Canvas.Right, Canvas.Top, Canvas.Bottom.

Left và Top có độ ưu tiên cao hơn Right và Bottom. Nghĩa là nếu gán giá trị cho cả Canvas.Left và Canvas.Right, khi đó tọa độ theo chiều ngang của control sẽ lấy từ Canvas.Left, tương tự với Canvas.Top và Canvas.Bottom.

# Canvas

```
<Canvas Background="White">
    <Button Width="100" Height="100"
        Canvas.Left="20" Canvas.Top="50">Button 1</Button>
    <Button Width="100" Height="100"
        Canvas.Left="100" Canvas.Right="100">Button 2</Button>
    <Button Width="100" Height="100"
        Canvas.Right="150" Canvas.Bottom="50">Button 3</Button>
</Canvas>
```

# Canvas



# Grid Layout

- ▶ Là một layout control linh hoạt và hiệu quả nhất, Grid bố trí các control bên trong nó theo dạng bảng gồm các dòng và cột.
- ▶ Ngoài ra, có thể thay đổi kích thước của dòng và cột trong quá trình thực thi với GridSplitter.
- ▶ Mặc định các đường lưới này sẽ không hiển thị, để hiển thị chúng chỉ cần thay đổi giá trị của thuộc tính Grid.ShowGridLines thành true.

# Grid Layout

## Định nghĩa dòng và cột:

Khi làm việc với Grid layout cần phải định nghĩa cấu trúc bao gồm các dòng và cột cũng như kích thước của chúng. Các đối tượng dòng, cột này được lưu trong hai collection là Grid.RowDefinitions và Grid.ColumnDefinitions.

Kích thước của dòng, cột được xác định bằng ba cách:

Cố định: sử dụng giá trị số với đơn vị là 1/96 inch.

Auto: Dòng/cột sẽ có kích thước vừa đủ để chứa các control bên trong.

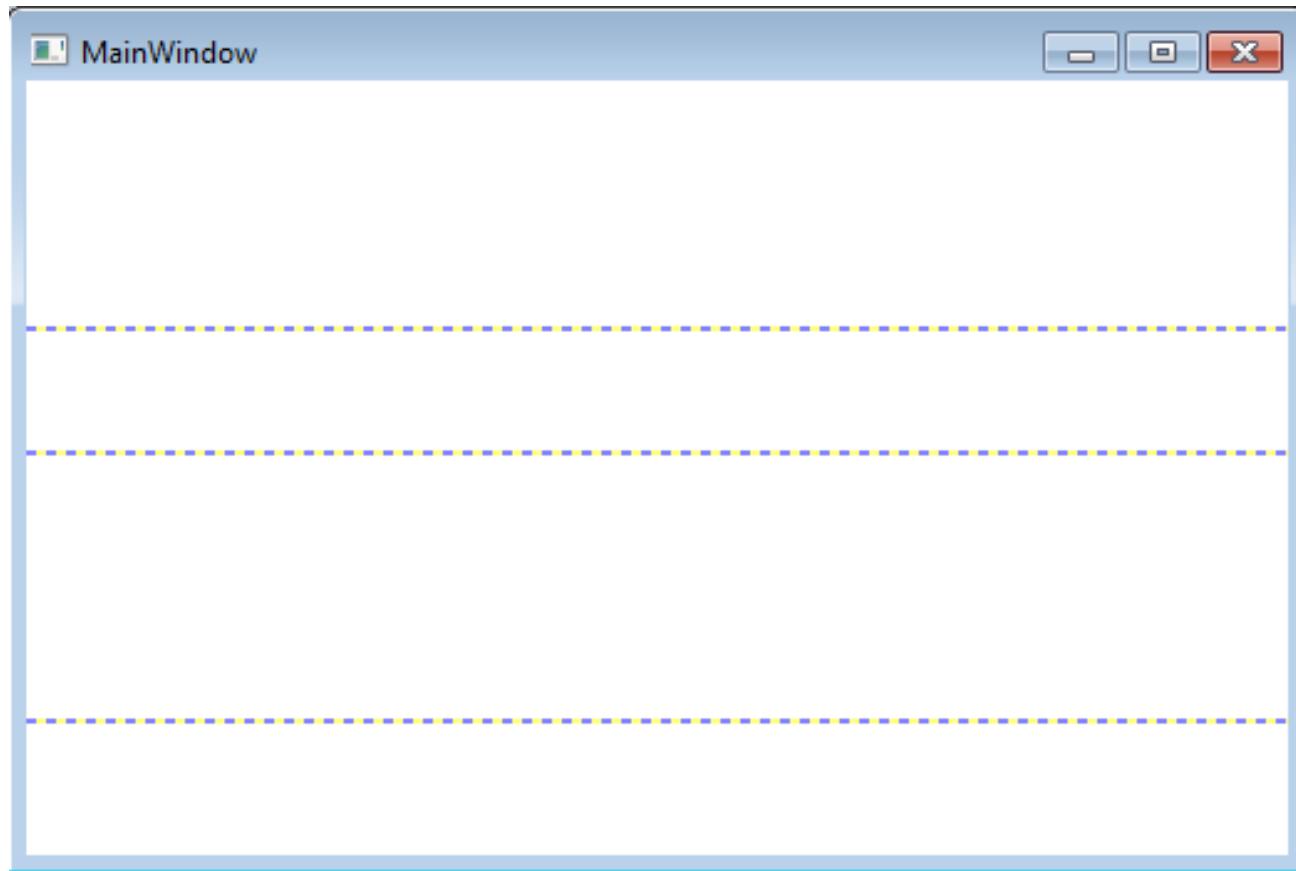
Tỷ lệ: Dùng dấu sao (\*) để xác định tỉ lệ. Giá trị của \* được tính bằng kích thước còn lại của Grid chia cho tổng số phần được chia.

# Grid Layout

Ví dụ tạo Grid control với bốn dòng. Dòng đầu có chiều cao 100. Dòng thứ hai có chiều cao bằng 50. Chiều cao còn lại của Grid được chia thành 3 phần: 2 phần cho dòng thứ ba và 1 phần cho dòng cuối.

```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="50" />
        <RowDefinition Height="*" />
        <RowDefinition Height="3*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
</Grid>
```

# Grid Layout



# Grid Layout

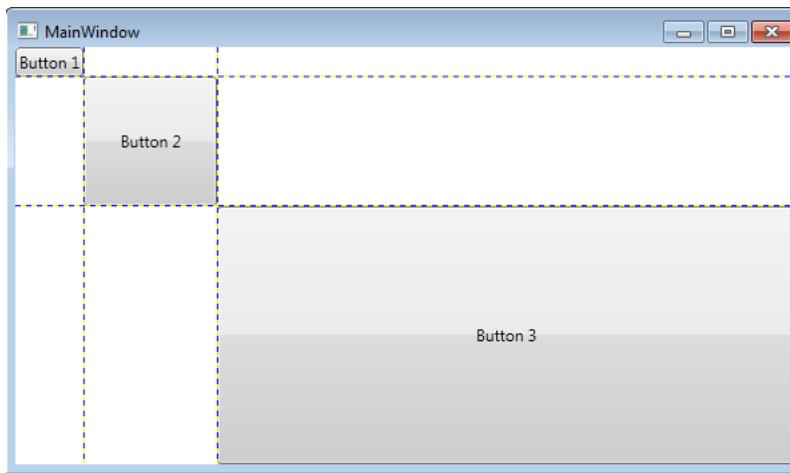
## Bố trí các control trong Grid

Khi được thêm vào trong Grid, các control có thể sử dụng hai attached property là Grid.Row và Grid.Column để xác định chỉ số dòng/cột của ô sẽ chứa chúng. Các chỉ số dòng/cột này được tính từ 0.

# Grid Layout

```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="2*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="100" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Button Grid.Row="0" Grid.Column="0">Button 1</Button>
    <Button Grid.Row="1" Grid.Column="1">Button 2</Button>
    <Button Grid.Row="2" Grid.Column="2">Button 3</Button>
</Grid>
```

# Grid Layout



# Grid Layout

## Spanning dòng và cột

Sử dụng hai attached property là Grid.RowSpan và Grid.ColumnSpan để xác định số lượng ô theo dòng/cột mà một control có thể nằm trong đó.

Một điểm lưu ý là các ô của Grid không bị gộp thành một như trong html table, chỉ có control là được mở rộng thêm không gian chứa nó qua các ô khác. Vì vậy, có thể truy xuất đến các ô theo chỉ số như thứ tự chúng được định nghĩa.

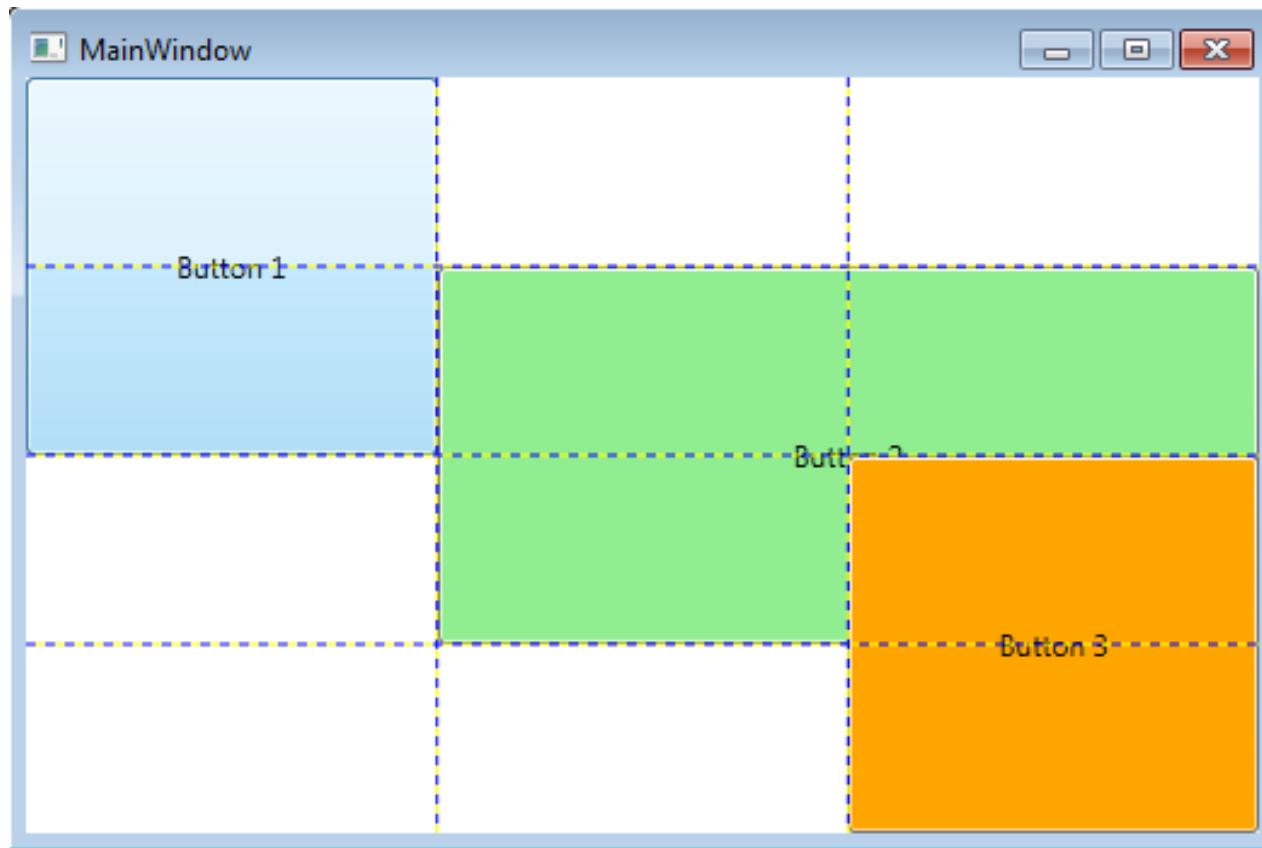
# Grid Layout

```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
```

# Grid Layout

```
<Button Background="Azure" Grid.Row="0" Grid.Column="0"  
    Grid.RowSpan="2">Button 1</Button>  
<Button Background="LightGreen" Grid.Row="1" Grid.Column="1"  
    Grid.RowSpan="2" Grid.ColumnSpan="2">Button 2</Button>  
<Button Background="Orange" Grid.Row="2" Grid.Column="2"  
    Grid.RowSpan="2">Button 3</Button>  
</Grid>
```

# Grid Layout



# Grid Layout

## Thay đổi kích thước dòng và cột (lúc runtime)

Bằng cách thêm các control GridSplitter vào Grid, có thể di chuyển các GridSplitter và điều này làm cho kích thước của các dòng/cột thay đổi theo. Một vài thuộc tính của GridSplitter cần quan tâm:

- ▶ ResizeDirection: hướng thay đổi kích thước, gồm ba giá trị: Auto, Column và Row.
- ▶ ShowsPreview: giá trị false sẽ cập nhật sự thay đổi kích thước của các dòng/cột khi di chuyển GridSplitter.
- ▶ Height/Width và VerticalAlignment/HorizontalAlignment: sử dụng để xác định kích thước, canh lề của GridSplitter tùy theo hướng di chuyển.

# Grid Layout

Ví dụ sau cho thấy việc sử dụng hai GridSplitter. GridSplitter đầu tiên được chèn vào cột thứ hai và GridSplitter thứ hai chèn vào dòng thứ hai trong Grid. Để ý rằng trong phần định nghĩa, xác định kích thước của dòng/cột vừa đủ để chứa các GridSplitter.

# Grid Layout

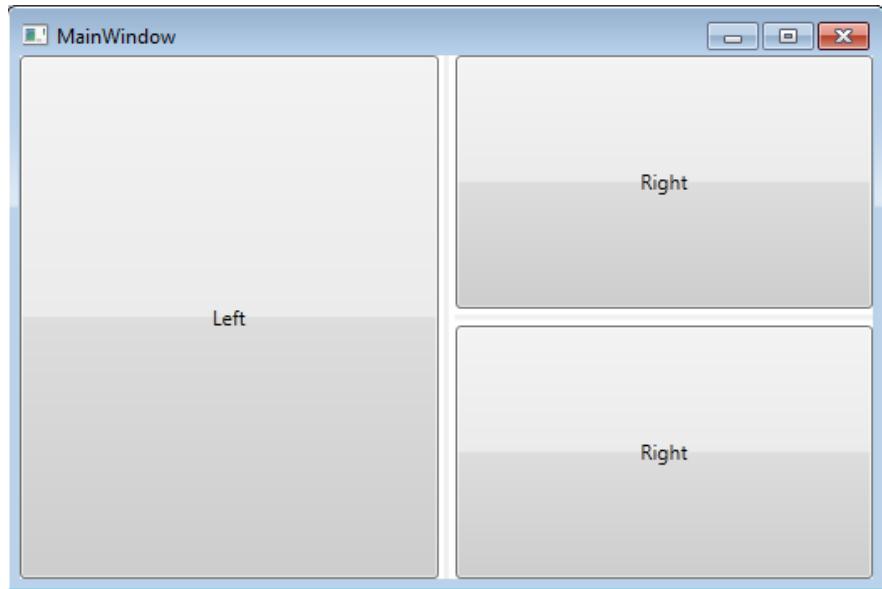
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="10" />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition Width="10" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
```

# Grid Layout

```
<Button Grid.Row="0" Grid.Column="0"
        Grid.RowSpan="3">Left</Button>
<Button Grid.Row="0" Grid.Column="2">Right</Button>
<Button Grid.Row="2" Grid.Column="2">Right</Button>

<GridSplitter Grid.Row="0" Grid.Column="1" Grid.RowSpan="3"
              Width="3" HorizontalAlignment="Center"
              ShowsPreview="False"></GridSplitter>
<GridSplitter Grid.Row="1" Grid.Column="2" Height="3"
              VerticalAlignment="Center" HorizontalAlignment="Stretch"
              ShowsPreview="False" ResizeDirection="Auto"></GridSplitter>
</Grid>
```

# Grid Layout



# Grid Layout

## Đồng bộ kích thước các dòng, cột của nhiều Grid

Khi định nghĩa dòng/cột, có thể dùng thuộc tính SharedSizeGroup nhằm xác định một tên nhóm để các dòng/cột thuộc cùng nhóm luôn có kích thước bằng nhau, mặc dù chúng nằm trong các Grid control khác nhau.

Khi dùng phương pháp này, cần đặt thuộc tính Grid.IsSharedSizeScope thành true cho control chứa các Grid cần đồng bộ kích thước dòng/cột. Thuộc tính Grid.IsSharedSizeScope sẽ xác định rằng các control con của nó thuộc phạm vi chia sẻ kích thước với nhau.

Ví dụ sau cho thấy cách để hai cột đầu tiên của hai Grid control luôn có chiều rộng bằng nhau:

# Grid Layout

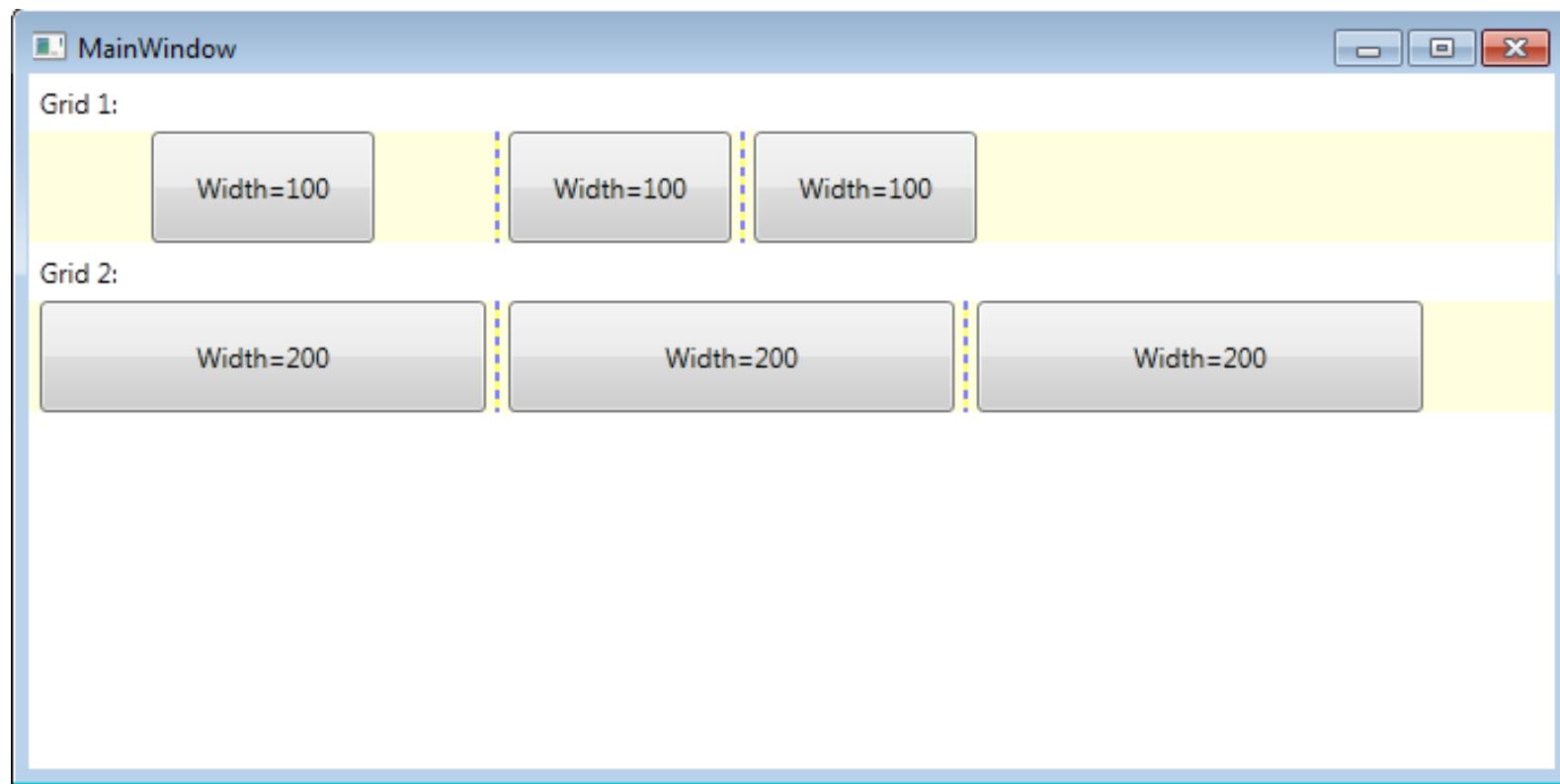
```
<StackPanel Grid.IsSharedSizeScope="True">  
    <Label>Grid 1:</Label>  
    <Grid Background="LightYellow" ShowGridLines="True">  
        <Grid.ColumnDefinitions>  
            <ColumnDefinition SharedSizeGroup="ColGroup" />  
            <ColumnDefinition Width="Auto" />  
            <ColumnDefinition Width="Auto" />  
        </Grid.ColumnDefinitions>  
        <Button Width="100" Height="50">Width=100</Button>  
        <Button Width="100" Height="50" Margin="5,0,5,0"  
               Grid.Column="1" >Width=100</Button>  
        <Button Width="100" Height="50" Margin="5,0,5,0"  
               Grid.Column="2" >Width=100</Button>  
    </Grid>
```

# Grid Layout

```
<Label>Grid 2:</Label>
<Grid Background="LightYellow" ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition SharedSizeGroup="ColGroup" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <Button Width="200" Height="50"
        Margin="5,0,5,0">Width=200</Button>
    <Button Width="200" Height="50" Margin="5,0,5,0"
        Grid.Column="1" >Width=200</Button>
    <Button Width="200" Height="50" Margin="5,0,5,0"
        Grid.Column="2" >Width=200</Button>
</Grid>
</StackPanel>
```

# Grid Layout



# Một số Controls thông dụng

- ▶ **Label:** Nhãn.
- ▶ **TextBox:** Hộp soạn thảo.
- ▶ **Button:** Nút bấm.
- ▶ **CheckBox:** Hộp chọn.
- ▶ **RadioButton:** Hộp chọn radio.
- ▶ **ListBox:** Hộp danh sách
- ▶ **ComboBox:** Hộp danh sách thả xuồng..

# Label

Label là các điều khiển để hiển thị các văn bản tĩnh, thường được sử dụng để làm nhãn cho các control khác như Textbox, ListBox, ComboBox,....

```
<Label Height="30" HorizontalAlignment="Left" Margin="10,15,0,0"  
Name="label1" VerticalAlignment="Top" Width="60">Họ đệm:</Label>
```

- ▶ Height="30" : Độ cao của khung nhãn là 30px
- ▶ HorizontalAlignment="Left" : Nhãn được căn trái trong cửa sổ
- ▶ Margin="10,15,0,0" : có 4 giá trị là Left,Top,Right,Bottom
- ▶ Name="label1" : Tên của nhãn là label1
- ▶ VerticalAlignment="Top" : Nhãn được căn theo đỉnh của cửa sổ.
- ▶ Width="60": Chiều rộng của nhãn là 60px

# TextBox

Hộp soạn thảo (TextBox) là control cho phép người dùng nhập dữ liệu dạng văn bản.

```
<TextBox Height="30" Margin="80,17,30,0" Name="textBox1"  
VerticalAlignment="Top" />Hộp soạn thảo  
</TextBox>
```

- ▶ Margin="80,17,30,0": Cách lề trái 80, đỉnh cửa sổ 17, cạnh phải 30
- ▶ Name="textBox1": Tên của hộp soạn thảo là textBox1
- ▶ VerticalAlignment="Top": Căn theo đỉnh cửa sổ

# Button

Button là loại điều khiển cho phép người dùng nhấn chuột để chọn lệnh, khi nhấn vào nút bấm, nó sẽ sinh ra sự kiện Click và sẽ chạy các lệnh gắn với sự kiện này.

```
<Button Height="35" HorizontalAlignment="Left" Margin="16,0,0,27"  
Name="button1" VerticalAlignment="Bottom" Width="110"  
Click="button1_Click">Button</Button>
```

- ▶ Height="35": Chiều cao nút bấm là 35
- ▶ Width="110": Chiều rộng là 110
- ▶ HorizontalAlignment="Left": Căn theo lề trái
- ▶ VerticalAlignment="Bottom": Căn theo đáy của sổ
- ▶ Margin="16,0,0,27": Cách lề trái 16, cách đáy 27
- ▶ Name="button1": Tên nút bấm là button1

Click="button1\_Click": Khi nhấn chuột vào nút sẽ kích hoạt phương thức button1\_Click()

# Radio Button và CheckBox

Radio Button: là hộp chọn theo nhóm, tại một thời điểm người dùng chỉ được chọn một trong các mục.

```
<RadioButton Height="22" Name="radioButton1" Width="79"  
GroupName="GioiTinh" IsChecked="True">Nam</RadioButton>  
<RadioButton Height="22" Name="radioButton2" Width="79"  
GroupName="GioiTinh">Nữ</RadioButton>
```

CheckBox: là hộp chọn mà người dùng có thể chọn một hoặc nhiều mục cùng một lúc.

```
<CheckBox Name="checkBox1" Height="20" IsChecked="True">Tiếng  
Anh</CheckBox>  
<CheckBox Name="checkBox2" Height="20">Tiếng Trung</CheckBox>
```

# ListBox

Hộp danh sách (ListBox) là điều khiển hiển thị một danh sách các mục theo từng dòng và cho phép người dùng chọn một hay nhiều phần tử của danh sách.

```
<ListBox Height="68" Name="listBox1" SelectedIndex="0">
    <ListBoxItem>Hà nội</ListBoxItem>
    <ListBoxItem>TP. Hồ Chí Minh</ListBoxItem>
    <ListBoxItem>Hải Phòng</ListBoxItem>
    <ListBoxItem>Đà Nẵng</ListBoxItem>
</ListBox>
```

SelectedIndex="k" để yêu cầu tự động chọn phần tử thứ k+1 trong danh sách khi mở cửa sổ. Phần tử đầu tiên của danh sách có giá trị là 0, phần tử cuối cùng là n-1. Nếu muốn khi mở cửa sổ không chọn phần tử nào thì đặt giá trị k bằng -1.

# ComboBox

Hộp danh sách thả xuống (ComboBox) là điều khiển hiển thị một danh sách theo từng dòng cho người dùng chọn. Tuy nhiên, khác với ListBox, ComboBox gọn gàng hơn bởi vì nó chỉ hiển thị 1 dòng và khi nhấn vào biểu tượng tam giác bên cạnh thì danh sách mới được mở ra. Combox chỉ cho phép chọn 1 dòng tại 1 thời điểm.

```
<ComboBox Height="26" Margin="84,0,27,126" Name="comboBox1"
VerticalAlignment="Bottom" SelectedIndex="0">
    <ComboBoxItem>Hà nội</ComboBoxItem>
    <ComboBoxItem>TP. Hồ Chí Minh</ComboBoxItem>
    <ComboBoxItem>Hải Phòng</ComboBoxItem>
    <ComboBoxItem>Đà Nẵng</ComboBoxItem>
</ComboBox>
```

# Ví dụ

## File XAML

```
01 <Window x:Class="Controls.MainWindow"
02     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
03     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
04     Title="Ví dụ Control - WPF" Height="352" Width="300">
05     <Grid>
06         <Label Height="30" HorizontalAlignment="Left" Margin="10,15,0,0"
Name="label1" VerticalAlignment="Top" Width="60" Focusable="False">Họ
đệm:</Label>
07         <Label Height="30" HorizontalAlignment="Left" Margin="10,50,0,0"
Name="label2" VerticalAlignment="Top" Width="60">Tên:</Label>
08         <TextBox Height="30" Margin="80,17,30,0" Name="textBox1"
VerticalAlignment="Top">Hộp soạn thảo</TextBox>
09         <TextBox Height="30" Margin="80,52,30,0" Name="textBox2"
VerticalAlignment="Top" />
10         <Button Height="35" HorizontalAlignment="Left" Margin="16,0,0,27"
Name="button1" VerticalAlignment="Bottom" Width="110"
Click="button1_Click">Xem thông tin</Button>
```

# Ví dụ

## File XAML

```
11      <Button Height="35" HorizontalAlignment="Right" Margin="0,0,24,27"
Name="button2" VerticalAlignment="Bottom" Width="110"
Click="button2_Click">Nhập lại</Button>
12      <Label Height="30" HorizontalAlignment="Left" Margin="10,94,0,0"
Name="label3" VerticalAlignment="Top" Width="60">Giới tính:</Label>
13      <RadioButton Height="22" Margin="80,99,0,0" Name="radioButton1"
VerticalAlignment="Top" HorizontalAlignment="Left" Width="79"
GroupName="GioiTinh" IsChecked="True">Nam</RadioButton>
14      <RadioButton Height="22" HorizontalAlignment="Right"
Margin="0,99,30,0" Name="radioButton2" VerticalAlignment="Top" Width="79"
GroupName="GioiTinh">Nữ</RadioButton>
15      <Label HorizontalAlignment="Left" Margin="10,127,0,154"
Name="label4" Width="69">Ngoại ngữ:</Label>
16      <CheckBox Margin="84,132,119,0" Name="checkBox1" Height="20"
VerticalAlignment="Top" IsChecked="True">Tiếng Anh</CheckBox>
17      <CheckBox HorizontalAlignment="Right" Margin="0,132,24,0"
Name="checkBox2" Width="85" Height="20" VerticalAlignment="Top">Tiếng
Trung</CheckBox>
```

# Ví dụ

## File XAML

```
18      <Label Height="30" HorizontalAlignment="Left" Margin="10,0,0,126"
Name="label5" VerticalAlignment="Bottom" Width="69">Quê quán:</Label>
19      <ComboBox Height="26" Margin="84,0,27,126" Name="comboBox1"
VerticalAlignment="Bottom" SelectedIndex="0">
20          <ComboBoxItem>Hà nội</ComboBoxItem>
21          <ComboBoxItem>TP. Hồ Chí Minh</ComboBoxItem>
22          <ComboBoxItem>Hải Phòng</ComboBoxItem>
23          <ComboBoxItem>Đà Nẵng</ComboBoxItem>
24      </ComboBox>
25  </Grid>
26 </Window>
```

# Ví dụ

## File Code-Behind

```
01  using System;
02  using System.Text;
03  using System.Windows;
04
05  namespace Controls
06  {
07      /// <summary>
08      /// Interaction logic for MainWindow.xaml
09      /// </summary>
10      public partial class MainWindow : Window
11      {
12          public MainWindow()
13          {
14              InitializeComponent();
15          }
16      }
```

# Ví dụ

## File Code-Behind

```
17     private void button1_Click(object sender, RoutedEventArgs e)
18     {
19         String strMessage, strHoTen, strTitle, strNgoaiNgu = "";
20         strHoTen = textBox1.Text + " " + textBox2.Text;
21
22         if (radioButton1.IsChecked == true)
23             strTitle = "Mr.";
24         else
25             strTitle = "Miss/Mrs.";
26
27         strMessage = "Xin chào: " + strTitle + " " + strHoTen;
28
29         if (checkBox1.IsChecked == true)
30         {
31             strNgoaiNgu = "Tiếng Anh";
32         }
33
```

# Ví dụ

## File Code-Behind

```
34         if (checkBox2.IsChecked == true)
35         {
36             strNgoaiNgu = (strNgoaiNgu.Length == 0) ? "Tiếng Trung" :
37             (strNgoaiNgu + " và Tiếng Trung");
38
39             strMessage += "\nNgoại ngữ: " + strNgoaiNgu;
40
41             if (comboBox1.SelectedIndex >= 0) // Nếu đã có một mục trong
danh sách được chọn
42             {
43                 strMessage += "\nQuê Quán: " + comboBox1.Text;
44             }
45             MessageBox.Show(strMessage);
46         }
47
```

# Ví dụ

## File Code-Behind

```
48     private void button2_Click(object sender, RoutedEventArgs e)
49     {
50         textBox1.Text = "";
51         textBox2.Text = "";
52         radioButton1.IsChecked = true;
53         radioButton2.IsChecked = false;
54         checkBox1.IsChecked = false;
55         checkBox2.IsChecked = false;
56         comboBox1.SelectedIndex = 0;
57     }
58 }
59 }
```

# Ví dụ

**Ví dụ Control - WPF**

Họ đệm:

Tên:

Giới tính:  Nam  Nữ

Ngoại ngữ:  Tiếng Anh  Tiếng Trung

Quê quán:

Xem thông tin

Nhập lại

# Menu

Menu là điều khiển gồm nhiều phần tử được tổ chức dưới dạng phân cấp. Thanh thực đơn thường nằm trên đỉnh cửa sổ (dưới thanh tiêu đề).

Các Menu Item xuất hiện trên thanh thực đơn còn được gọi là Menu Item mức đỉnh. Mỗi Menu Item mức đỉnh có thể chứa nhiều Menu Item cấp dưới (Sub Menu) hoặc được gắn trực tiếp với các bộ quản lý sự kiện (Event handler) như sự kiện Click hay các lệnh của hệ thống được xây dựng sẵn (như Copy, Cut, Paste,...). Tương tự như vậy, mỗi Menu Item cấp dưới lại có thể chứa nhiều Menu Item cấp dưới của chính nó.

# Menu

Khi một Menu Item chứa các Menu Item cấp dưới thì thường được gọi là Popup Menu, các Menu Item cấp dưới sẽ xuất hiện khi người dùng nhấn chuột lên Popup Menu.

Nếu Menu Item được gắn trực tiếp với bộ quản lý sự kiện hay một lệnh của hệ thống thì được gọi là Command Menu, nó sẽ thực thi một câu lệnh mong muốn khi người dùng nhấn chuột hoặc nhấn phím tắt.

# Menu

Thanh Menu được bắt đầu bằng thẻ <Menu> và kết thúc bằng thẻ đóng </Menu>

Các Popup Menu được tạo bởi thẻ <MenuItem> và kết thúc bằng thẻ đóng </MenuItem>. Giữa cặp thẻ này là các thẻ <MenuItem> khác để tạo nên các Menu Item cấp dưới của nó.

Các Command Menu thì được tạo bởi thẻ <MenuItem/>, không có thẻ đóng.

# Menu

Một số thuộc tính cơ bản của Menu Item:

- ▶ Header: Tiêu đề Menu Item. Dấu gạch dưới đặt trước ký tự sẽ được sử dụng làm phím tắt khi kết hợp với phím Alt để gọi Menu Item bằng bàn phím.
- ▶ Name: Tên của Menu Item, cần thiết để có thể can thiệp vào Menu Item.
- ▶ ToolTip: Lời chú thích cho Menu Item khi di chuột qua.

# Menu

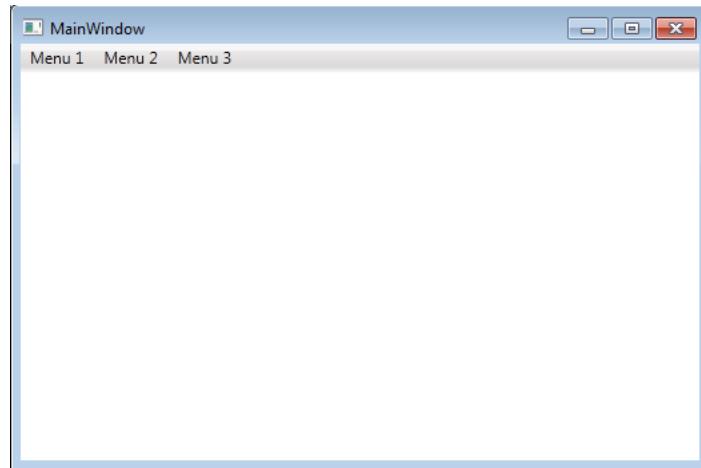
Đối với các Command Menu, có hai cơ chế thực thi lệnh khi chọn Menu.

- ▶ Nếu muốn gắn Command Menu với các lệnh có sẵn của hệ thống như: Copy, Cut, Paste,... sử dụng thuộc tính Command của Menu Item. Ví dụ, Command = "ApplicationCommands.Copy" làm cho Menu Item này sẽ thực hiện công việc copy dòng văn bản đang được chọn trong cửa sổ vào bộ nhớ đệm.
- ▶ Nếu muốn gắn Command Menu với các hàm xử lý sự kiện tự định nghĩa thì sử dụng thuộc tính Click của Menu Item.

# Menu

```
<Menu Height="22" Name="Menu" VerticalAlignment="Top">
    <MenuItem Header="Menu _1" Name="Menu1">
        <MenuItem Header="_Copy" Command="ApplicationCommands.Copy"
                  ToolTip="Copy"/>
        <MenuItem Header="_Cut" Command="ApplicationCommands.Cut"
                  ToolTip="Cut"/>
        <MenuItem Header="_Paste" Command="ApplicationCommands.Paste"
                  ToolTip="Paste"/>
    </MenuItem>
    <MenuItem Header="Menu _2" Name="Menu2">
        <MenuItem Header="Menu 21">
            <MenuItem Header="Menu 211" Click="MenuItem211_click" />
            <MenuItem Header="Menu 212" Click="MenuItem212_click" />
        </MenuItem>
        <MenuItem Header="Menu 22" Click="MenuItem22_Click" />
    </MenuItem>
</Menu>
```

# Menu



# Menu

Khi làm việc với Menu, đôi khi cần có những chức năng với đặc thù có hai trạng thái On/Off. Do đó Menu của WPF cung cấp loại Menu Item với hai trạng thái Checked và UnChecked.

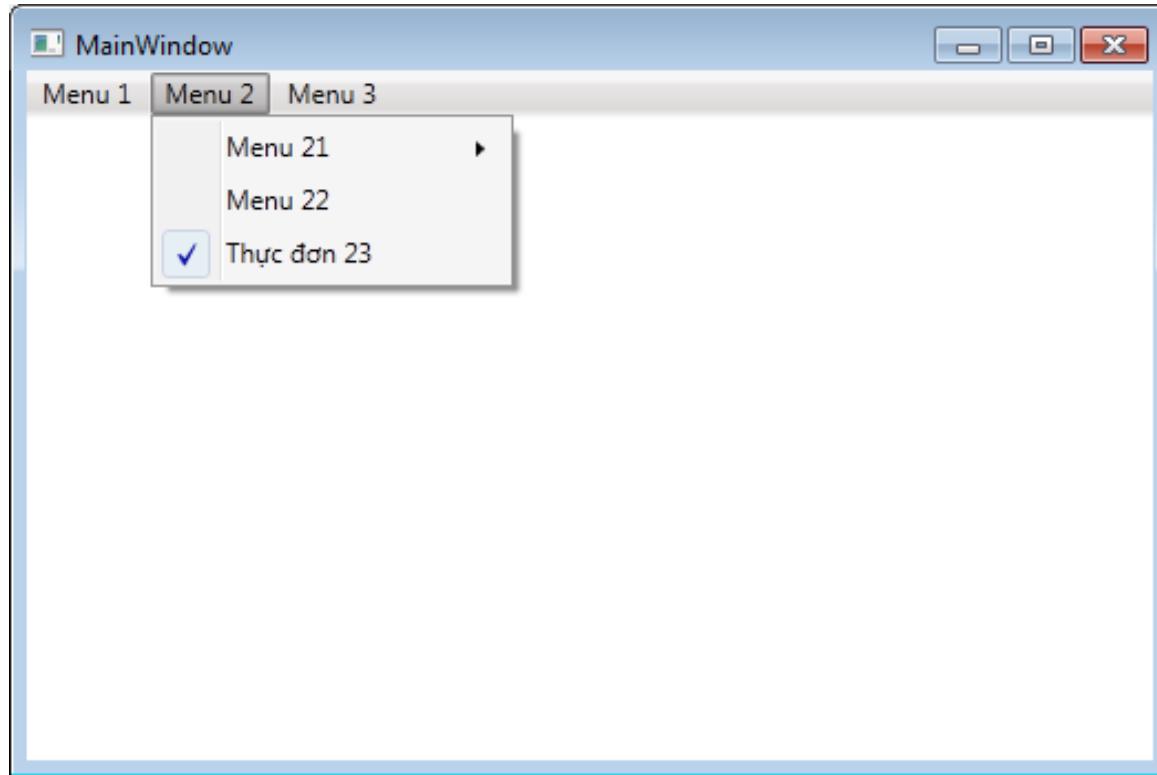
Để tạo ra Menu Item có trạng thái, sử dụng thuộc tính IsCheckable="True" của Menu Item.

Đối với Menu Item có trạng thái, mỗi khi Menu được chọn sẽ phát sinh một trong hai sự kiện Checked và Unchecked tương ứng. Sử dụng các thuộc tính Checked và Unchecked để gắn các hàm xử lý sự kiện cần được thực thi.

# Menu

```
<Menu Height="22" Name="Menu" VerticalAlignment="Top">
    <MenuItem Header="Menu _1" Name="Menu1">
        <MenuItem Header="_Copy" Command="ApplicationCommands.Copy"
            ToolTip="Copy"/>
        <MenuItem Header="_Cut" Command="ApplicationCommands.Cut"
            ToolTip="Cut"/>
        <MenuItem Header="_Paste" Command="ApplicationCommands.Paste"
            ToolTip="Paste"/>
    </MenuItem>
    <MenuItem Header="Menu _2" Name="Menu2">
        <MenuItem Header="Menu 21">
            <MenuItem Header="Menu 211" Click="MenuItem211_Click" />
            <MenuItem Header="Menu 212" Click="MenuItem212_Click" />
        </MenuItem>
        <MenuItem Header="Menu 22" Click="MenuItem22_Click" />
        <!--Thực đơn có trạng thái Checked và UnChecked-->
        <MenuItem Header="Thực đơn 23" IsCheckable="True"
            Checked="Menu23_Checked" Unchecked="Menu23_Unchecked"/>
    </MenuItem>
    <MenuItem Header="Menu _3" Click="MenuItem3_Click" Name="Menu3" />
</Menu>
```

# Menu



# ToolBar

Toolbar là thanh chứa các chức năng dưới dạng các dãy hình ảnh biểu tượng, mỗi biểu tượng gắn với một mục chức năng cụ thể. Thông thường các Toolbar chứa những chức năng thiết yếu mà người dùng hay quan tâm nhất, bởi vì thanh Toolbar có ưu điểm là dễ dàng thao tác. Một cửa sổ có thể có một hoặc nhiều thanh Toolbar.

Mã XAML tạo toolbar được bắt đầu bằng thẻ `<ToolBar>` và kết thúc bằng thẻ đóng `</ToolBar>`. Các nút lệnh (Button) của thanh công cụ được tạo bởi thẻ `<Button>` và kết thúc bằng thẻ đóng `</Button>`.

# ToolBar

Tương tự như Menu, Toolbar cũng có hai cơ chế thực thi lệnh khi chọn nút lệnh:

- ▶ Nếu muốn gắn nút lệnh với các lệnh có sẵn của hệ thống như: Copy, Cut, Paste, thì ta sử dụng thuộc tính Command của Button. Ví dụ, Command = "ApplicationCommands.Copy" làm cho nút lệnh này sẽ thực hiện công việc copy dòng văn bản đang được chọn trong cửa sổ vào bộ nhớ đệm.
- ▶ Nếu muốn gắn nút lệnh với các hàm xử lý sự kiện tự định nghĩa thì sử dụng thuộc tính Click của Button.

# ToolBar

Giữa cặp thẻ `<Button>` và `</Button>` là thẻ `<Image Source="Resources/Copy.png" Width="16" Height="16"/>` để định nghĩa hình ảnh biểu tượng của nút bấm.

Thẻ `<Separator/>` dùng để tạo ra vạch phân cách giữa cách nút bấm.

# ToolBar

Thêm hình ảnh, biểu tượng... vào tài nguyên của ứng dụng

1. Trên menu Visual Studio, chọn Project → Properties sẽ hiện ra bảng cài đặt các thông số cài đặt cho ứng dụng.
2. Chọn mục resources.
3. Trong mục Add Resource chọn Add Existing File nếu đã có sẵn File biểu tượng hình ảnh trên máy hoặc chọn New Images hay Add New Icon tùy ý.
4. Chú ý, sau khi thêm được các File hình ảnh biểu tượng vào tài nguyên, để các điều khiển trên cửa sổ như Menu, Toolbar sử dụng được chúng, phải thiết lập thuộc tính 'Build Action: Resource' và 'Copy to Output Directory : Do not copy'.

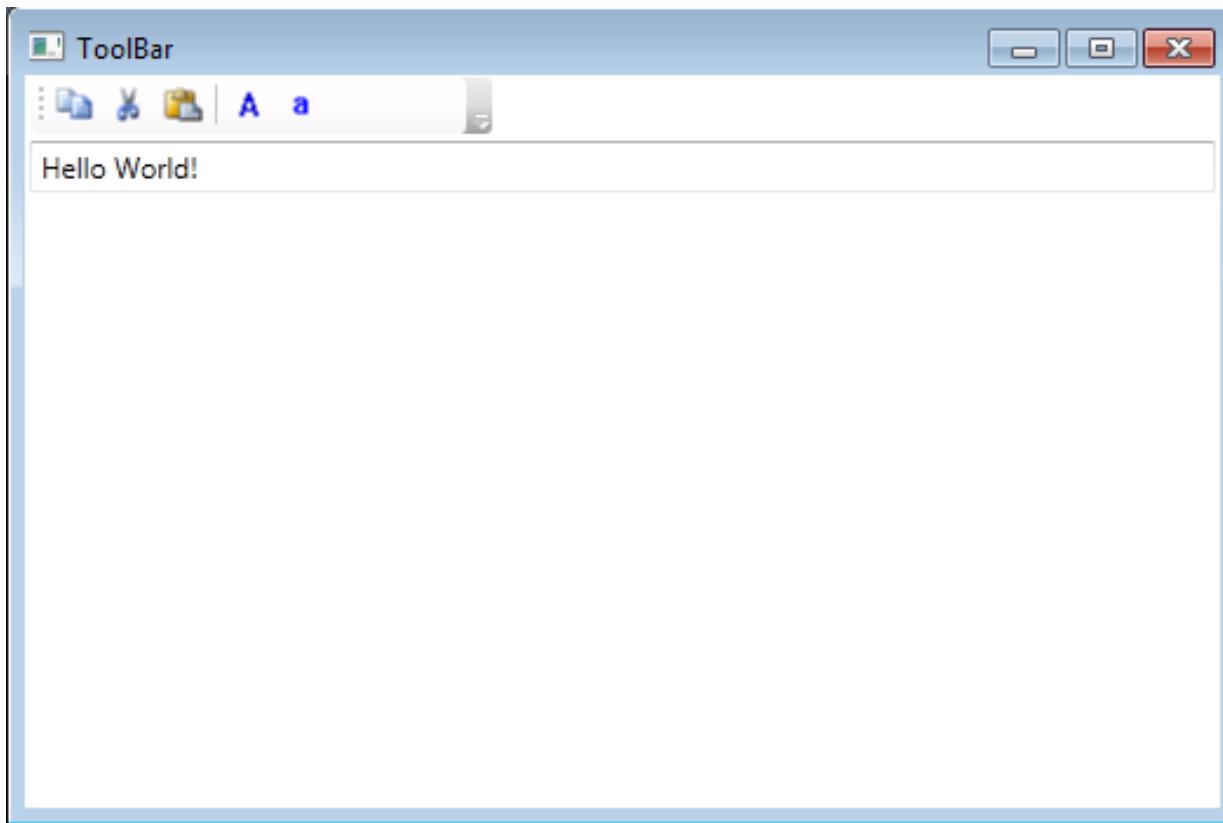
# ToolBar

```
<StackPanel>
    <ToolBar Height="26" Name="toolBar1" Width="200"
        HorizontalAlignment="Left" >
        <Button Height="23" Name="button1" Width="23"
            Command="ApplicationCommands.Copy" ToolTip="Copy">
            <Image Source="Resources/Copy.png" Width="16" Height="16"
                HorizontalAlignment="Left" />
        </Button>
        <Button Height="23" Name="button2" Width="23"
            Command="ApplicationCommands.Cut" ToolTip="Cut">
            <Image Source="Resources/Cut.png" Width="16" Height="16"
                HorizontalAlignment="Left" />
        </Button>
        <Button Height="23" Name="button3" Width="23"
            Command="ApplicationCommands.Paste" ToolTip="Dán văn bản">
            <Image Source="Resources/Paste.png" Width="16" Height="16"
                HorizontalAlignment="Left" />
        </Button>
    <Separator/>
```

# ToolBar

```
<Button Height="23" Name="button4" ToolTip="Increase size of  
font" Width="23" Click="IncreaseFont_Click">  
    <Image Source="Resources/Inc.ico" Width="16" Height="16"  
        HorizontalAlignment="Left" />  
</Button>  
  
<Button Height="23" Name="button5"      ToolTip="Decrease size of  
font" Width="23" Click="DecreaseFont_Click">  
    <Image Source="Resources/Dec.ico" Width="16" Height="16"  
        HorizontalAlignment="Left" />  
</Button>  
</ToolBar>  
<!--Khai báo hộp soạn thảo-->  
<TextBox Name="textBox" TextWrapping="Wrap" Margin="2">  
    Hello World!  
</TextBox>  
</StackPanel>
```

# ToolBar



# ToolBar

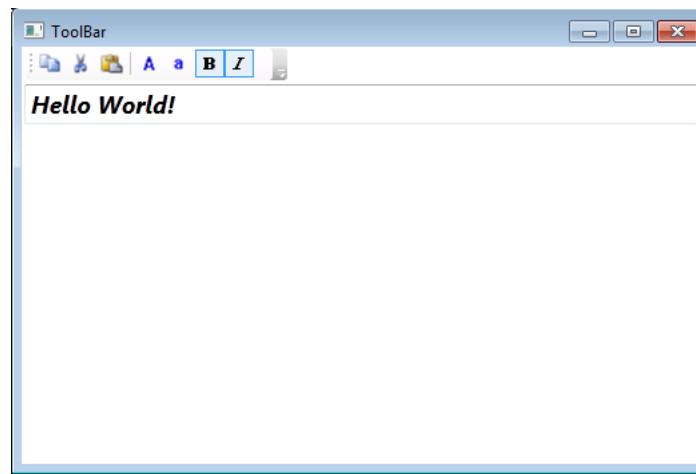
Ngoài các nút bấm thông thường, toolbar còn cho phép tạo ra các nút bấm có trạng thái, khi ở trạng thái được chọn (Checked) thì sẽ có màu nền khác và có đường viền để người dùng có thể nhận biết được trạng thái của nút đó

Để sử dụng nút bấm có trạng thái sử dụng thuộc tính checked

# ToolBar

```
<ToolBar Height="26" Name="toolBar1" Width="200"
    HorizontalAlignment="Left" >
    ...
    <!--Nút bấm với trạng thái Checked và UnChecked-->
    <CheckBox Name="check1" ToolTip="Chữ đậm" Checked="Bold_Checked"
        Unchecked="Bold_Unchecked">
        <Image Source="Resources/Bold.png" Width="16" Height="16"
            HorizontalAlignment="Left" />
    </CheckBox>
    <CheckBox Name="check2" ToolTip="Chữ nghiêng"
        Checked="Italic_Checked" Unchecked="Italic_Unchecked">
        <Image Source="Resources/Italic.png" Width="16" Height="16"
            HorizontalAlignment="Left" />
    </CheckBox>
</ToolBar>
```

# ToolBar



# XỬ LÝ SỰ KIỆN VÀ LỆNH TRONG WPF

Sự kiện là một hành động được phát động bởi người dùng, bởi một thiết bị như đồng hồ đếm (timer) hay bàn phím, hoặc thậm chí là bởi hệ điều hành, tại những thời điểm phần lớn là không theo chu trình nhất định.

Mỗi đơn vị xử lý sự kiện (event handler) đơn giản là một phương thức (hàm) nhận đầu vào từ một thiết bị như chuột hay bàn phím và thực hiện một việc nào đó để phản ứng lại với một sự kiện xảy ra trên thiết bị đó.

# XỬ LÝ SỰ KIỆN VÀ LỆNH TRONG WPF

Có hai bước cần thực hiện để xử lý một sự kiện:

- ▶ Liên kết đơn vị xử lý sự kiện với điều khiển (nút bấm, trường văn bản, menu...), nơi sự kiện tương ứng được phát động.
- ▶ Viết mã lệnh trong đơn vị xử lý sự kiện để lập trình các công việc phản ứng lại với sự kiện.

Có hai cách để liên kết một sự kiện với một đơn vị xử lý sự kiện.

- ▶ Sử dụng một môi trường phát triển tích hợp (IDE) như WPF Designer của Visual Studio (cách trực quan).
- ▶ Viết mã lệnh trực tiếp.

# XỬ LÝ SỰ KIỆN VÀ LỆNH TRONG WPF

WPF mở rộng mô hình lập trình hướng sự kiện chuẩn của .NET, bằng việc đưa ra một loại sự kiện mới gọi là sự kiện có định tuyến (routed event). Loại sự kiện này nâng cao tính linh hoạt trong các tình huống lập trình hướng sự kiện. Việc thiết lập và xử lý một sự kiện có định tuyến có thể thực hiện với cùng cú pháp với một sự kiện “thường” (CLR event).

# Visual Tree (Cây trực quan)

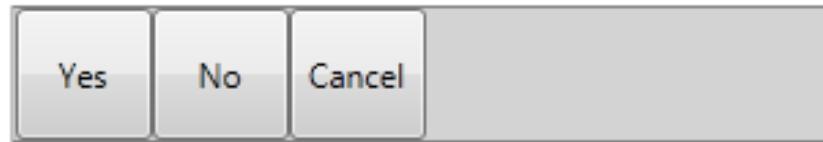
Một giao diện người dùng WPF được xây dựng theo phương thức phân lớp, trong đó một phần tử trực quan không có hoặc có các phần tử con. Cấu trúc phân cấp của các lớp phần tử trực quan trên một giao diện người dùng được gọi là cây trực quan của giao diện đó.

Ví dụ:

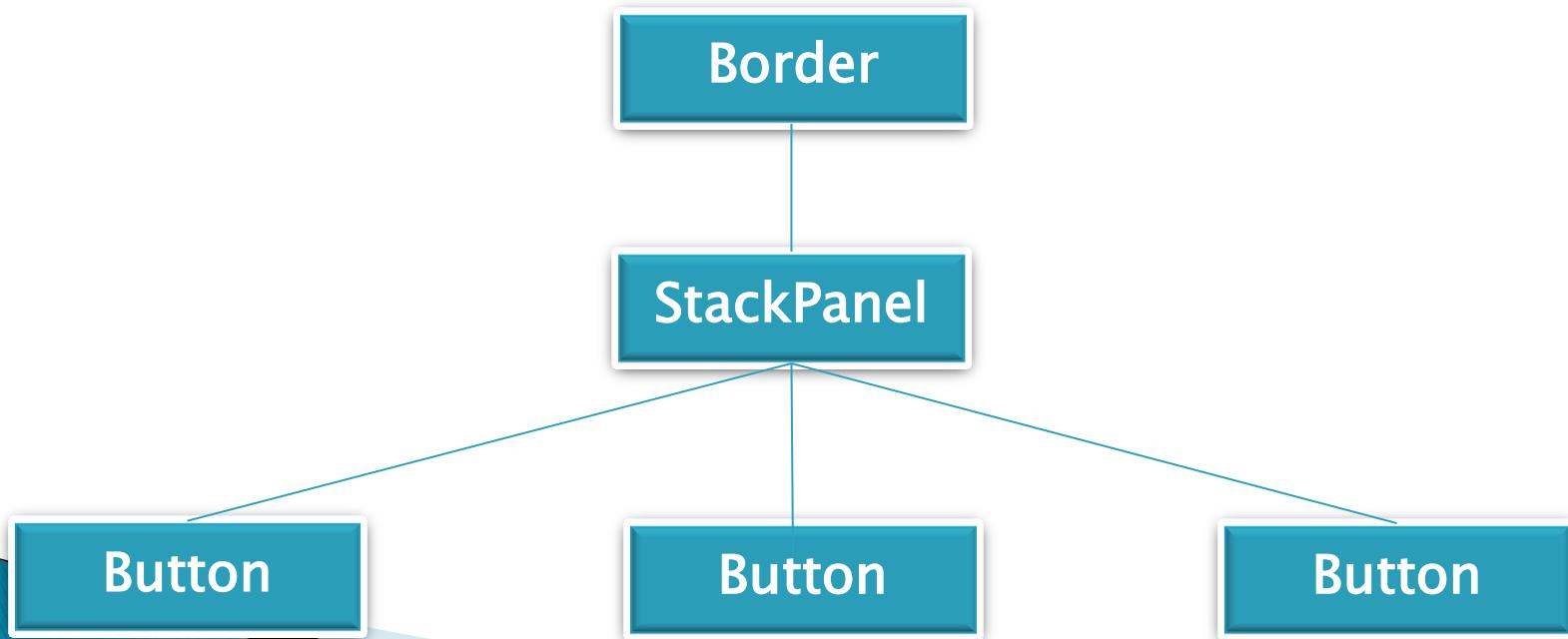
```
<Border Height="50" Width="300" BorderBrush="Gray"  
BorderThickness="1">  
  <StackPanel Background="LightGray" Orientation="Horizontal"  
    Button.Click="CommonClickHandler">  
    <Button Name="YesButton" Width="50" >Yes</Button>  
    <Button Name="NoButton" Width="50" >No</Button>  
    <Button Name="CancelButton" Width="50" >Cancel</Button>  
  </StackPanel>  
</Border>
```

# Cây trực quan

Kết quả khi chạy chương trình:



Cây trực quan tương ứng sẽ là:



# Sự kiện có định tuyến

- ▶ Sự kiện có định tuyến là một loại sự kiện có thể kích hoạt nhiều đơn vị xử lý sự kiện thuộc về nhiều điều khiển khác nhau trên cây trực quan, chứ không chỉ trên đối tượng đã phát động sự kiện.
- ▶ Một ứng dụng WPF thường chứa nhiều phần tử UI. Bất kể được tạo ra bằng mã lệnh hay được khai báo bằng XAML, các thành phần này tồn tại trong mối quan hệ kiểu cây trực quan với nhau - tạo nên các tuyến quan hệ đi từ thành phần này tới thành phần kia. Theo các tuyến quan hệ đó, có ba phương thức định tuyến sự kiện: lan truyền lên (bubble) lan truyền xuống (tunnel) và trực tiếp (direct).

# Sự kiện có định tuyến

Sự kiện Lan truyền lên (bubble) là phương thức thường thấy nhất. Nó có nghĩa là một sự kiện sẽ được truyền đi trên cây trực quan từ thành phần nguồn (nơi sự kiện được phát động) cho tới khi nó được xử lý hoặc nó chạm tới nút gốc. Điều này cho phép xử lý một sự kiện trên một đối tượng nằm ở cấp trên so với thành phần nguồn.

Ví dụ, có thể gắn một hàm xử lý sự kiện Button.Click vào đối tượng Grid có chứa nút bấm thay vì gắn hàm xử lý đó vào bản thân nút bấm.

Sự kiện lan truyền lên có tên gọi thể hiện hành động của sự kiện, ví dụ: MouseDown.

# Sự kiện có định tuyến

Sự kiện lan truyền xuống (tunnel) đi theo hướng ngược lại, bắt đầu từ nút gốc và truyền xuống cây trực quan cho tới khi nó được xử lý hoặc chạm tới thành phần gốc của sự kiện đó. Điều này cho phép các thành phần cấp trên có thể chặn sự kiện và xử lý nó trước khi sự kiện đó chạm tới thành phần nguồn (nơi dự định xảy ra sự kiện). Các sự kiện lan truyền xuống có tên được gắn thêm tiền tố Preview, ví dụ, sự kiện PreviewMouseDown.

Sự kiện trực tiếp (direct) hoạt động giống như sự kiện thông thường trong .NET Framework. Chỉ có một đơn vị xử lý duy nhất sẽ được gắn với sự kiện trực tiếp.

# Sự kiện có định tuyến

Cơ chế thông báo sự kiện kiểu định tuyến có nhiều lợi ích.

Một thành phần UI trực quan không cần móc nối cùng một sự kiện trên tất cả các thành phần con trong nó. Thay vào đó, nó có thể móc nối sự kiện này vào bản thân nó.

Các thành phần ở tất cả các mức trong cây trực quan có thể tự động thực thi mã lệnh để phản ứng lại các sự kiện của các thành phần con của chúng, mà không cần các thành phần con phải thông báo khi sự kiện xảy ra.

## Ví dụ

Form bao gồm một StackPanel chứa 2 Button và 1 TextBlock. StackPanel được phân định bắt sự kiện Click trên hai nút bấm nằm trong nó. Nhiệm vụ của đơn vị xử lý sự kiện Click là cho biết đối tượng nào đã xử lý sự kiện Click, sự kiện Click phát ra từ loại đối tượng nào, tên gọi là gì nào, và loại lan truyền định tuyến đã được thực hiện. Các thông tin trên được đưa vào nội dung của TextBlock và hiển thị lên màn hình sau mỗi sự kiện Click.

# Ví dụ

## File XAML

```
<StackPanel Name="My_StackPanel" Button.Click="HandleClick">
    <!--Khai báo tạo Button 1-->
    <Button Name="Button1">Nút bấm 1</Button>
    <!--Khai báo tạo Button 2-->
    <Button Name="Button2">Nút bấm 2</Button>
    <!--Khai báo tạo TextBlock hiển thị kết quả-->
    <TextBlock Name="Results"/>
</StackPanel>
```

# Ví dụ

## File Code-Behind

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    // Dùng một StringBuilder để lưu trữ thông tin kết quả
    StringBuilder eventstr = new StringBuilder();

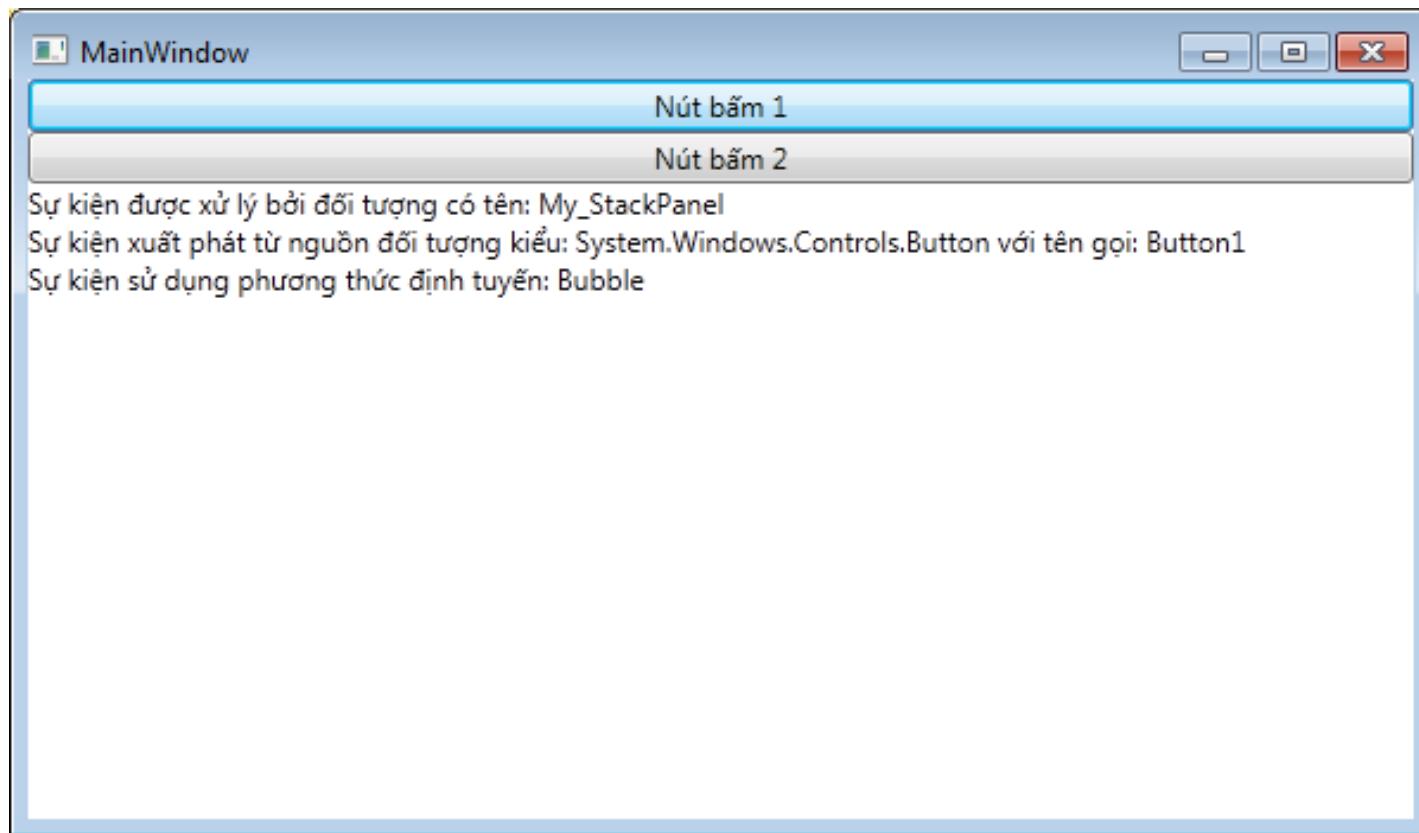
    // Đơn vị xử lý sự kiện Click của Button
    void HandleClick(object sender, RoutedEventArgs args)
    {
        // Lấy thông tin về đối tượng xử lý sự kiện Click, trong
        trường hợp này là My_StackPanel
        FrameworkElement fe = (FrameworkElement)sender;
        eventstr.Append("Sự kiện được xử lý bởi đối tượng có tên: ");
        eventstr.Append(fe.Name);
        eventstr.Append("\n");
    }
}
```

# Ví dụ

## File Code-Behind

```
// Lấy thông tin về nguồn phát ra sự kiện Click:  
FrameworkElement fe2 = (FrameworkElement)args.Source;  
eventstr.Append("Sự kiện xuất phát từ nguồn đối tượng kiểu: ");  
// Loại thành phần UI;  
eventstr.Append(args.Source.GetType().ToString());  
// Định danh;  
eventstr.Append(" với tên gọi: ");  
eventstr.Append(fe2.Name);  
eventstr.Append("\n");  
// Lấy thông tin về phương thức định tuyến  
eventstr.Append("Sự kiện sử dụng phương thức định tuyến: ");  
eventstr.Append(args.RoutedEvent.RoutingStrategy);  
eventstr.Append("\n");  
// Đưa thông tin ra màn hình  
Results.Text = eventstr.ToString();  
}  
}
```

# Ví dụ



# Lệnh (Command) trong WPF

Lệnh tách bạch giữa ngữ nghĩa cũng như nguồn phát hành động với logic thực hiện hành động đó. Điều này cho phép nhiều nguồn khác biệt nhau hoàn toàn có thể phát động cùng một logic lệnh, đồng thời, cho phép tùy biến logic lệnh tùy vào các đối tượng bị tác động khác nhau.

Ví dụ Lệnh Copy, Cut và Paste, được thấy ở rất nhiều ứng dụng. Ngữ nghĩa của các lệnh này là nhất quán với tất cả các ứng dụng và lớp khác nhau (Copy - tạo bản sao từ đối tượng được chọn; Cut - tạo bản sao rồi xoá bỏ đối tượng được chọn (cắt); Paste – Chèn đối tượng được copy/cắt vào vị trí được chọn).

# Lệnh (Command) trong WPF

Tuy nhiên, logic hành động lại tùy thuộc vào đối tượng cụ thể mà ta tác động lên.

Ví dụ, lệnh Cut có thể tác động trên các lớp văn bản, các lớp hình ảnh và trên trình duyệt Web, nhưng logic thực sự thực hiện hành động Cut lại được định nghĩa bởi đối tượng hoặc ứng dụng mà lệnh cắt tác động lên chứ không phải từ nguồn đã phát ra lệnh.

Cụ thể hơn, một đối tượng văn bản có thể cắt đoạn văn bản được chọn vào clipboard, trong khi một đối tượng hình ảnh có thể cắt lấy vùng ảnh được chọn, nhưng nguồn phát lệnh là như nhau - một tổ hợp phím hay một nút bấm trên thanh công cụ.

# Lệnh (Command) trong WPF

Một cách đơn giản để sử dụng lệnh trong WPF là sử dụng một RoutedCommand đã được định sẵn trong các lớp thư viện lệnh; sử dụng một điều khiển có hỗ trợ sẵn xử lý lệnh đó và một điều khiển hỗ trợ sẵn khả năng phát động lệnh.

# Lệnh (Command) trong WPF

Những khái niệm chính trong hệ thống lệnh của WPF

- ▶ Lệnh là hành động được thực hiện
- ▶ Nguồn lệnh là đối tượng phát động lệnh
- ▶ Đích lệnh là đối tượng mà lệnh tác động lên
- ▶ Liên kết lệnh là đối tượng ánh xạ logic thực hiện lệnh với lệnh

# Lệnh có định tuyến

Sự khác biệt giữa lệnh có định tuyến và sự kiện có định tuyến là cách mà lệnh được dẫn đường từ nơi phát động lệnh (nguồn lệnh) tới nơi xử lý lệnh (đích lệnh). Trong mô hình lệnh có định tuyến, sự kiện có định tuyến được sử dụng dưới dạng các thông báo giữa nguồn lệnh vào đích lệnh (thông qua liên kết lệnh).

# Lệnh có định tuyến

Trong một thời điểm nhất định, chỉ có một đơn vị xử lý lệnh (gắn với đích lệnh) sẽ được thực sự kích hoạt (Đơn vị xử lý lệnh hoạt động).

Đơn vị xử lý lệnh hoạt động được xác định bằng việc kết hợp giữa vị trí của nguồn lệnh và đích lệnh trên cây, và đâu là thành phần UI đang nhận được focus.

Khi lệnh được phát đi, sự kiện có định tuyến sẽ được sử dụng để gọi đến đơn vị xử lý lệnh hoạt động, để hỏi xem **lệnh này có được cho phép không** ( thông qua phương thức CanExecute), cũng như **thực hiện logic hành động** ( thông qua phát động phương thức Executed).

# Lệnh có định tuyến

Thông thường, nơi phát lệnh sẽ tìm liên kết lệnh giữa vị trí của nó trên cây trực quan và nút gốc của cây trực quan. Nếu nó tìm thấy một liên kết lệnh như thế, đơn vị xử lý lệnh tương ứng sẽ xác định lệnh này có được cho phép thực hiện không. Nếu như lệnh được gắn với một điều khiển trên thanh công cụ hay menu, thì một vài bước logic thêm sẽ được thực hiện để tìm dọc theo đường đi trên cây trực quan từ nút gốc tới phần tử đang nhận được focus để tìm kiếm một liên kết lệnh.

Một điểm quan trọng cần hiểu về việc định tuyến trong lệnh có định tuyến của WPF là một khi một đơn vị xử lý lệnh đã được kích hoạt, sẽ không có đơn vị xử lý nào khác được gọi.

# Ví dụ

```
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Delete" Header="Delete" />
    </Menu>

    <Label>Khách hàng:</Label>
    <ListBox Name="lsbCustomers">
        <ListBox.CommandBindings>
            <CommandBinding
                Command="ApplicationCommands.Delete"
                CanExecute="DeleteCustomer_CanExecute"
                Executed="DeleteCustomer_Executed" />
        </ListBox.CommandBindings>

        <ListBoxItem>Nguyễn Văn A</ListBoxItem>
        <ListBoxItem>Nguyễn Văn B</ListBoxItem>
        <ListBoxItem>Nguyễn Văn C</ListBoxItem>
    </ListBox>
</StackPanel>
```

## Ví dụ

Đoạn mã ban đầu xác định giá trị cho thuộc tính Command cho mục Delete trên menu. Đồng thời gắn một liên kết lệnh vào ListBox danh sách khách hàng. Trong trường hợp này, menu Delete là nguồn lệnh, và ListBox đóng vai trò là đích lệnh. CommandBinding xác định hàm thực hiện đối với hai thuộc tính CanExecute và Executed. CanExecute xác định khi nào lệnh Delete có thể được thực hiện, trong khi Executed xác định thực hiện logic lệnh trên đích lệnh như thế nào.

# Ví dụ

Thêm vào một ListBox:

```
<Label>Sản phẩm:</Label>
<ListBox x:Name="lsbProducts" >
    <ListBox.CommandBindings>
        <CommandBinding
            Command="ApplicationCommands.Delete"
            CanExecute="DeleteProduct_CanExecute"
            Executed="DeleteProduct_Executed" />
    </ListBox.CommandBindings>
    <ListBoxItem>Sản phẩm 1</ListBoxItem>
    <ListBoxItem>Sản phẩm 2</ListBoxItem>
    <ListBoxItem>Sản phẩm 3</ListBoxItem>
</ListBox>
```

# Ví dụ

Mã lệnh xử lí lệnh Delete cho mỗi điều khiển:

```
private void DeleteCustomer_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = lsbCustomers.SelectedItem != null;
}

private void DeleteCustomer_Executed(object sender, ExecutedRoutedEventArgs e)
{
    lsbCustomers.Items.Remove(lsbCustomers.SelectedItem);
}

private void DeleteProduct_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = lsbProducts.SelectedItem != null;
}

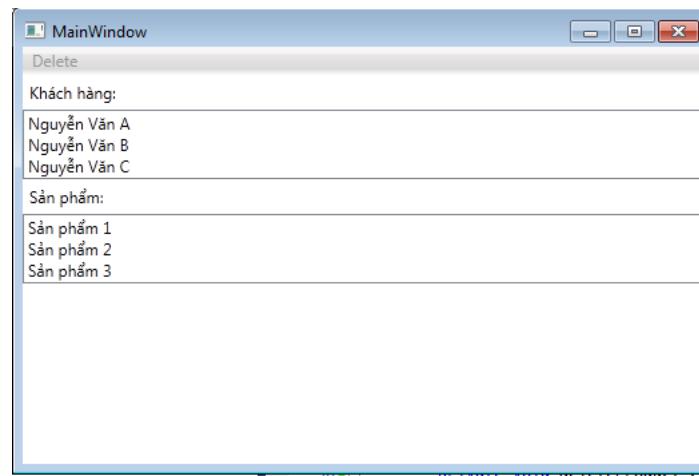
private void DeleteProduct_Executed(object sender, ExecutedRoutedEventArgs e)
{
    lsbProducts.Items.Remove(lsbProducts.SelectedItem);
}
```

## Ví dụ

Khi sử dụng phương thức lệnh trong WPF, không cần phải thay đổi mã của nguồn lệnh (menu Delete) khi thêm ListBox thứ hai. Đồng thời cũng không cần phải cân nhắc Listbox nào nhận được focus. Lớp CommandManager (lớp phối hợp hoạt động của các lệnh trong WPF) sẽ tương tác với FocusManager để xác định điều khiển nào hiện đang nhận focus.

Lệnh trong WPF đơn thuần chỉ thông báo cho các phần tử UI biết rằng có một lệnh có ngữ nghĩa như thế đang được phát động - Bản thân phần tử UI phải triển khai/hiện thực hóa logic hành động phản ứng lại. Ở Ví dụ trên, lệnh Delete được phát động từ cùng nguồn (Delete menu), nhưng việc cài đặt được thực hiện riêng cho 2 đối tượng hoàn toàn khác nhau.

# Ví dụ



# Lệnh tự tạo

Tự tạo các lệnh của riêng trong nhiều trường hợp là cần thiết. Việc này cũng không quá phức tạp trong WPF.

Để làm được điều này, lớp lệnh tự tạo phải hiện thực hoá giao diện ICommand. Tuy nhiên, có thể dùng lớp RoutedUICommand là lớp có sẵn trong framework đã hiện thực hoá tốt giao diện ICommand.

# Lệnh tự tạo

Ví dụ:

Tạo một lệnh mới InsertCustomer:

```
public static class MyCommands
{
    public readonly static RoutedUICommand InsertCustomer;

    static MyCommands()
    {
        InsertCustomer = new RoutedUICommand(
            "Insert Customer", "InsertCustomer",
            typeof(MyCommands));
    }
}
```

# Lệnh tự tạo

Khai báo thêm một CommandBinding cho lệnh mới này cho Window chính:

```
<Window.CommandBindings>
    <CommandBinding
        Command="local:MyCommands.InsertCustomer"
        CanExecute="InsertCustomer_CanExecute"
        Executed="InsertCustomer_Executed" />
</Window.CommandBindings>
```

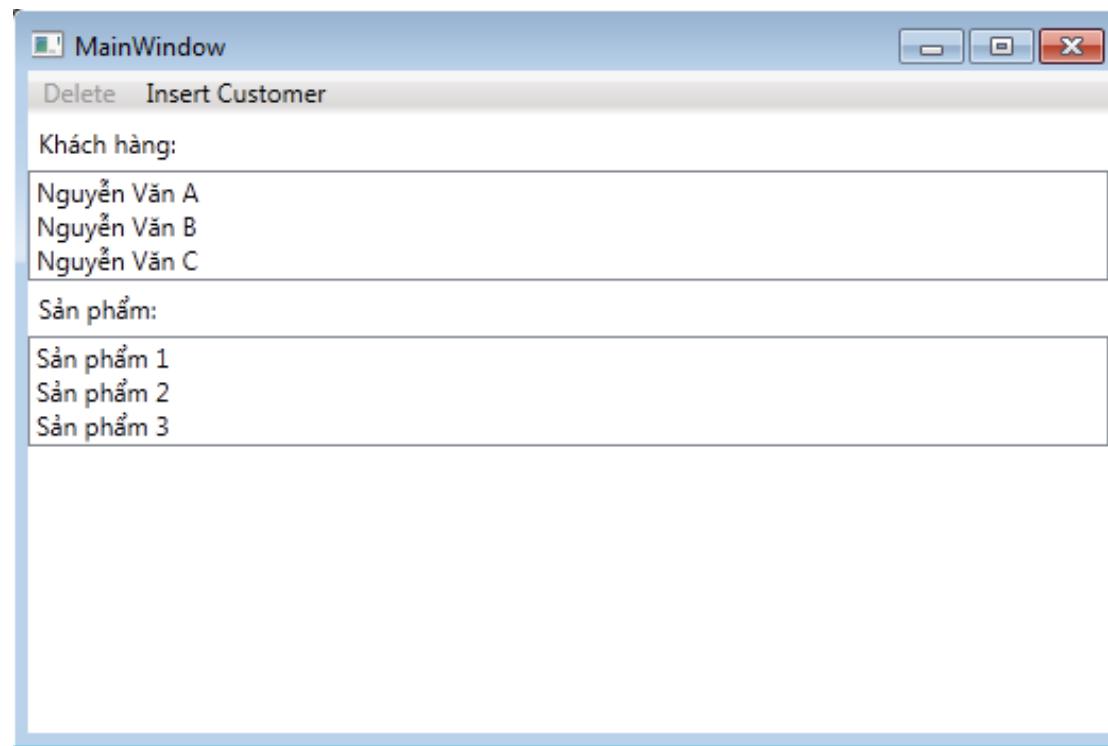
Đồng thời thêm dòng sau vào phần khai báo Window trong file xaml:

```
xmlns:local="clr-namespace:Command2"
```

Cuối cùng tạo một menu mới trên form:

```
<MenuItem Command="local:MyCommands.InsertCustomer"
Header="Insert Customer" />
```

# Lệnh tự tạo



# Kiểu hiển thị (Style)

Style là một kĩ thuật để định dạng cách hiển thị, xử lý cho một/nhiều control bằng cách gán giá trị cho các property, event. Sử dụng Style giúp cho việc thiết kế giao diện trở nên dễ dàng và linh hoạt hơn nhiều so với Windows Forms.

Thành phần Style cho phép người lập trình lưu trữ một danh sách các giá trị thuộc tính vào một nơi thuận tiện. Nó tương tự như cách làm việc của CSS trong các ứng dụng Web.

Thông thường, các Style được lưu trữ trong phần Resource hoặc một thư mục Resource riêng của project.

Các thuộc tính quan trọng nhất của thành phần Style bao gồm: BasedOn, TargetType, Setters và Triggers.

# Kiểu hiển thị (Style)

Được xem như một loại tài nguyên, Style có thể được định nghĩa ở bất kỳ phân cấp nào trong cây trực quan, ví dụ cho một StackPanel, Window hoặc thậm chí ở mức Application.

Việc đặt khai báo Style nên lưu chúng trong một file xaml tài nguyên riêng. Khi định nghĩa một tài nguyên trong XAML, nên định nghĩa một giá trị khoá duy nhất cho tài nguyên đó thông qua thuộc tính x:Key. Kể từ sau đó, có thể tham chiếu tới tài nguyên này bằng việc sử dụng giá trị này.

# BasedOn

Thuộc tính này giống như tính chất kế thừa, trong đó, một Style kế thừa thuộc tính chung của một Style khác. Mỗi kiểu hiện thị chỉ hỗ trợ một giá trị BaseOn.

```
<!--Khai báo Style được kế thừa-->
<Style x:Key="Style1">
...
</Style>
<!--Khai báo Style kế thừa-->
<Style x:Key="Style2" BasedOn="{StaticResource Style1}">
...
</Style>
```

# TargetType

Thuộc tính TargetType được sử dụng để giới hạn loại điều khiển nào được sử dụng Style đó.

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

# Setters

Setters cho phép thiết lập một sự kiện hay một thuộc tính với một giá trị nào đó. Trong trường hợp thiết lập một sự kiện, chúng liên kết với một sự kiện và kích hoạt hàm xử lý tương ứng. Trong trường hợp thiết lập một thuộc tính, chúng đặt giá trị cho thuộc tính đó.

Setter dùng để gán giá trị cho property và EventSetter để gán giá trị cho các event.

# Setters

```
<Style TargetType="{x:Type Button}">
    <EventSetter Event="Click" Handler="b1SetColor"/>
</Style>

<Style TargetType="{x:Type Button}">
    <Setter Property="BackGround" Value="Yellow"/>
</Style>
```

# Trigger

Trigger là phương pháp giúp đặt giá trị cho các property dựa trên một điều kiện cụ thể. Các Trigger được lưu trong collection Style.Triggers. Một Trigger bao gồm hai phần chính là:

- ▶ Điều kiện kích hoạt Trigger được tạo bằng cách gán hai giá trị Trigger.Property và Trigger.Value.
- ▶ Một collection Setter để thay đổi giá trị của các property khi điều kiện của Trigger được đáp ứng.

# Trigger

Ví dụ sử dụng các Trigger để tự động thay đổi màu chữ của Button khi property Button.Content chỉ ra tên của màu đó:

```
<Style TargetType="Button">
    <Style.Triggers>
        <Trigger Property="Content" Value="Blue" >
            <Setter Property="Foreground" Value="Blue" />
        </Trigger>
        ...
        <Trigger Property="Content" Value="Purple" >
            <Setter Property="Foreground" Value="Purple" />
        </Trigger>
    </Style.Triggers>
</Style>
...
<Button Content="Purple" />
```

# MultiTrigger

Khi cần đặt nhiều điều kiện cho Trigger, cần sử dụng lớp MultiTrigger. Lớp này chứa một collection các đối tượng Condition tương ứng với mỗi điều kiện. Với Condition, có thể đặt điều kiện bằng cách xác định property hoặc sử dụng binding. Tuy nhiên chỉ được phép dùng binding khi làm việc với lớp MultiDataTrigger.

# MultiTrigger

```
<Style TargetType="Button">
    <Style.Triggers>
        <MultiTrigger>
            <MultiTrigger.Conditions>
                <Condition Property="Content" Value="Blue" />
                <Condition Property="FontSize" Value="12" />
            </MultiTrigger.Conditions>
            <Setter Property="Foreground" Value="Blue"/>
        </MultiTrigger>
    </Style.Triggers>
</Style>
```

# DataTrigger và MultiDataTrigger

Đặc điểm chính của DataTrigger là không xét điều kiện dựa trên các property mà dựa trên binding. Như vậy sử dụng DataTrigger tương tự như Trigger chỉ khác ở điểm cần sử dụng thuộc tính Binding thay vì Property.

Ví dụ: sử dụng binding kết hợp với style để highlight các item có giá trị đặc biệt trong một ListBox.

# DataTrigger và MultiDataTrigger

## File XAML

```
<Window.Resources>
    <Style TargetType="ListBoxItem">
        <Style.Triggers>
            <DataTrigger Binding="{Binding Path=Age}" Value="11">
                <Setter Property="Background" Value="Yellow" />
            </DataTrigger>
        </Style.Triggers>
    </Style>
</Window.Resources>

<Grid Width="300" Height="200">
    <ListBox Name="listBox1"/>
</Grid>
```

# DataTrigger và MultiDataTrigger

## File Code-Behind

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        listBox1.ItemsSource = new Person[]{
            new Person() { Age = 11, Name = "Alan" },
            new Person() { Age = 11, Name = "Tessa" },
            new Person() { Age = 12, Name = "Lenny" }
        };
    }

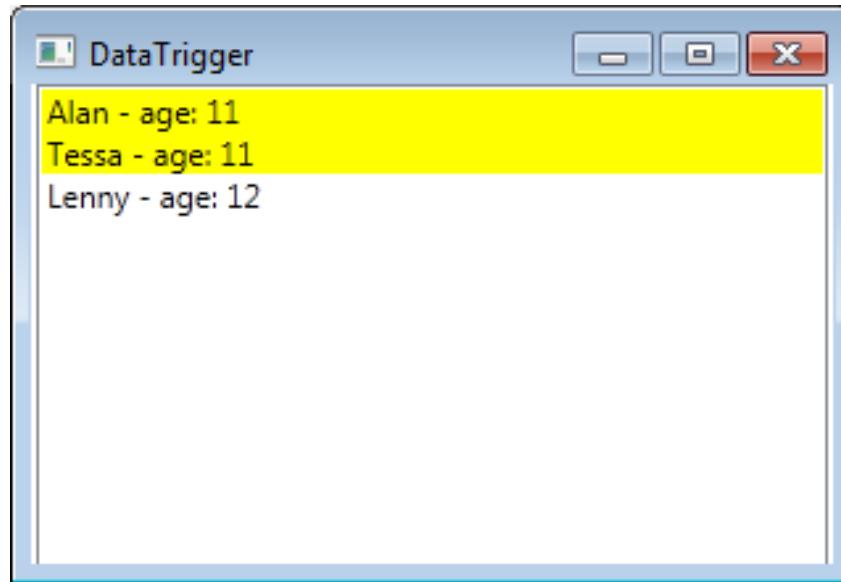
}
```

# DataTrigger và MultiDataTrigger

## File Code-Behind

```
public class Person
{
    public int Age
    {
        get;
        set;
    }
    public string Name
    {
        get;
        set;
    }
    public override string ToString()
    {
        return Name + " - age: " + Age;
    }
}
```

# DataTrigger và MultiDataTrigger



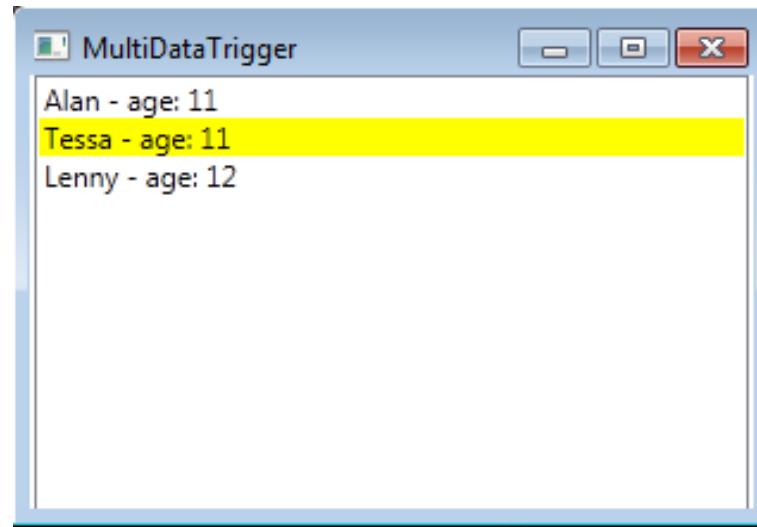
# DataTrigger và MultiDataTrigger

## File XAML

```
<Window.Resources>
    <Style TargetType="ListBoxItem">
        <Style.Triggers>
            <MultiDataTrigger>
                <MultiDataTrigger.Conditions>
                    <Condition Binding="{Binding Path=Age}" Value="11" />
                    <Condition Binding="{Binding Path=Name}" Value="Tessa" />
                </MultiDataTrigger.Conditions>
                <Setter Property="Background" Value="Yellow" />
            </MultiDataTrigger>
        </Style.Triggers>
    </Style>
</Window.Resources>

<Grid Width="300" Height="200">
    <ListBox Name="listBox1" />
</Grid>
```

# DataTrigger và MultiDataTrigger



# EventTrigger

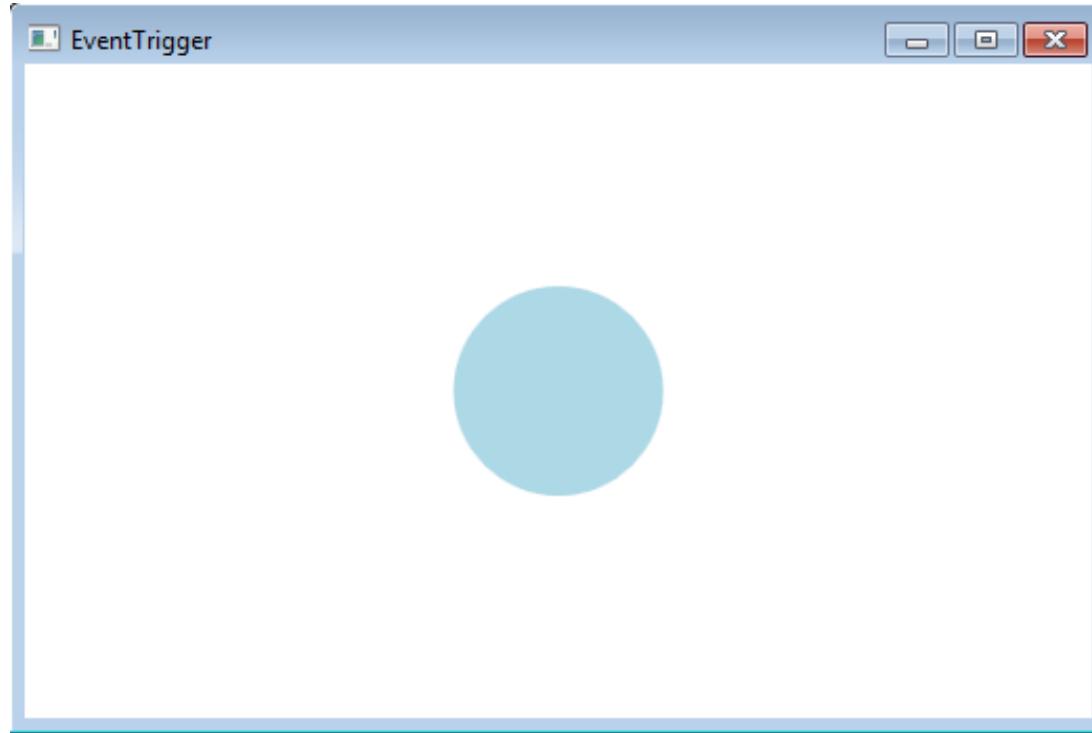
Thay vì sử dụng điều kiện là property và binding như các loại Trigger trên, EventTrigger được kích hoạt khi một event xảy ra trên control. EventTrigger không cho phép thay đổi các property của control mà chỉ cho phép sử dụng các animation để áp dụng cho control.

Ví dụ thực hiện animation đổi màu của các đối tượng Ellipse khi rê chuột vào trong control (Mouse.MouseEnter).

# EventTrigger

```
<Window.Resources>
    <Style TargetType="Ellipse">
        <Style.Triggers>
            <EventTrigger RoutedEvent="Mouse.MouseEnter">
                <EventTrigger.Actions>
                    <BeginStoryboard>
                        <Storyboard>
                            <ColorAnimation
                                Storyboard.TargetProperty="Fill.Color"
                                To="Orange" Duration="0:0:0.5"
                                AutoReverse="True" />
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger.Actions>
            </EventTrigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
<Ellipse Width="100" Height="100" Fill="LightBlue"/>
```

# EventTrigger



# Control Template

ControlTemplate của một điều khiển định nghĩa diện mạo cho điều khiển đó. Mỗi control template được đại diện bằng lớp ControlTemplate. Để thay đổi template của một control, cần gán giá trị cho property Control.Template.

Ví dụ: hình dáng của Button không phải là hình chữ nhật như mặc định mà sẽ có hình tròn, chỉ có thể tác động đến Button bằng chuột nếu như tọa độ của nó nằm bên trong Ellipse này.

```
<Button>
    <Button.Template>
        <ControlTemplate>
            <Ellipse Fill="Blue" Width="100" Height="100"/>
        </ControlTemplate>
    </Button.Template>
</Button>
```

# Control Template

Có thể tách riêng và để ControlTemplate trong phần Resource:

```
<Window.Resources>
    <ControlTemplate x:Key="template1">
        <Ellipse Fill="Blue" Width="100" Height="100"/>
    </ControlTemplate>
</Window.Resources>

<Button Content="Hello" Template="{StaticResource template1}" />
```

# Control Template

Hoặc kết hợp với Style bằng cách sử dụng Setter với property Control.Template:

```
<Window.Resources>
    <ControlTemplate x:Key="template1">
        <Ellipse Fill="Blue" Width="100" Height="100"/>
    </ControlTemplate>
    <Style TargetType="Button">
        <Setter Property="Template" Value="{StaticResource template1}"/>
    </Style>
</Window.Resources>

<Button Content="Hello" />
```

# Template Binding

Đây là kĩ thuật dùng để binding đến các property của đối tượng được áp dụng control template. Chẳng hạn các thành phần con bên trong template có thể binding để lấy các giá trị như kích thước, màu sắc,... của Button để sử dụng và thay đổi theo Button đó.

```
<ControlTemplate x:Key="template1" TargetType="Button">
    <Ellipse Fill="{TemplateBinding Property=Background}"
Width="100" Height="100"/>
</ControlTemplate>
```

# Content Presenter

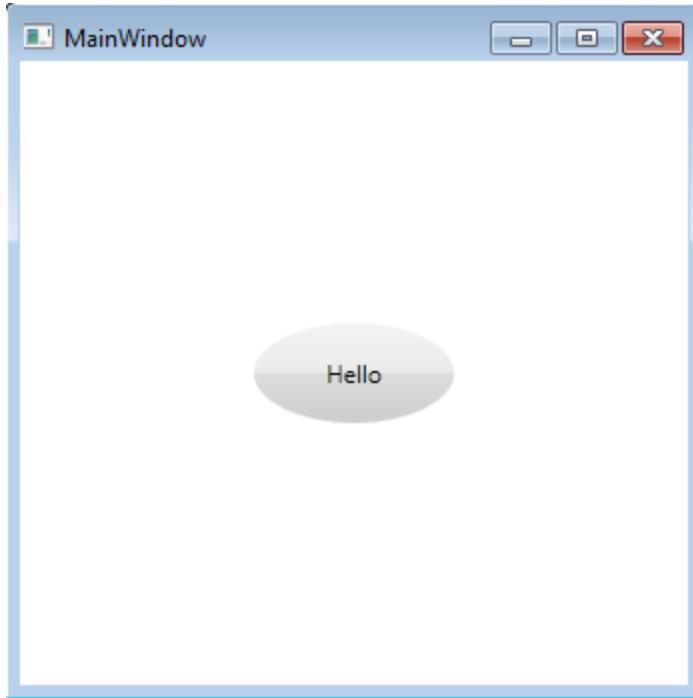
ContentPresenter là đối tượng dùng để hiển thị nội dung của control. Vị trí đặt đối tượng này cũng chính là vị trí mà nội dung của control sử dụng template sẽ xuất hiện.

Ví dụ tạo một button template và hiển thị dòng chữ “Hello” (từ property Button.Content) vào giữa đối tượng Ellipse:

```
<Window.Resources>
    <ControlTemplate x:Key="template1" TargetType="Button">
        <Grid>
            <Ellipse Fill="{TemplateBinding Property=Background}"/>
            <ContentPresenter HorizontalAlignment="Center"
                VerticalAlignment="Center"/>
        </Grid>
    </ControlTemplate>
</Window.Resources>

<Button Width="100" Height="50" Content="Hello"
    Template="{StaticResource template1}" />
```

# Content Presenter



# DataTemplate

DataTemplate được sử dụng để định ra cách thức hiển thị các đối tượng dữ liệu. Mỗi phần tử dữ liệu có thể là một kiểu dữ liệu phức tạp và không phải lúc nào cũng muốn hiển thị nó theo dạng chuỗi đơn giản. Thay vào đó, có thể hiển thị nó theo một cách riêng, trên nhiều control, với định dạng khác nhau,...

# DataTemplate

# File XAML

# DataTemplate

## File XAML

```
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Column="0" Text="{Binding
        Path=Name}" FontStyle="Italic" />
    <TextBlock Grid.Column="1" Text=" Age: "/>
    <TextBlock Grid.Column="2" Text="{Binding
        Path=Age}" />
</Grid>
</Border>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
</Grid>
</Window>
```

# DataTemplate

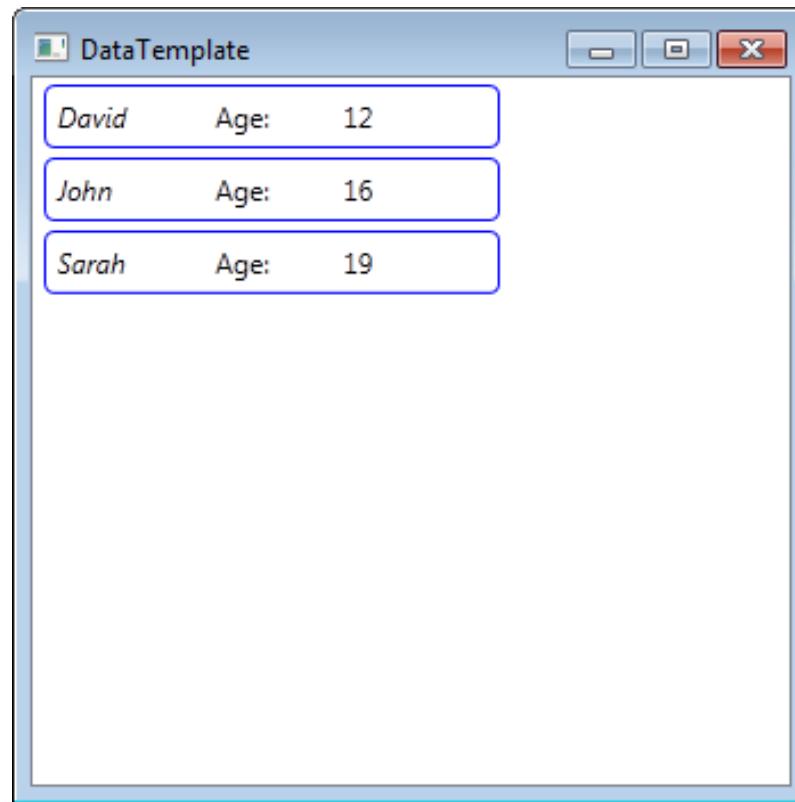
## File Code-Behind

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        List<Person> persons = new List<Person>();
        persons.Add(new Person() { Age = 12, Name = "David" });
        persons.Add(new Person() { Age = 16, Name = "John" });
        persons.Add(new Person() { Age = 19, Name = "Sarah" });

        this.DataContext = persons;
    }
}

public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
}
```

# DataTemplate



# DataTemplate

Có thể định nghĩa DataTemplate trong phần Resouce để có thể sử dụng lại và làm code được rõ ràng.

```
<Window.Resources>
    <DataTemplate x:Key="personTemplate">
        ...
    </DataTemplate>
</Window.Resources>
<Grid>
    <ListBox x:Name="listBox1" ItemsSource="{Binding}"
ItemTemplate="{StaticResource personTemplate}" />
</Grid>
```

# DataTemplate Trigger

Nếu muốn định dạng riêng cho một phần tử trong collection dựa vào một số điều kiện, Có thể tạo một DataTrigger. Các DataTrigger được chứa trong property DataTemplate.Triggers.

# DataTemplate Trigger

## File XAML

```
<Grid>
```

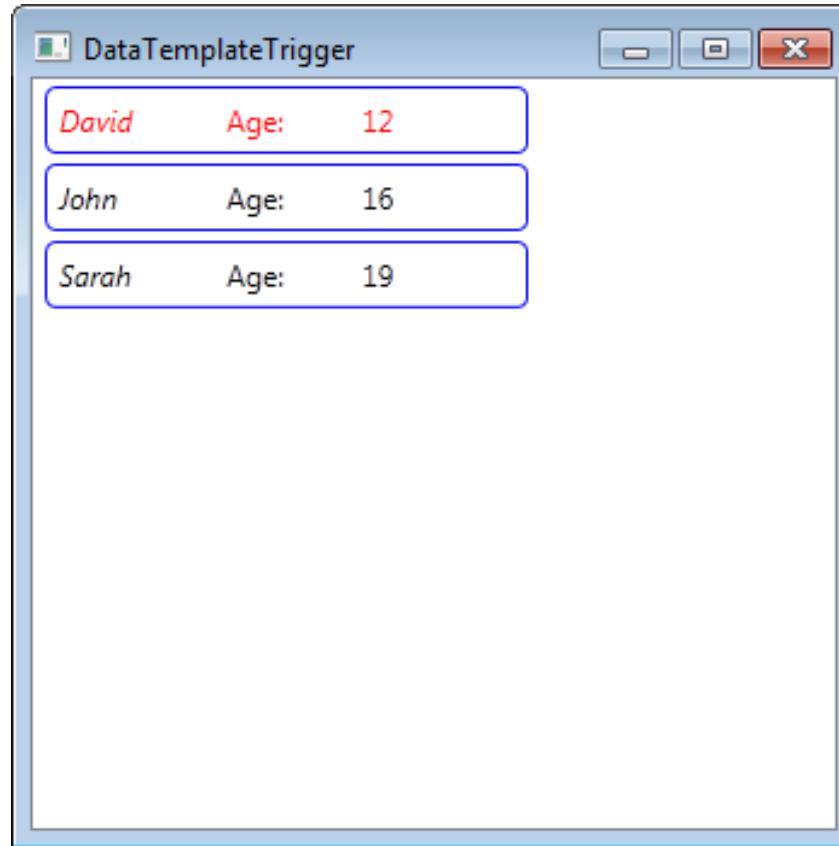
```
  <ListBox x:Name="listBox1" ItemsSource="{Binding}">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <Border BorderThickness="1" BorderBrush="Blue"
          Margin="2" Padding="5" Width="200" CornerRadius="4" >
          <Grid>
            <Grid.ColumnDefinitions>
              <ColumnDefinition />
              <ColumnDefinition />
              <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <TextBlock Grid.Column="0" Text="{Binding
              Path=Name}" FontStyle="Italic" />
            <TextBlock Grid.Column="1" Text=" Age: "/>
            <TextBlock Grid.Column="2" Text="{Binding
              Path=Age}" />
          </Grid>
        </Border>
```

# DataTemplate Trigger

## File XAML

```
<DataTemplate.Triggers>
    <DataTrigger Binding="{Binding Path=Name}"
        Value="David">
        <Setter Property="ListBoxItem.Foreground"
            Value="Red"></Setter>
    </DataTrigger>
</DataTemplate.Triggers>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
</Grid>
```

# DataTemplate Trigger



# Data Template Selector

Giống như DataTrigger nhưng DataTemplateSelector sẽ tạo ra các DataTemplate riêng để áp dụng cho mỗi phần tử của collection, thay vì gán giá trị mới cho các property để thay đổi cách hiển thị.

Sử dụng kỹ thuật này bằng cách tạo một subclass của DataTemplateSelector và override phương thức SelectTemplate(). Phương thức này có kiểu trả về là một DataTemplate.

# Data Template Selector

## File XAML

```
<Window x:Class="DataTemplateSelectorExample.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:DataTemplateSelectorExample"
    Title="DataTemplateSelector" Height="350" Width="350">
    <Window.Resources>
        <DataTemplate x:Key="DefaultTemplate">
            <Border Margin="5" BorderThickness="1" BorderBrush="Gray"
                CornerRadius="4">
                <StackPanel Margin="5">
                    <TextBlock Text="{Binding Path=Name}"/>
                    <TextBlock Text="{Binding Path=Age}"/>
                </StackPanel>
            </Border>
        </DataTemplate>
```

# Data Template Selector

## File XAML

```
<DataTemplate x:Key="HighlightTemplate">
    <Border Margin="5" BorderThickness="1" BorderBrush="Blue"
        Background="LightBlue" CornerRadius="4">
        <StackPanel Margin="5">
            <TextBlock Text="{Binding Path=Name}"/>
            <TextBlock Text="{Binding Path=Age}"/>
        </StackPanel>
    </Border>
</DataTemplate>
</Window.Resources>
<Grid>
    <ListBox Name="lstProducts"
HorizontalContentAlignment="Stretch" ItemsSource="{Binding}">
        <ListBox.ItemTemplateSelector>
            <local:PersonHighlightTemplateSelector/>
        </ListBox.ItemTemplateSelector>
    </ListBox>
</Grid>
</Window>
```

# Data Template Selector

## File Code-Behind

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        List<Person> persons = new List<Person>();
        persons.Add(new Person() { Age = 12, Name = "David" });
        persons.Add(new Person() { Age = 16, Name = "John" });
        persons.Add(new Person() { Age = 19, Name = "Sarah" });

        this.DataContext = persons;
    }
}
```

# Data Template Selector

## File Code-Behind

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
}

public class PersonHighlightTemplateSelector : DataTemplateSelector
{
    public override DataTemplate SelectTemplate(object item,
        DependencyObject container)
    {
        Person person = item as Person;
        FrameworkElement element = container as FrameworkElement;
```

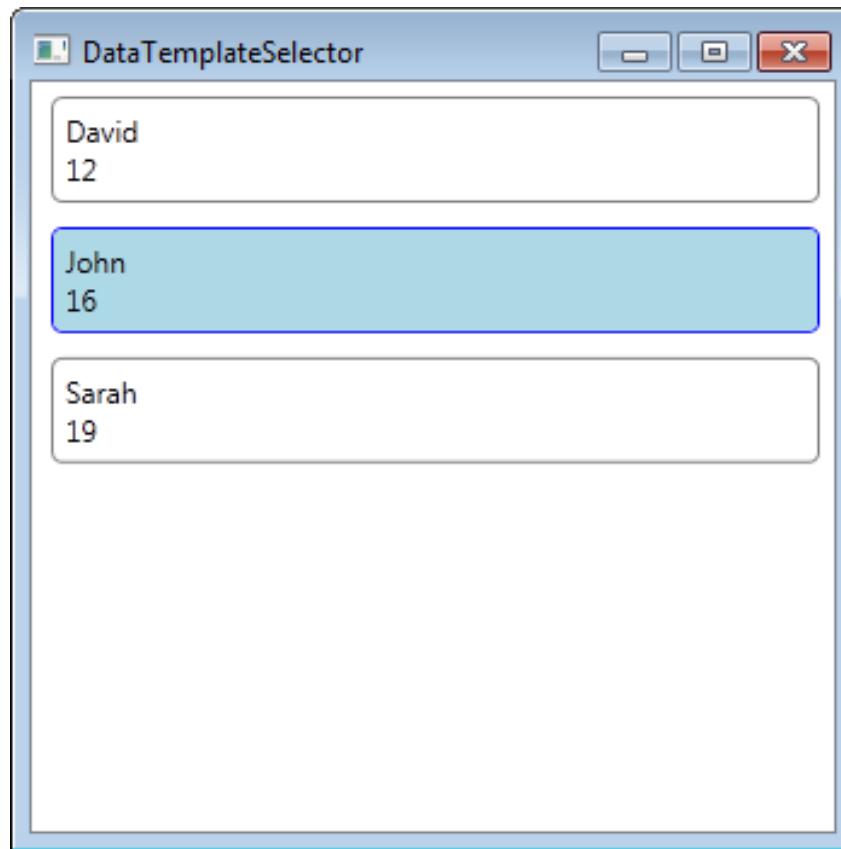
# Data Template Selector

## File Code-Behind

```
if (person != null && element != null)
{
    if (person.Name == "John")
    {
        return element.FindResource("HighlightTemplate") as
DataTemplate;
    }
    else
    {
        return element.FindResource("DefaultTemplate") as
DataTemplate;
    }
}

return null;
}
```

# Data Template Selector



# Đồ họa trong WPF

WPF đã tích hợp sẵn đồ họa vector, đa phương tiện, hình ảnh động (animation) và các đối tượng đồ họa phức hợp. Các đối tượng đồ họa trong WPF không chỉ để hiển thị một các đơn thuần, chúng còn có khả năng phát sinh các sự kiện mà thông thường chỉ có trong các điều khiển thông dụng của Window.

# Các đối tượng đồ họa cơ bản

Các đối tượng đồ họa cơ bản như Line (đoạn thẳng), Ellipse (hình elip), Polygon (đa giác), Polyline (chuỗi đoạn thẳng), Rectangle (chữ nhật) và Path (hình phức hợp) được kế thừa từ đối tượng cơ sở Shape. Các đối tượng kế thừa từ Shape có chung một số thuộc tính như:

- ▶ Stroke: Mô tả màu sắc đường viền của một hình hoặc màu của một đoạn thẳng.
- ▶ StrokeThickness: Độ dày của đường viền.
- ▶ Fill: Cách tô phần bên trong của một hình.
- ▶ Data: Mô tả các tọa độ, các đỉnh của một hình, đơn vị đo là pixel.

# Line

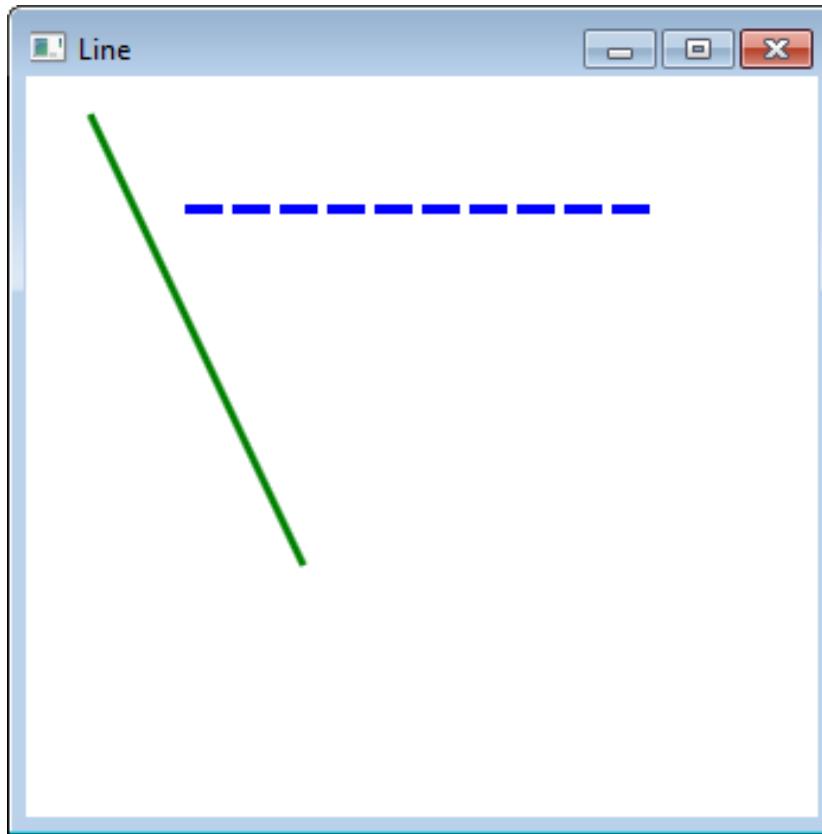
Đoạn thẳng là một đối tượng được định nghĩa dựa trên hai đầu mút là hai điểm.

- ▶ X1, Y1: Tọa độ đỉnh thứ nhất
- ▶ X2, Y2: Tọa độ đỉnh thứ hai
- ▶ StrokeThickness: Độ dày của đoạn thẳng
- ▶ Stroke: Màu của đoạn thẳng
- ▶ StrokeDashArray: Tô vẽ theo nét đứt

# Line

```
<Canvas Height="300" Width="300">
<Line
    X1="10" Y1="10"
    X2="100" Y2="100" Stroke="Green" StrokeThickness="3" />
<Line
    X1="10" Y1="50"
    X2="150" Y2="50" Stroke="Blue"
    StrokeThickness="4" StrokeDashArray="4 1" />
</Canvas>
```

# Line



# Polyline

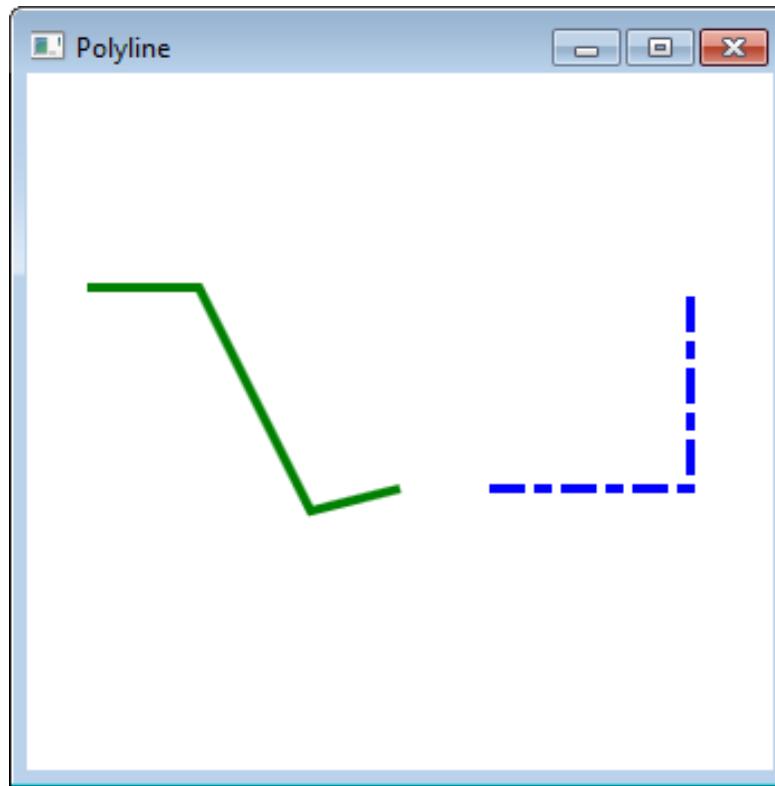
Polyline là đối tượng bao gồm nhiều đoạn thẳng liên tiếp nối với nhau. Một Polyline gồm N đoạn thẳng thì được định nghĩa bởi N+1 điểm.

- ▶ Thuộc tính Points khai báo tập hợp các điểm tạo nên Polyline.
- ▶ Thuộc tính StrokeDashArray quy định nét vẽ đứt

# Polyline

```
<Canvas Height="300" Width="300">
    <Polyline
        Points="10, 10, 60, 10 110,110 150,100" Stroke="Green"
        StrokeThickness="4" Canvas.Left="0" Canvas.Top="80" />
    <Polyline
        Points="10,100 100,100 100,10" Stroke="Blue"
        StrokeThickness="4" StrokeDashArray="4 1 2 1"
        Canvas.Left="180" Canvas.Top="80" />
</Canvas>
```

# Polyline



# Rectangle

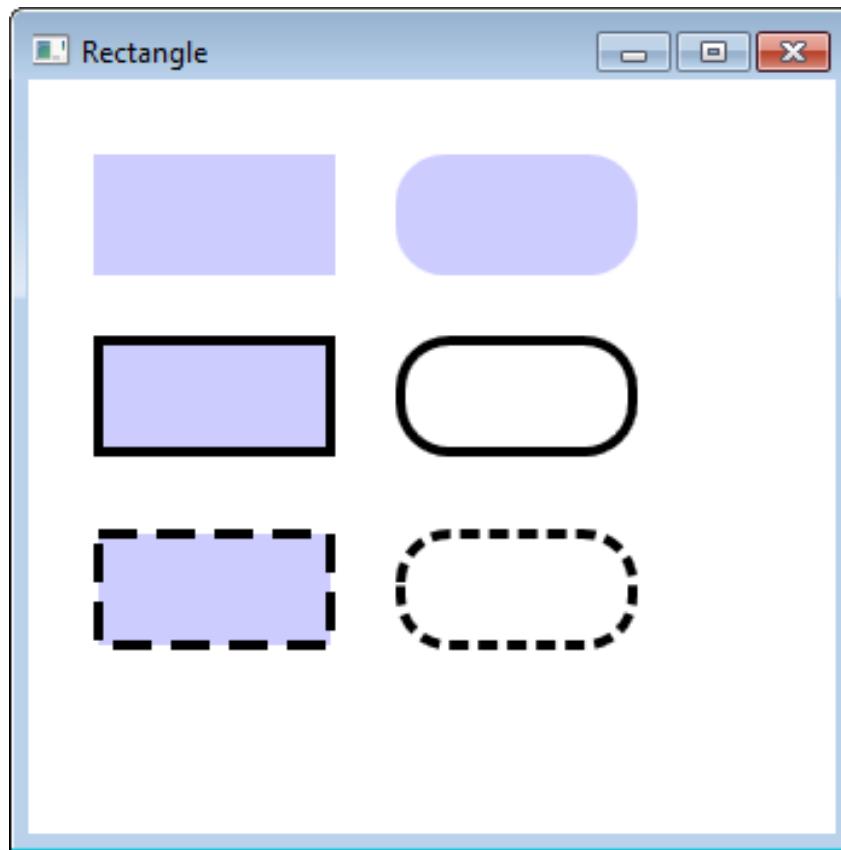
Đối tượng Rectangle được xác định bởi tọa độ của góc trên trái và độ rộng, độ cao của hình chữ nhật cần hiển thị. Ngoài ra, có thể thiết lập các thuộc tính cho đường viền (màu sắc, độ dày, kiểu dáng) và tô phần bên trong của hình.

- ▶ Thuộc tính Fill chỉ định màu tô bên trong hình chữ nhật. Nếu bỏ qua thuộc tính này thì hình chữ nhật sẽ là trong suốt.
- ▶ Các thuộc tính Stroke, StrokeThickness, StrokeDashArray chỉ định kiểu đường viền của hình chữ nhật. Nếu không chỉ định giá trị cho các thuộc tính này thì hình chữ nhật sẽ không có đường viền.
- ▶ Các thuộc tính RadiusX, RadiusY là bán kính của hình ellipse để tạo ra các góc tròn của hình chữ nhật.

# Rectangle

```
<Canvas Height="300" Width="300">
    <Rectangle Width="100" Height="50" Fill="#CCCCFF"
        Canvas.Left="10" Canvas.Top="25" />
    <Rectangle Width="100" Height="50" Fill="#CCCCFF" Stroke="Black"
        StrokeThickness="4" Canvas.Left="10" Canvas.Top="100"/>
    <Rectangle Width="100" Height="50" RadiusX="20" RadiusY="20"
        Fill="#CCCCFF" Canvas.Left="135" Canvas.Top="25"/>
    <Rectangle Width="100" Height="50" RadiusX="20" RadiusY="20"
        Stroke="Black" StrokeThickness="4" Canvas.Left="135"
        Canvas.Top="100" />
    <Rectangle Width="100" Height="50" Fill="#CCCCFF" Stroke="Black"
        StrokeThickness="4" StrokeDashArray="4 2" Canvas.Left="10"
        Canvas.Top="180"/>
    <Rectangle Width="100" Height="50" RadiusX="20" RadiusY="20"
        Stroke="Black" StrokeThickness="4" StrokeDashArray="2 1"
        Canvas.Left="135" Canvas.Top="180" />
</Canvas>
```

# Rectangle



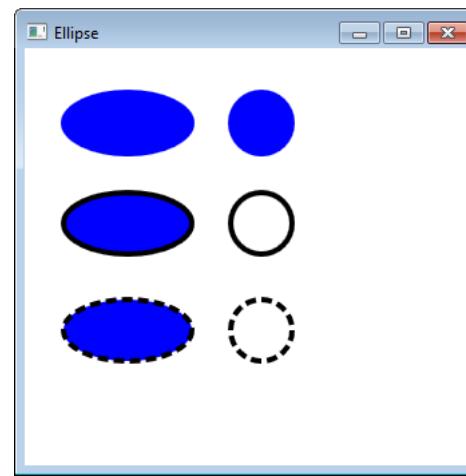
# Ellipse

Hình Ellipse được xác định bởi tọa độ của góc trên trái và độ rộng, độ cao của hình chữ nhật ngoại tiếp của Ellipse cần hiển thị. Hình tròn là hình Ellipse với chiều rộng và chiều cao bằng nhau. Ellipse cũng có các thuộc tính cho đường viền (màu sắc, độ dày, kiểu dáng) và tô phần bên trong của hình tương tự như hình chữ nhật.

# Ellipse

```
<Canvas Height="300" Width="300" Background="AntiqueWhite">
    <Ellipse Width="100" Height="50" Fill="Blue"
        Canvas.Left="10" Canvas.Top="25" />
    <Ellipse Width="100" Height="50" Fill="Blue"
        Stroke="Black" StrokeThickness="4"
        Canvas.Left="10" Canvas.Top="100"/>
    <Ellipse Width="100" Height="50" Fill="Blue"
        Stroke="Black" StrokeThickness="4"
        StrokeDashArray="2 1" Canvas.Left="10"
        Canvas.Top="180"/>
    <Ellipse Width="50" Height="50" Fill="Blue"
        Canvas.Left="135" Canvas.Top="25"/>
    <Ellipse Width="50" Height="50" Stroke="Black"
        StrokeThickness="4" Canvas.Left="135"
        Canvas.Top="100" />
    <Ellipse Width="50" Height="50" Stroke="Black"
        StrokeThickness="4" StrokeDashArray="2 1"
        Canvas.Left="135" Canvas.Top="180" />
</Canvas>
```

# Ellipse



# Polygon

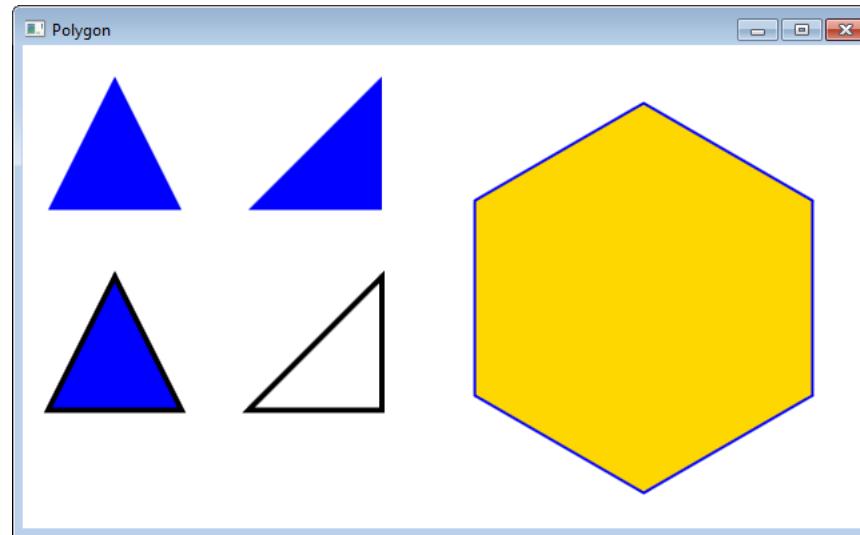
Polygon là đối tượng dùng để trình diễn các hình dạng phức tạp, gồm đoạn thẳng nối tiếp khép kín. Một Polygon N đỉnh được định nghĩa bởi một tập hợp N cặp tọa độ tương ứng với mỗi đỉnh của nó .

Thuộc tính Points của nó để khai báo tọa độ của các đỉnh,

# Polygon

```
<Canvas Height="335" Width="616">
<Polygon Points="10,110 60,10 110,110" Fill="Blue" />
<Polygon Points="10,110 60,10 110,110" Fill="Blue"
          Stroke="Black" StrokeThickness="4" Canvas.Top="150" />
<Polygon Points="10,110 110,110 110,10" Fill="Blue"
          Canvas.Left="150" />
<Polygon Points="10,110 110,110 110,10" Stroke="Black"
          StrokeThickness="4" Canvas.Left="150" Canvas.Top="150" />
<Polygon Stroke="Blue" StrokeThickness="2.0" Fill="Gold"
          Points="176,30 302.44,103 302.44,249 176,322 49.5603,249
        49.5603,103" Canvas.Left="280" Canvas.Top="0" />
</Canvas>
```

# Polygon



# Sử dụng chổi tô Brush

Tất cả những gì hiển thị trên màn hình, chúng hiển thị được là nhờ được tô bởi chổi tô (Brush).

Chổi tô có thể sử dụng để tô nền của một nút bấm (Button), tô các nét chữ (Text) hay tô màu bên trong cho một đối tượng hình học như hình chữ nhật, đa giác,...

Có nhiều kiểu tô khác nhau như tô màu đồng nhất (Solid Color), tô kiểu đổ màu theo tuyến tính (Linear Gradient Color), tô đổ màu dọc theo bán kính hình tròn (Radial Gradient Color) và sử dụng ảnh để tô.

# Solid Color

Có nhiều cách để tô màu đồng nhất cho một đối tượng:

- ▶ Có thể sử dụng trực tiếp thuộc tính tô màu của đối tượng (ví dụ đối tượng hình học sử dụng thuộc tính Fill, đối tượng Button sử dụng thuộc tính Background,...)
- ▶ Hoặc tạo đối tượng tên là SolidColorBrush để định nghĩa màu sắc cần tô và gắn cho đối tượng cần sử dụng chổi tô đồng nhất này.

# Solid Color

Đối tượng SolidColorBrush sử dụng thuộc tính Color để chỉ định màu cần tô.

Giá trị gắn cho Color có thể là:

- ▶ Tên của màu đã được định nghĩa sẵn (là các thuộc tính tĩnh của đối tượng Brush) như Red, MediumBlue,..
- ▶ Hoặc sử dụng công thức biểu diễn màu theo dạng màu 32 bit "#RRGGBB" hoặc "#AARRGGBB", AA là độ Alpha dùng để chỉ độ trong suốt của màu.

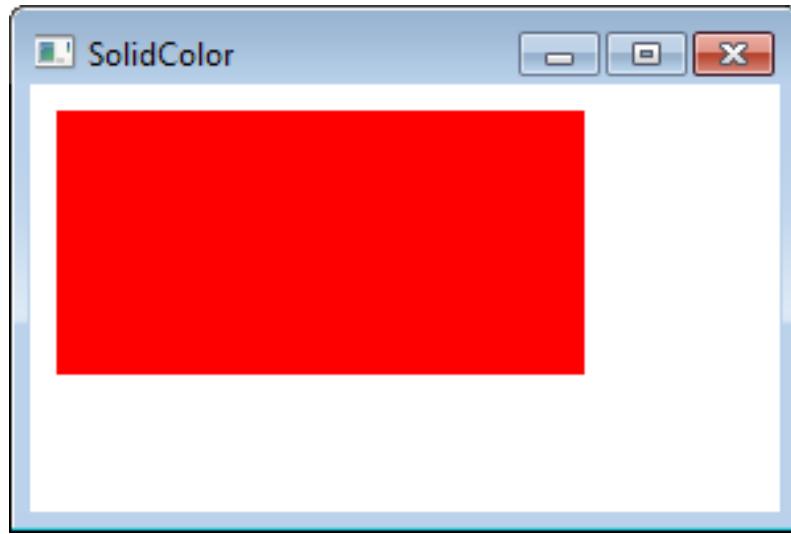
# Solid Color

```
<!--Sử dụng mã định nghĩa sẵn -->
<Rectangle Width="200" Height="100">
    <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
    </Rectangle.Fill>
</Rectangle>

<!--Sử dụng mã màu dạng #RRGGBB -->
<Rectangle Width="200" Height="100">
    <Rectangle.Fill>
        <SolidColorBrush Color="#FF0000" />
    </Rectangle.Fill>
</Rectangle>

<!-- Sử dụng trực tiếp thuộc tính Fill -->
<Rectangle Width="200" Height="100" Fill="#FF0000"/>
```

# Solid Color



# Linear Gradient Color

Đối tượng LinearGradientBrush dùng để tô một vùng theo kỹ thuật tản màu dựa trên kỹ thuật nội suy các màu nằm giữa các cặp màu chỉ định dọc theo một trục dạng đường thẳng. Hướng của đường thẳng chính là hướng đổ màu, cần phải chỉ định những màu cần xuất hiện tại các điểm nằm trên đường thẳng này nhờ đối tượng GradientStop. Có thể chỉ định hướng đổ màu nằm ngang từ trái qua phải, nằm dọc từ trên xuống, hay theo đường chéo bất kỳ. Mặc định thì hướng đổ màu sẽ theo đường chéo từ góc trên bên trái tới góc dưới bên phải.

# Linear Gradient Color

- ▶ Thuộc tính StartPoint và EndPoint của LinearGradientBrush xác định điểm đầu và điểm cuối của trục tô mỗi thuộc tính này được xác định bởi một cặp giá trị “x,y” . Trong đó, x và y là các số thực (double) có giá trị từ 0 đến 1.
- ▶ Tọa độ các điểm StartPoint và EndPoint của trục tô ảnh hưởng đến hướng tô màu. Giá trị mặc định của StartPoint là (0,0) và EndPoint là (1,1), nghĩa là trục tô là đường chéo của hình chữ nhật bao.

# Linear Gradient Color

- ▶ Thẻ `<GradientStop Color="Color value" Offset="m.n" />` chỉ định các điểm chốt nằm dọc theo trục tô.
- Thuộc tính Color xác định màu cần tô tại điểm chốt.
- Thuộc tính Offset nhằm xác định vị trí của điểm chốt, giá trị này là một số thực nằm trong khoảng từ 0 đến 1, giá trị càng gần 0 thì càng gần với điểm xuất phát `StartPoint`, giá trị gần với 1 thì càng gần với điểm kết thúc `EndPoint` của trục tô.
- Hệ thống sẽ tự động nội suy các màu nằm giữa các cặp điểm chốt.

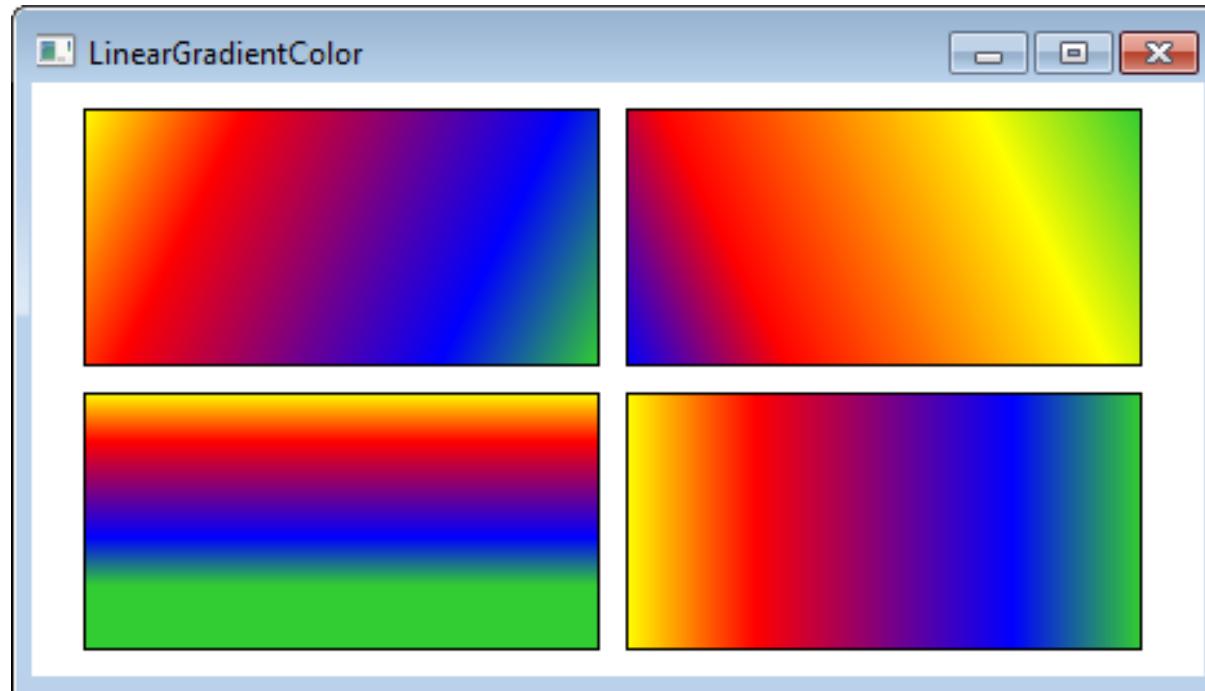
# Linear Gradient Color

```
<Rectangle Canvas.Left="10" Canvas.Top="10" Height="100" Stroke="Black"
Width="200" >
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="Yellow" Offset="0.0" />
            <GradientStop Color="Red" Offset="0.25" />
            <GradientStop Color="Blue" Offset="0.75" />
            <GradientStop Color="LimeGreen" Offset="1.0" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
<Rectangle Canvas.Left="220" Canvas.Top="10" Height="100" Stroke="Black"
Width="200" >
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,1" EndPoint="1,0">
            <GradientStop Color="Blue" Offset="0.0" />
            <GradientStop Color="Red" Offset="0.25" />
            <GradientStop Color="Yellow" Offset="0.75" />
            <GradientStop Color="LimeGreen" Offset="1.0" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

# Linear Gradient Color

```
<Rectangle Canvas.Left="10" Canvas.Top="120" Height="100" Stroke="Black"  
Width="200" >  
    <Rectangle.Fill>  
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,.75">  
            <GradientStop Color="Yellow" Offset="0.0" />  
            <GradientStop Color="Red" Offset="0.25" />  
            <GradientStop Color="Blue" Offset="0.75" />  
            <GradientStop Color="LimeGreen" Offset="1.0" />  
        </LinearGradientBrush>  
    </Rectangle.Fill>  
</Rectangle>  
<Rectangle Canvas.Left="220" Canvas.Top="120" Height="100" Stroke="Black"  
Width="200" >  
    <Rectangle.Fill>  
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5">  
            <GradientStop Color="Yellow" Offset="0.0" />  
            <GradientStop Color="Red" Offset="0.25" />  
            <GradientStop Color="Blue" Offset="0.75" />  
            <GradientStop Color="LimeGreen" Offset="1.0" />  
        </LinearGradientBrush>  
    </Rectangle.Fill>  
</Rectangle>
```

# Linear Gradient Color



# Radial Gradient Color

Kỹ thuật tô đổ màu theo bán kính hình tròn tương tự như kỹ thuật đổ màu tuyến tính, nhưng điểm xuất phát bắt đầu từ tâm đường tròn và màu được lan dần ra ngoài cho tới biên của đường tròn, sử dụng đối tượng tên là RadialGradientBrush. Các điểm chốt vẫn sử dụng đối tượng GradientStop tương tự kỹ thuật đổ màu tuyến tính.

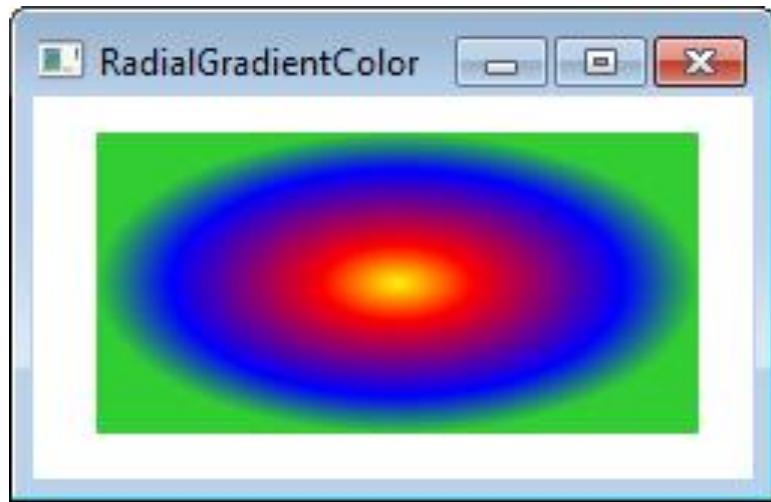
# Radial Gradient Color

- ▶ Các thông số Center - Tọa độ tâm đường tròn
- ▶ RadiusX - Bán kính ngang, RadiusY - Bán kính dọc
- ▶ Thông số GradientOrigin là tọa độ của điểm xuất phát. Các điểm chốt sẽ chạy từ điểm GradientOrigin này, dọc theo đường kính hình tròn.

# Radial Gradient Color

```
<Rectangle Width="200" Height="100" Canvas.Left="21"
    Canvas.Top="12">
    <Rectangle.Fill>
        <RadialGradientBrush GradientOrigin="0.5,0.5"
            Center="0.5,0.5" RadiusX="0.5" RadiusY="0.5">
            <GradientStop Color="Yellow" Offset="0" />
            <GradientStop Color="Red" Offset="0.25" />
            <GradientStop Color="Blue" Offset="0.75" />
            <GradientStop Color="LimeGreen" Offset="1" />
        </RadialGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

# Radial Gradient Color



# Tô bằng ảnh

WPF còn hỗ trợ tô một vùng bằng những hình ảnh có sẵn (ảnh Bitmap, JPG,...) một cách dễ dàng nhờ đối tượng ImageBrush. Chỉ cần đưa tệp ảnh vào tài nguyên của Project và gắn đường dẫn anh cho thuộc tính ImageSource của đối tượng ImageBrush, sau đó dùng chổi tô ImageBrush để tô các đối tượng hình học (Shape), các điều khiển (Control), Panel hay Text,....

# Tô bằng ảnh

- ▶ Thuộc tính `ImageSource` của `ImageBrush` để chỉ định đường dẫn đến tệp hình ảnh.
- ▶ Thuộc tính `Stretch` để chỉ định các co dãn hình khi tô vùng, giá trị mặc định của `Stretch` là `Fill`. Thuộc tính này có các giá trị:
  - `None`: Chỗi tô không tự động co dãn hình.
  - `Uniform`: Chỗi tô co dãn hình trùng khít với một chiều của vùng tô nhưng giữ nguyên tỷ lệ của ảnh gốc.
  - `UniformToFill`: Chỗi tô co dãn hình phủ kín vùng tô nhưng giữ nguyên tỷ lệ của ảnh gốc.
  - `Fill`: Chỗi tô co dãn hình phủ kín vùng tô, không giữ tỷ lệ ảnh. Nếu tỷ lệ hai chiều của vùng tô khác với tỷ lệ hai chiều của ảnh trả ảnh tô sẽ bị méo.

# Tô bằng ảnh

```
<Rectangle Canvas.Left="10" Canvas.Top="10" Height="150"  
    Width="150" Stroke="Black" >  
    <Rectangle.Fill>  
        <ImageBrush ImageSource="Image\image.png" />  
    </Rectangle.Fill>  
</Rectangle>  
<Rectangle Canvas.Left="170" Canvas.Top="10" Height="100"  
    Width="200" Stroke="Black" >  
    <Rectangle.Fill>  
        <ImageBrush ImageSource="Image\image.png"  
Stretch="Fill"/>  
    </Rectangle.Fill>  
</Rectangle>
```

# Tô bằng ảnh



# Transform

WPF cung cấp một số lớp để hỗ trợ cho công việc biến đổi hình học

- ▶ TranslateTransform: thay đổi tọa độ của đối tượng thông qua hai property là X và Y.
- ▶ RotateTransform: xoay đối tượng theo một góc Angle với tâm xác định bởi CenterX và CenterY.
- ▶ ScaleTransform: co giãn kích thước của đối tượng theo tỉ lệ với ScaleX, ScaleY và tâm CenterX, CenterY.
- ▶ SkewTransform: làm xiên đối tượng với góc AngleX, AngleY và tâm CenterX, CenterY.

# Transform

```
<Rectangle Canvas.Left="10" Canvas.Top="10"
    Width="100" Height="100" Fill="RosyBrown">
</Rectangle>
<Rectangle Canvas.Left="10" Canvas.Top="10"
    Width="100" Height="100" Fill="Blue">
    <Rectangle.RenderTransform>
        <TranslateTransform
            X="200"
            Y="0" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Left="80" Canvas.Top="141"
    Width="100" Height="100" Fill="RosyBrown">
</Rectangle>
<Rectangle Canvas.Left="80" Canvas.Top="141"
    Width="100" Height="100" Fill="Blue">
    <Rectangle.RenderTransform>
        <RotateTransform Angle="45" />
    </Rectangle.RenderTransform>
</Rectangle>
```

# Transform

```
<Rectangle Canvas.Left="12" Canvas.Top="305"
           Width="100" Height="100" Fill="RosyBrown">
</Rectangle>
<Rectangle Canvas.Left="12" Canvas.Top="305"
           Width="100" Height="100" Fill="Blue">
    <Rectangle.RenderTransform>
        <ScaleTransform
            ScaleX="0.5"
            ScaleY="0.5"
            CenterX="0"
            CenterY="0"/>
    </Rectangle.RenderTransform>
</Rectangle>
```

# Transform

```
<Rectangle Canvas.Left="12" Canvas.Top="425"
           Width="100" Height="100" Fill="RosyBrown">
</Rectangle>
<Rectangle Canvas.Left="12" Canvas.Top="425"
           Width="100" Height="100" Fill="Blue" Opacity="0.5">
    <Rectangle.RenderTransform>
        <SkewTransform
            AngleX="10"
            AngleY="10"
            CenterX="0"
            CenterY="0"/>
    </Rectangle.RenderTransform>
</Rectangle>
```

# Transform

