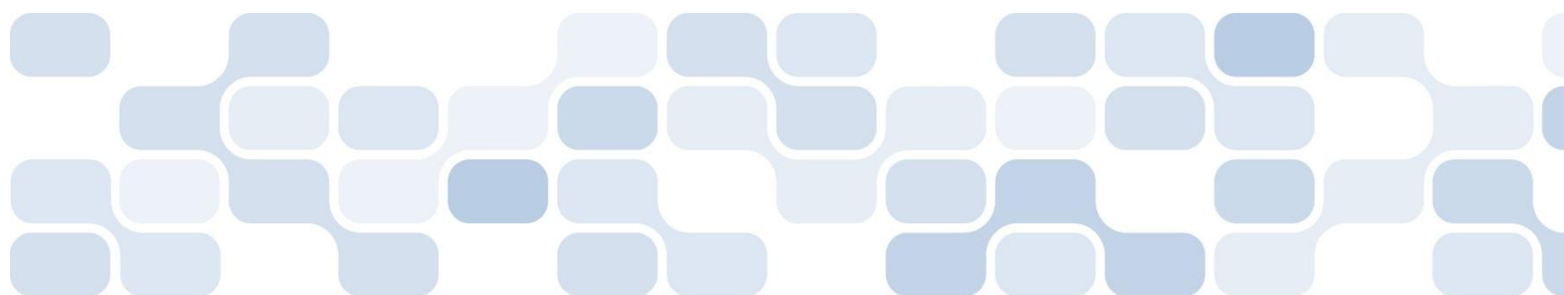
Visual Studio Do-It-Yourself シリーズ

第 4 回 イベント



Visual Studio Do-It-Yourself
第 4 回 イベント

今回は、ユーザー インターフェイスとプログラムがどのように動作するかというロジックを連動するための仕組みについて学んでいきます。

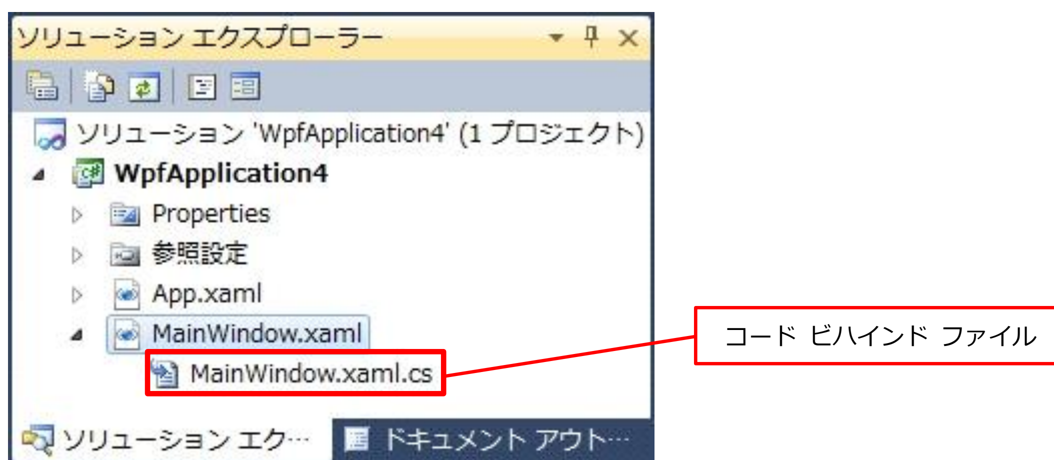
ここでは、次のことを学習します。

- イベントとイベント ハンドラー
- コマンドの使用
- トリガー

■イベントとイベント ハンドラー

これまでに学んできたことは、ほとんどがユーザー インターフェイスというアプリケーションの外観をどのようにデザインするかということでした。実際のアプリケーションでは、ユーザー インターフェイスから入力されたデータを処理したり、処理した結果をユーザー インターフェイスに表示したりするためのプログラム コードを連動させる必要があります。

このとき、ユーザー インターフェイス側でプログラム コードを呼び出すような要因となるものを「イベント」と言い、これに対応して実行するプログラムを「イベント ハンドラー」と言います。イベント ハンドラーは、ウィンドウの定義（拡張子.xaml）に対応するソースコードで定義します。ソリューション エクスプローラーでは、.xaml ファイルの左側にある三角記号をクリックして表示される.xaml.cs という拡張子のファイルです（これを「コード ビハインド ファイル」と言います）。



イベント ハンドラーは、C#（または Visual Basic）言語を使って記述します。本書では、言語の詳細については取り上げません。C#言語については、MSDN オンラインの「C#言語」などを参照してください。
Windows アプリケーション開発でよく使われるイベントを次の表に示します。

イベント	主なコントロール	機能
Click	Button 、 CheckBox 、 RadioButton など	マウスでクリックしたり、入力フォーカスがあるときに［スペース］キーを押すことで発生
Checked	CheckBox、RadioButton	クリックしてチェックされたり、プログラムで IsChecked プロパティに true を代入したときに発生
Unchecked	CheckBox、RadioButton	クリックしてチェックが外れたり、プログラムで IsChecked プロパティに false を代入したときに発生
TextChanged	TextBox	キー入力したり、Text プロパティが変更されたときに発生
SelectionChanged	ListBox、ComboBox など	項目をクリックしたり、プログラムで SelectedIndex に代入して、選択されている項目が変わったときに発生（項目が変わらないときは発生しない）
ValueChanged	ProgressBar、Slider など	Value プロパティが変更したときに発生
Loaded	ほとんどのコントロール（主に Window で使う）	レイアウト処理やプロパティの設定が終わった後に発生（再レイアウトするような変更を行うと、このイベントが再発生する可能性がある）
GetFocus	ほとんどのコントロール	入力フォーカス（キーボードからの入力を受け付けられる状態）を受け取ったときに発生
LostFocus	ほとんどのコントロール	入力フォーカスを失った時に発生
KeyDown	ほとんどのコントロール	キーが押されたときに発生（キーリピートのときは連続して発生）
KeyUp	ほとんどのコントロール	キーが離されたときに発生します
MouseEnter	ほとんどのコントロール	マウスカーソルがコントロール上に入ってきたときに発生
MouseMove	ほとんどのコントロール	マウスカーソルがコントロール上を移動しているときに発生
MouseLeave	ほとんどのコントロール	マウスカーソルがコントロール上を出ていくときに発生
MouseDown	ほとんどのコントロール	マウスのボタンが押されたときに発生
MouseUp	ほとんどのコントロール	マウスのボタンが離されたときに発生
MouseLeftButtonDown	ほとんどのコントロール	マウスの左ボタンが押されたときに発生
MouseLeftButtonUp	ほとんどのコントロール	マウスの左ボタンが離されたときに発生
MouseRightButtonDown	ほとんどのコントロール	マウスの右ボタンが押されたときに発生
MouseRightButtonUp	ほとんどのコントロール	マウスの右ボタンが離されたときに発生
MouseWheel	ほとんどのコントロール	マウスのホイールを回転したときに発生
SizeChanged	ほとんどのコントロール	大きさが変更されたときに発生

■イベントにイベント ハンドラーを割り当てる

コントロールのイベントにイベント ハンドラーを割り当てるには、WPF デザイナーでコントロールをダブルクリックする、プロパティ ウィンドウを使う、XAML 定義で直接割り当てる、プログラムで割り当てるといった、さまざまな方法があります。ここでは、よく使われる最初の 2 つの方法について説明します。

●WPF デザイナーでコントロールをダブルクリックする

これまでも、ウィンドウに配置されている Button をダブルクリックして、Click イベントに対するイベント ハンドラーを定義しました。同じように TextBox をダブルクリックすると、そのテキストボックスの TextChanged イベントに割り当てるイベント ハンドラーが自動的に生成されて割り当てられます。このように、コントロールをダブルクリックすると、そのコントロールでよく使われるイベントに対して、空のイベント ハンドラーが生成されて割り当てられます（Label や TextBlock のように、あまりイベントが使われないものはダブルクリックしても何も起きません）。

次のようにウィンドウ上に配置された ListBox コントロールをダブルクリックします。

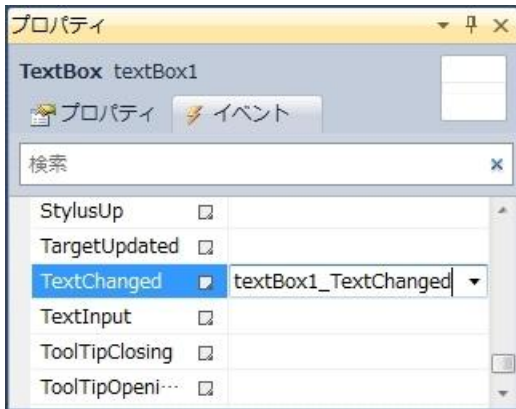
すると、対応する.xaml.cs 中に、次のように空のイベント ハンドラーが生成されます。このイベント ハンドラーは、上記の TextBox コントロールの TextChanged イベントに割り当てられています。

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

  private void textBox1_TextChanged(object sender, TextChangedEventArgs e)
  {
    |  ←ここにカーソルが表示される
  }
}
```

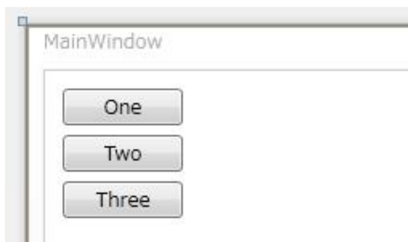
●プロパティ ウィンドウを使う

これまでプロパティを設定するために使っていたプロパティ ウィンドウで、[イベント] というタブをクリックすると、選んでいるコントロールのイベント一覧が表示されます。それぞれの右側の空欄をダブルクリックすると、空のイベント ハンドラーが生成されて、割り当てられます。

TextBox を配置して、プロパティ ウィンドウの [イベント] タブで TextChanged の右側をダブルクリックすると、前述と同じような空のイベント ハンドラーが生成されます。



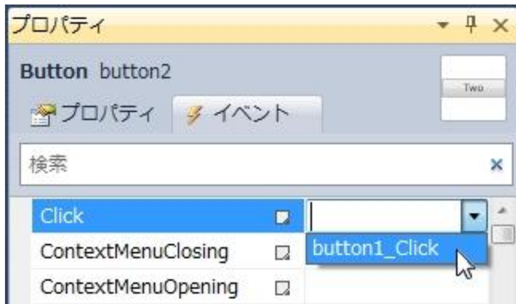
イベント ハンドラーは、イベントごとに別々である必要はありません。同じ引数を持つメソッド（関数）であれば、他のコントロールや他のイベントに割り当てることもできます。たとえば、3 つの Button コントロールを配置して、それぞれの Content を「One」「Two」「Three」としておきます（下図）。



最初のボタン（One）をダブルクリックして、次のようなイベント ハンドラーを割り当てます。

```
private void button1_Clicked(object sender, TextChangedEventArgs e)
{
  MessageBox.Show("Clicked!");
}
```

これは、ボタンを押すと「Clicked!」というメッセージを表示するイベント ハンドラーです。このイベント ハンドラーを他の 2 つのボタンの Click イベントに割り当てることができます。2 番目のボタン（button2）を選んで、プロパティ ウィンドウの [イベント] ページで Click イベントの右側で、下向きの▼ボタンをクリックすると、次のように表示されます。



これを選べば、button2 を Click したときに、さきほど生成された button1_Click が呼び出されることになります。3 番目のボタンについても同様です。複数のボタンをまとめて選択すれば、プロパティ ウィンドウで一度にイベント ハンドラーを割り当てることもできます。

どのコントロールで発生したイベントなのかはイベント ハンドラーの第 1 引数（object 型の sender）で判別します。たとえば、次のようにプログラムすれば、ボタンごとに違うメッセージを表示できます。

```
private void button1_Click(object sender, RoutedEventArgs e)
{
  if (sender == button1)
    MessageBox.Show("One Clicked");
  else if (sender == button2)
    MessageBox.Show("Two Clciked");
  else
    MessageBox.Show("Three Clicekd");
}
```

sender が Button 型ではなく object 型なのは、異なるコントロールのイベントに割り当てられるかもしれないためです。たとえば、button1_Click は CheckBox の Click イベントに割り当てることもできます。

イベントを呼び出したものが Button コントロールかどうかを判別して、次のようにプログラムすることもできます。

```
private void button1_Click(object sender, RoutedEventArgs e)
{
  if (sender is Button)
  {
    Button btn = sender as Button;
    MessageBox.Show(btn.Content.ToString() + " Clicked");
  }
}
```

■イベント ハンドラーの引数

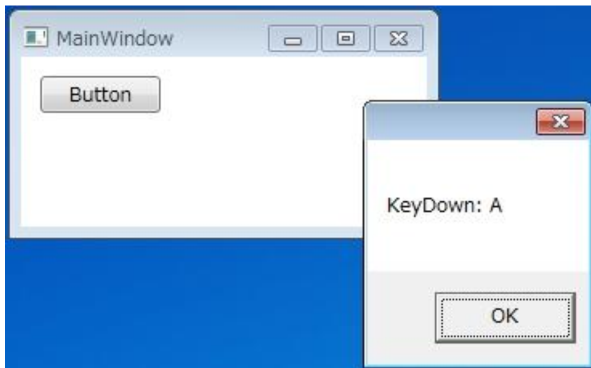
通常、イベント ハンドラーは 2 つの引数を受け取ります。前述のとおり、第 1 引数にはイベントを発生させたコントロールが渡されます。そして、第 2 引数は、そのイベント固有の情報が渡されます。

新しいプロジェクトを作成して、ウィンドウに Button コントロールを配置し、プロパティ ウィンドウで KeyDown イベントをダブルクリックして、次のようなイベント　ハンドラーを割り当てます。

```
private void button1_KeyDown(object sender, KeyEventArgs e)
{
  MessageBox.Show("KeyDown: " + e.Key.ToString());
}
```

ここでは KeyDown というキーが押されたイベントに対して、イベント ハンドラーの第 2 引数には KeyEventArgs 型の値 e が渡されます。これには、Key 以外にもキーリピートの状態や [Num Lock] キーのトグル状態などが含まれています。

プログラムを実行して、ボタンを選んでからキーを押すと次のように表示されます。



第 2 引数の型は、イベントによって異なります。Click のようにイベント特有の情報を持たない場合は、RoutedEventArgs 型が使われます。

前述のとおり、異なるコントロールやイベントで、同一のイベント ハンドラーを使うことはできますが、第 2 引数の型が違う場合は渡される情報が異なるため共通化はできません。

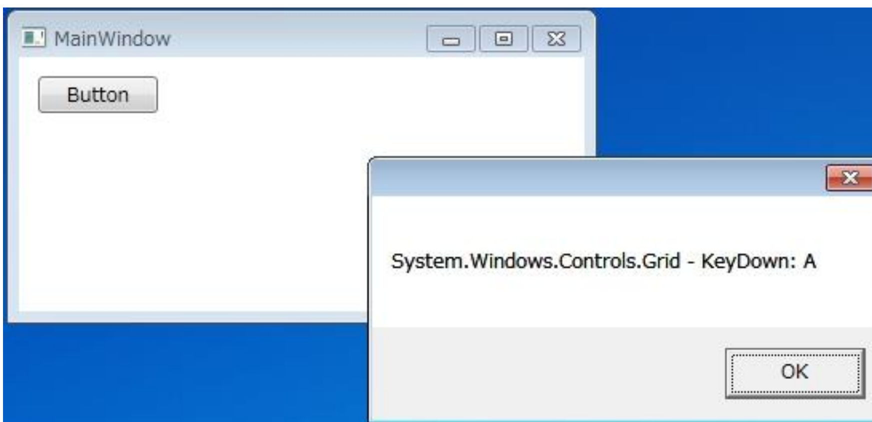
■イベントのルーティング

イベントの発生でもう一つ注意すべきことが「ルーティング」という構造です。

上記の例で、ボタンに対して発生した KeyDown というイベントは、ボタンが配置されている Grid、そして Grid が配置されている Window に対して伝搬していきます。この様子を確かめるため、Grid を選び、プロパティ ウィンドウで KeyDown イベントに button1_KeyDown を選んでください。さらに、外側の Window を選んで同じように KeyDown イベントに button1_KeyDown を選んでください。button1_KeyDown イベント ハンドラーは次のように変更しておくとよいでしょう。

```
private void button1_KeyDown(object sender, KeyEventArgs e)
{
  MessageBox.Show(sender.ToString() + " - KeyDown: " + e.Key.ToString());
}
```

プログラムを実行して、ボタンを選び、キーボードを押すとメッセージボックスが 3 回表示されます。表示される文字列を見ると Button、Grid、Window の順に呼び出されていることがわかります。



もし、次のように第 2 引数 e の Handled プロパティを true にすると、メッセージボックスは 1 回しか表示されません。

```
private void button1_KeyDown(object sender, KeyEventArgs e)
{
  MessageBox.Show(sender.ToString() + " - KeyDown: " + e.Key.ToString());
  e.Handled = true;
}
```

Handled プロパティを true にすることは、そのイベント ハンドラーで「イベントを処理したので、これ以上の処理は必要ない」ということを意味します。このため、上位のコントロールにイベントが伝搬されなくなるのです。

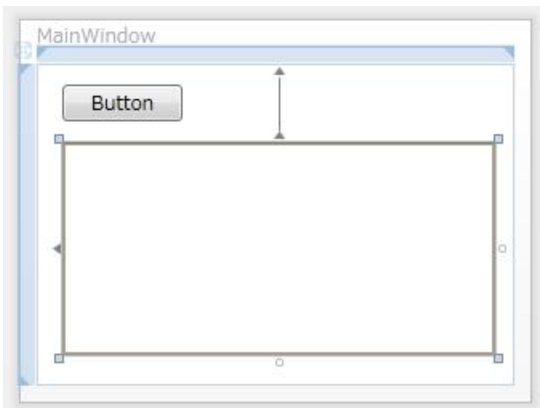
■コマンドの使用

ボタンを押したり、メニューを選ぶといった何かの機能を呼び出す動作については「コマンド」という仕組みがあります。第 3 回のコントロールの説明で、ツールバーに配置したボタンには、イベント ハンドラーではなく Cut、Copy、Paste といったコマンドを割り当てました。これらは、ApplicationCommands というクラスのプロパティとして用意されているもので、それぞれクリップボードへの切り取り、コピー、貼り付け動作に対応しています。

コマンドの便利なところは、アプリケーション全体から呼び出せるような機能をコマンドとして用意しておくことで、特定のウィンドウに対してイベント ハンドラーを定義せずに、必要なコントロールに対して Command プロパティを割り当てるだけでその機能を呼び出せることです。あらかじめ用意されているコマンドには、ApplicationCommands クラス以外に含まれるもの以外にも NavigationCommands（BrowseForward、Browseback など）や EditingCommands（AlignCenter、Delete、IncreaseFontSize など)、MediaCommands（Play、Pause、Stop など）などがあります。ただし、いつもツールバーのボタンにコマンドを割り当てるように簡単に使えるわけではありません。

Menu や ToolBar コントロールは、コマンドのルーティング（伝搬）を独自に処理して、自分自身に配置されているコントロールへ分岐する仕組みを持っています。また、TextBox はクリップボードに対する操作を内蔵しており、ApplicationCommands のクリップボード用のコマンドはこれを使う仕組みを持っています。したがって、メニューの項目やツールバーに配置したボタンは、Command プロパティにクリップボード用のコマンドを割り当てるだけで、その機能が使えるようになります。

通常は、もう少し複雑な手順が必要です。ここで、新しい WPF アプリケーションに Button と TextBox を配置します。TextBox の TextWrapping プロパティは Wrap にし、AcceptsReturn プロパティをチェックしておくとよいでしょう。



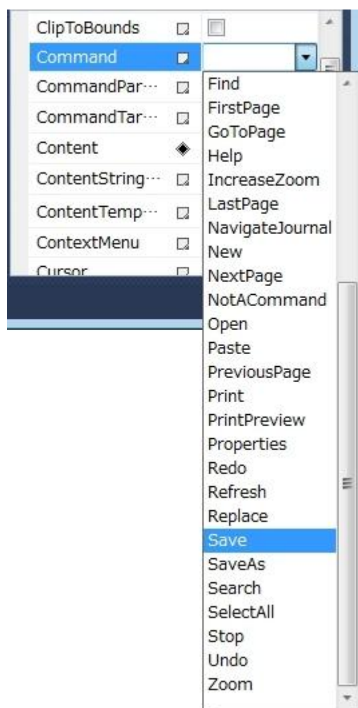
ここで、[F7] を押して、このウィンドウの定義（MainWindow.xaml）に対応するコードビハインドファイル（MainWindow.xaml.cs）を開きます。

MainWindow クラスは、次のように記述します。

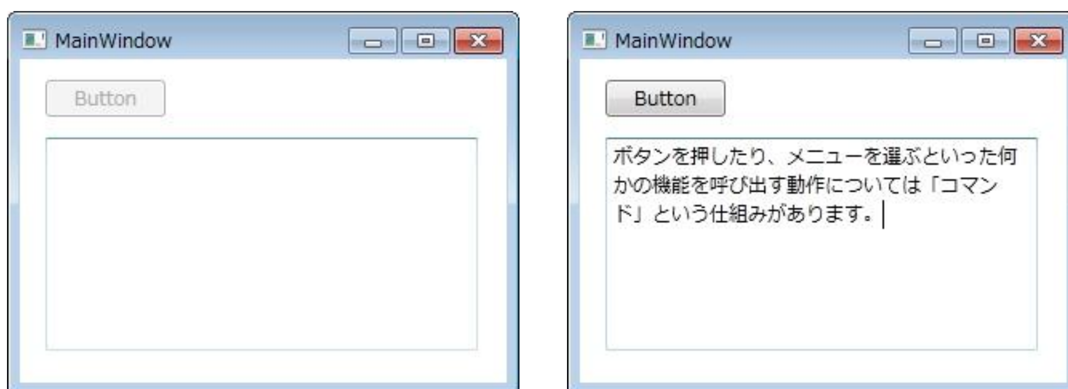
```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();

    CommandBinding binding = new CommandBinding(ApplicationCommands.Save);
    binding.CanExecute+=new CanExecuteRoutedEventHandler(binding_CanExecute);
    binding.Executed += new ExecutedRoutedEventHandler(binding_Executed);
    CommandBindings.Add(binding);
  }

  void binding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
  {
    e.CanExecute = textBox1.Text.Length > 0;
  }

  void binding_Executed(object sender, ExecutedRoutedEventArgs e)
  {
    var dlg = new Microsoft.Win32.SaveFileDialog();
    dlg.Filter = "text file|*.txt";
    if (dlg.ShowDialog() == true)
    {
      using (var writer = new System.IO.StreamWriter(dlg.FileName))
        writer.Write(textBox1.Text);
    }
  }
}
```

コマンドは、実際の機能と結びつける必要があります（バインディング）。MainWindow クラスでは、そのコンストラクターで、ApplicationCommands.Save というコマンドに対する「バインディング」を作成しています。バインディングの CanExecute イベント ハンドラーは、テキストボックスに何か入力されている場合に true を返します。このイベントは、コマンドが有効かどうかを判断するため、しばしば呼び出されるものです。Executed イベント ハンドラーは、ファイルを保存するダイアログを開いて、テキストボックスの内容を保存します。最後に、ウィンドウ クラス自身の CommandBindings というプロパティに、新しいバインディングを追加しています。

WPF デザイナーに戻り、ボタンを選んでプロパティ ウィンドウで Command プロパティの右側の下向き▼を押します。すると、選択できるコマンド一覧が表示されるので「Save」を選びます。



作成したアプリケーションを実行すると、最初はボタンが無効になっていますが、テキストボックスに何か入力するとボタンが有効になります。ここでボタンをクリックすれば、テキストの内容を指定したファイルに保存できます。



メニューやツールバーを作成するときも、そのメニュー項目やツールバーに配置したボタンの Command プロパティを設定するだけで、コマンドに結びつけた機能を呼び出すことができます。

コマンドと機能のバインディングは、プログラムで追加するだけでなく、XAML の記述として定義することもできます。また、定義済みのコマンドを使うだけでなく、アプリケーション独自のコマンドを作成することもできます。

■**トリガー**

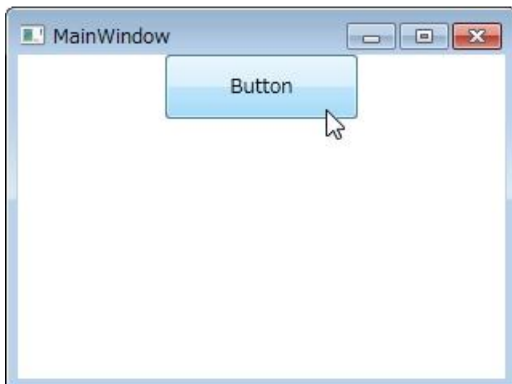
トリガーは、イベント ハンドラーを定義せずに動きを与える仕組みです。

トリガーには、プロパティ トリガー、イベント トリガー、データ トリガーという種類があります。プロパティ トリガーはもっとも単純なもので、コントロールのプロパティ値を使った条件に合う場合に、（別の）プロパティの値を変更します。

トリガーは、WPF デザイナーでは設定できないため、XAML エディターを使います。新しい WPF アプリケーションの XAML を次のように編集します（なお、通常は第 7 回で説明するリソースを使います）。

```
<Window …>
  <StackPanel>
    <Button Width="120">
      <Button.Style>
        <Style TargetType="Button">
          <Style.Triggers>
            <Trigger Property="IsMouseOver" Value="true">
              <Trigger.Setters>
                <Setter Property="Height" Value="40"/>
              </Trigger.Setters>
            </Trigger>
          </Style.Triggers>
        </Style>
      </Button.Style>
      Button
    </Button>
  </StackPanel>
</Window>
```

すると、マウスカーソルがボタン上にある場合（IsMouseOver プロパティが true になる場合）に、ボタンの高さが広がります。



■まとめ

今回は、イベントについて学びました。イベント　ハンドラーをどのように記述するかについては、プログラミング言語についての学習も必要です。C#や Visual Basic は、わかりやすく強力な機能を持つ言語です。また、これまでに Visual Studio でプログラミングした経験があれば、新しい言語の特長も学びやすいでしょう。

次回は、データベースの取り扱いを含むデータ バインディングについて学びます。