

WINDOWS PRESENTAION FOUNDATION 4.5 入門

目次

1.	はじめに	8
1.1.	本書の目的	8
1.2.	本書の対象者	8
1.3.	執筆環境	9
2.	WPF とは	9
2.1.	WPF のプログラミングモデル	10
2.1.1.	WPF で作成可能なアプリケーション	10
2.2.	Hello world	13
2.2.1.	App クラス	13
2.2.2.	MainWindow.xaml	15
2.2.3.	デザイナーによる画面の設計	16
2.2.4.	イベントハンドラの追加とコードの記述	18
2.2.5.	コンパイルして実行	18
2.2.6.	Main メソッドはどこにいった？	19
2.3.	全て C# で Hello world	21
2.4.	WPF を構成するものを考えてみる	25
2.4.1.	WPF はクラスライブラリ？	25
2.4.2.	XAML は何？	26
2.4.3.	WPF を構成するもののまとめ	27

2.5.	WPF のコンセプト	28
2.5.1.	コンテンツモデル	28
2.6.	スタイル	32
2.7.	データバインディング	35
2.8.	まとめ	37
3.	XAML	37
3.1.	オブジェクト要素と XAML 名前空間	38
3.2.	オブジェクト要素のプロパティ	39
3.3.	コレクション構文	43
3.4.	コンテンツ構文	45
3.5.	マークアップ拡張	47
3.5.1.	ProvideValue メソッドの IServiceProvider って何者?	49
3.6.	添付プロパティ	50
3.7.	添付イベント	50
3.8.	TypeConverter	50
3.9.	その他の機能	51
4.	WPF のコントロール	51
4.1.	レイアウト	52
4.1.1.	Border コントロール	52
4.1.2.	BulletDecorator コントロール	54
4.1.3.	Canvas コントロール	55

4.1.4.	StackPanel コントロール	57
4.1.5.	DockPanel コントロール.....	61
4.1.6.	WrapPanel コントロール	63
4.1.7.	ViewBox コントロール	66
4.1.8.	ScrollViewer コントロール	68
4.1.9.	Grid コントロール	74
4.1.10.	Grid コントロールでのレイアウト例	80
4.1.11.	GridSplitter コントロール	84
4.1.12.	レイアウトに影響を与えるプロパティ	87
4.2.	ボタン	90
4.2.1.	Button コントロール.....	90
4.2.2.	RepeatButton コントロール.....	92
4.3.	データ表示.....	93
4.3.1.	DataGrid コントロール.....	93
4.3.2.	TreeView コントロール	105
4.4.	日付表示および選択	113
4.4.1.	Calendar コントロール.....	113
4.4.2.	DatePicker コントロール	119
4.5.	メニュー	122
4.5.1.	ContextMenu コントロール.....	122
4.5.2.	Menu コントロール.....	124

4.5.3. ToolBar コントロール	126
4.6. 選択系コントロール	129
4.6.1. CheckBox コントロール	129
4.6.2. ComboBox コントロール	131
4.6.3. ListBox コントロール	135
4.6.4. RadioButton コントロール	136
4.6.5. Slider コントロール	138
4.7. ナビゲーションコントロール.....	142
4.7.1. TabControl.....	142
4.8. ファイルダイアログ	145
4.9. 情報を表示するコントロール.....	149
4.9.1. Label コントロール.....	149
4.9.2. ProgressBar コントロール	150
4.9.3. StatusBar コントロール	151
4.9.4. TextBlock コントロール.....	151
4.9.5. Popup コントロール	152
4.9.6. ToolTip コントロール.....	154
4.10. 入力	155
4.10.1. TextBox コントロール.....	155
4.11. メディア.....	157
4.11.1. Image コントロール	157

4.11.2. MediaElement コントロール	160
5. WPF deep dive	161
5.1. DispatcherObject	162
5.2. WPF のプロパティシステム	165
5.2.1. 依存関係プロパティ	165
5.2.2. 添付プロパティ	176
5.3. WPF のイベントシステム	179
5.3.1. ルーティングイベントの定義方法	180
5.3.2. イベントのキャンセル	181
5.3.3. 添付イベント	182
5.4. コンテンツモデル	182
5.4.1. DataTemplate	183
5.4.2. DataTemplateSelector	188
5.5. WPF のアニメーション	190
5.5.1. By による値の指定方法	194
5.5.2. アニメーションの繰り返しの指定	194
5.5.3. その他の型のアニメーション	195
5.5.4. コードからのアニメーション	196
5.5.5. キーフレームアニメーション	196
5.5.6. イージング関数	198
5.5.7. Blend によるアニメーションの作成	199

5.6.	Style	204
5.6.1.	スタイルの継承	204
5.6.2.	トリガー	205
5.7.	リソース	206
5.7.1.	リソースの定義	206
5.7.2.	リソースの参照方法	207
5.8.	ControlTemplate	208
5.9.	コントロールの作成	211
5.9.1.	UserControl	212
5.9.2.	カスタムコントロール	219
5.10.	Binding	225
5.10.1.	単純な Binding	225
5.10.2.	Binding の Mode	227
5.10.3.	ElementName によるソースの指定	228
5.10.4.	RelativeSource によるソースの指定	229
5.10.5.	入力値の検証	229
5.10.6.	コレクションのデータバインド	235
5.11.	コマンド	249
6.	応用	254
6.1.	Behavior	254
6.1.1.	Trigger と Action	255

6.1.2.	Trigger と Action の使い方	255
6.1.3.	Behavior	259
6.1.4.	Behavior や Trigger や Action の作成	261
6.2.	データバインディングを前提としたプログラミングモデル	267
6.2.1.	MVVM パターンとは	267
6.2.2.	変更通知の仕組み	268
6.2.3.	ユーザーからの入力の処理	270
6.2.4.	最初のアプリケーションの作成	270
6.2.5.	四則演算アプリケーション	275
7.	さいごに	287

1. はじめに

1.1. 本書の目的

2012 年 8 月に .NET Framework 4.5 がリリースされました。対応する開発環境として Visual Studio 2012 もリリースされ、Windows 8 の時代に対応するアプリケーション開発の環境が整ってきています。Windows 8 で動くアプリケーションには大別して Windows ストア アプリとデスクトップアプリケーションの 2 種類があります。

Windows ストア アプリが注目されがちですが、デスクトップアプリケーションも従来と変わらず重要なファクターになります。今後は、デスクトップアプリケーションにもタッチ対応スクリーンへの対応や、拡大されたときの表示などに対応することが求められます。それに対応するためには従来の Windows Form よりも Windows Presentation Foundation(以下 WPF)のほうが有利になります。

本書では、日本語としてまとまった情報が MSDN 以外にあまりない WPF 4.5 の現状を著者の自習も兼ねながらまとめることを目的としています。そのため、間違った情報を含んでいる可能性あるため、その際は以下の連絡先に連絡ください。

大田 一希

k.ota.0130@gmail.com

1.2. 本書の対象者

本書は、以下のような方を意識して書いています。

- WPF 4.5 の学習をしたい方
- C# についての基本的な知識については知っている方
具体的には以下のキーワードについて知っている方
 - LINQ
 - async, await
 - ラムダ式
- Visual Studio 2012 の基本的な操作方法について理解しているかた
 - プロジェクトの新規作成やクラスなどの新規作成方法
 - 参照の追加方法
 - NuGet を使った参照の追加方法など

1.3. 執筆環境

本書は、以下の環境で作成しています。

- Windows 8 Pro 64bit/Windows 8.1 Pro Update 1
- Visual Studio 2012 Ultimate/Visual Studio 2013 Ultimate
(おそらく Express Edition でも同様に作成可能なものが主になると思います)

2. WPF とは

MSDN の WPF の概要の章に以下のように説明されています。

WPF の概要より抜粋 <http://msdn.microsoft.com/ja-jp/library/aa970268.aspx>

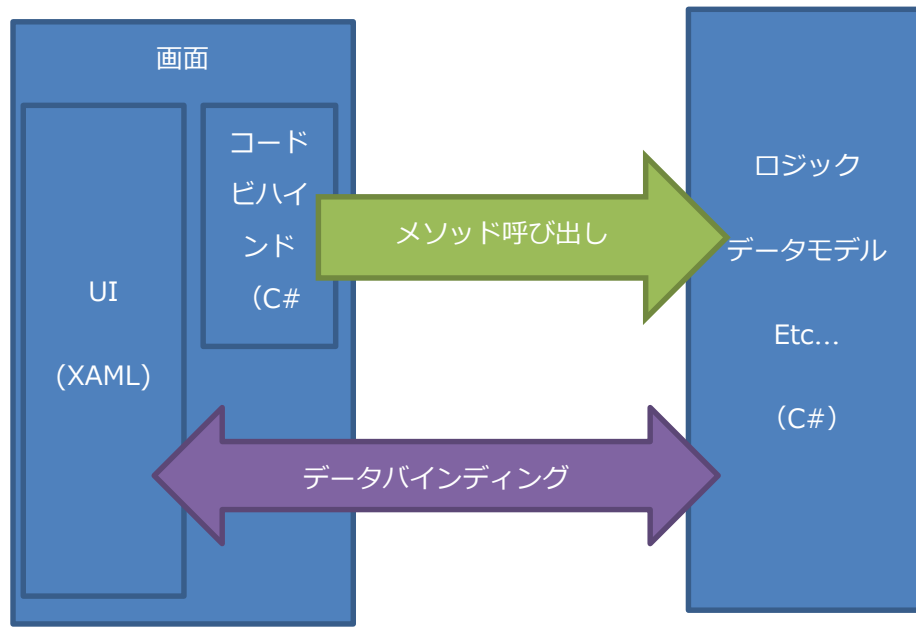
Windows Presentation Foundation とは、魅力的な外観のユーザーエクスペリエンスを持つ Windows クライアント アプリケーションを作成するための次世代プレゼンテーション システムです。

.NET Framework 3.0 から搭載されている WPF の概要をそのまま引き継いでいるためだと思いますが、次世代のプレゼンテーションシステムというよりは、現在求められようとしているデスクトップアプリケーションを開発するために必要な機能が含まれたプラットフォームという言い方のほうがしっくりくると思います。WPF が登場してからの GUI アプリケーションを作るプラットフォームがすべて WPF の考えを踏襲して作られていることから、このことが伺えます。

- Silverlight
WPF/E という名前で開発され、Silverlight という名前でリリースされた。WPF と同じプログラミングモデルで開発が可能。
- Silverlight for Windows Phone
Silverlight を Windows Phone 向けにしたもの
- Windows Runtime
Windows ストア アプリ開発のためのプラットフォームで、WPF と同じプログラミングモデルで開発が行えるスタイルが提供されている。
- Windows Phone Runtime
Windows Runtime と Silverlight for Windows Phone を混ぜたような雰囲気を醸し出しているプラットフォーム。当然 WPF と同じプログラミングモデルで開発が行える。

2.1. WPF のプログラミングモデル

WPF で確立されたプログラミングモデルをベースに、最近の GUI アプリケーション開発のためのプラットフォームが作成されていることについて紹介しました。では、WPF のプログラミングモデルとはどのようなものか、ここで簡単に説明したいと思います。WPF のアプリケーションは、主に以下のような形で作成されます。(カッコ内は記述言語)



XAML（ザムルと読みます）と、C#によって画面を作成し、それとロジックやデータモデルをデータバインディングやメソッド呼び出しによって連携させて1つのアプリケーションとして作成します。

2.1.1. WPF で作成可能なアプリケーション

ここでは、WPF で作成可能なアプリケーションの形態について説明します。

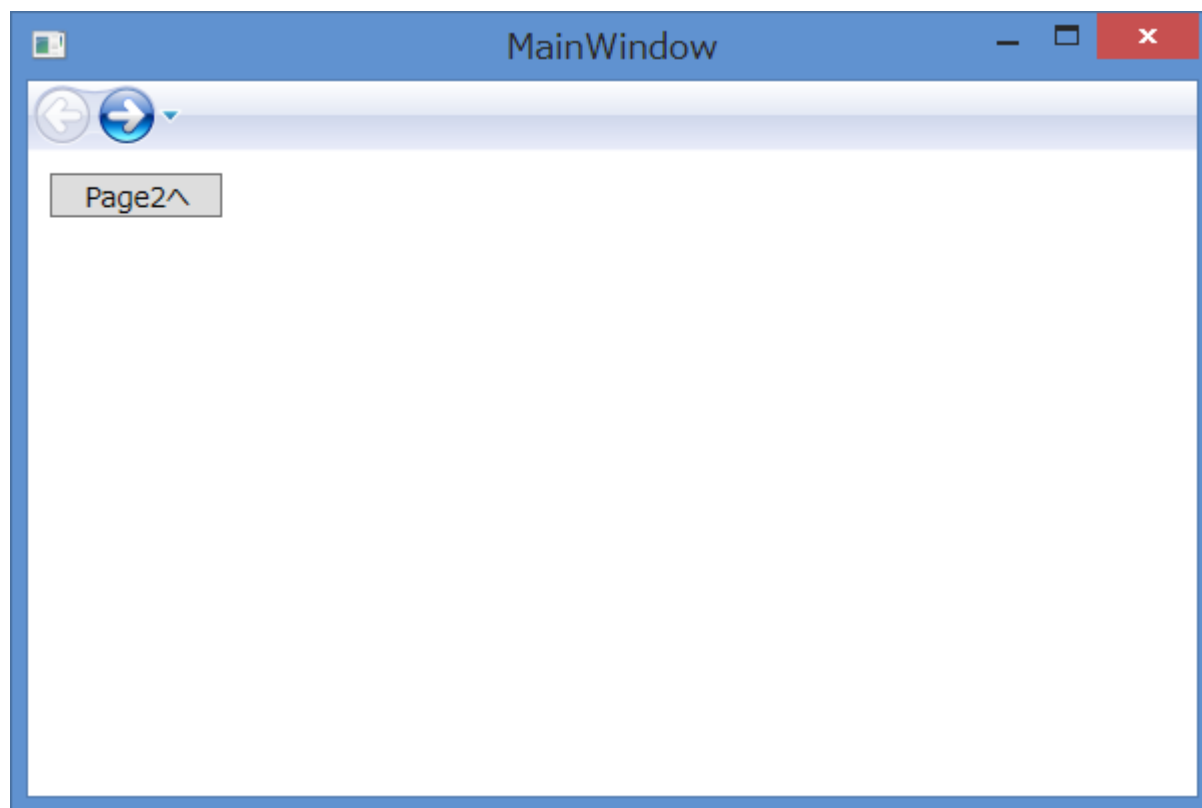
2.1.1.1. Windows アプリケーション

デスクトップアプリケーションとして一般的な複数のウィンドウを持つアプリケーションです。Windows 標準のメモ帳や、ペイントのようなアプリケーションです。Window クラスをベースにして作成します。本書で扱うものは、ほぼ全てこの形のアプリケーションを前提としています。

2.1.1.2. ナビゲーション アプリケーション

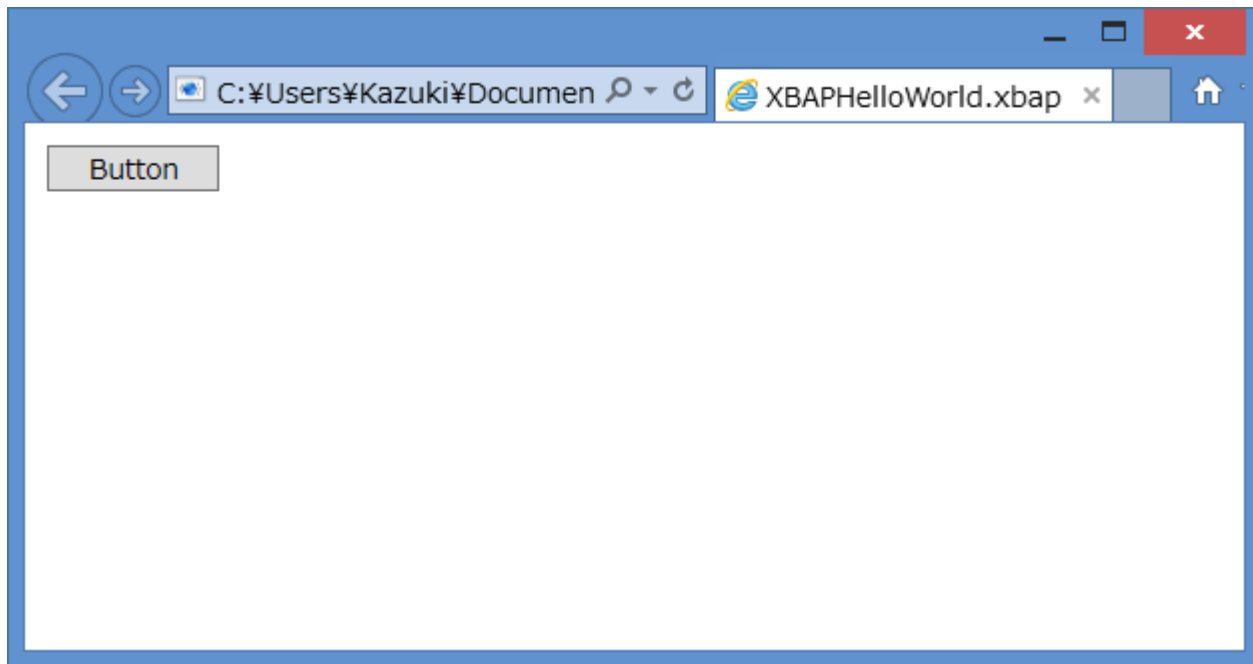
Window の中でブラウザのような画面遷移を行うかたちのアプリケーションです。NavigationWindow クラスをベースにして作成します。NavigationWindow の中で Page クラスを使って作成した画面を切り替えることが出来ます。ブラウザのように戻るや進むといった画面遷移の履歴もコントロールできます。ナビゲーション アプリケーションの画面を以下に示します。画面上部にブラウザのような進むボタンと戻るボタンがあるのが特徴です。このボタ

ンは、表示・非表示を切り替えることができるので、単一画面内でページが切り替わるアプリケーションの作成に向いています。

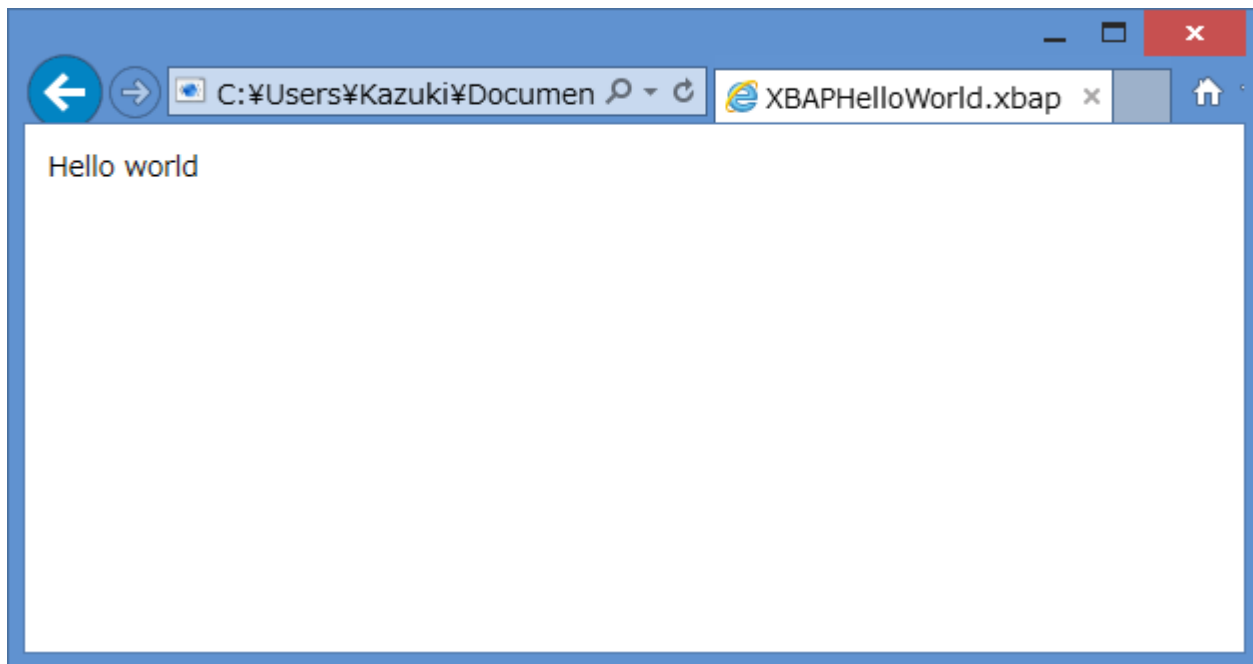


2.1.1.3. XAML ブラウザー アプリケーション

WPF は、デスクトップアプリケーションの他に、XAML ブラウザー アプリケーション(XBAP)というブラウザにホストする形のアプリケーションを開発できます。XBAP は、ブラウザの中に WPF で作成したページを表示することができます。ナビゲーションアプリケーションと同様に Page クラスを継承した画面を遷移する形のアプリケーションです。



ページ単位での画面遷移も行えます。

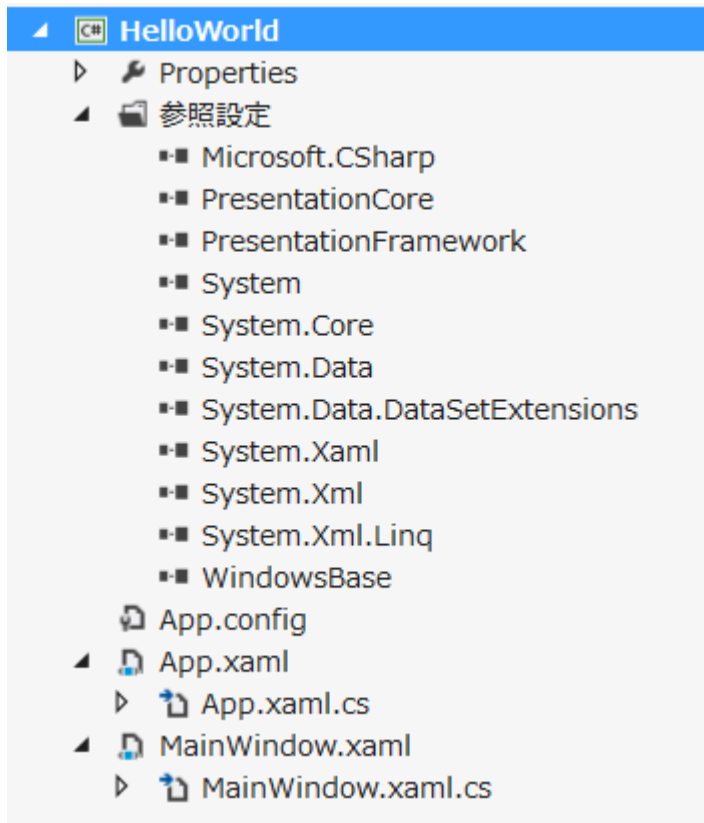


クライアントに .NET Framework が必要な点と、インターネットやイントラネットを経由して配備する際にセキュリティに関する制限事項がある点を除くと、通常の WPF と同じように開発ができます。

2.2. Hello world

ここでは、Visual Studio 2012 を使って簡単な Hello world アプリケーションを作成して、WPF のアプリケーション作成の流れと、アプリケーションの構成要素がどんなものか実際に見て行こうと思います。

Visual Studio 2012 で WPF アプリケーションを新規作成すると以下のような画面になります。



参照設定に、PresentationCore と PresentationFramework と WindowsBase と System.Xaml という 4 つが追加されています。この 4 つが WPF のクラスを含むアセンブリになります。そのほかに、App.xaml(xaml.cs とペア)や MainWindow.xaml(xaml.cs とペア)が作成されています。

2.2.1. App クラス

App.xaml は、以下のような内容の XAML で書かれたファイルになります。XAML は、WPF では主に GUI を記述するための言語として使われますが App.xaml では、GUI ではなくアプリケーション全体を制御するクラスを定義しています。App.xaml のコードを以下に示します。

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

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

見ていただければわかると思いますが XAML は XML をベースとして作られています。XML 名前空間や XML の開始タグや閉じタグがあります。x:Class は、この XAML と対になるコードビハインドのクラスを表しています。

HelloWorld.App クラスは、App.xaml.cs の中に以下のように定義されています。

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace HelloWorld
{
    /// <summary>
    /// App.xaml の相互作用ロジック
    /// </summary>
    public partial class App : Application
    {
    }
}
```

初期状態では、何も定義されていません。この App.xaml と App.xaml.cs は、コンパイル時に 1 つのクラスとして解釈されます。そのため、App.xaml で定義してあることと、App.xaml.cs で記述したコードが 1 つ App クラスになります。**App クラスは、従来のアプリケーションでいうところの Main メソッドを持つエントリーポイントのクラスになります。**

App.xaml で重要な点は、StartupUri 属性で MainWindow.xaml を指定している点です。StartupUri で指定したウィンドウを起動時に表示するようになっているため、このアプリケーションを実行すると MainWindow.xaml が表示されます。

2.2.2. MainWindow.xaml

MainWindow.xaml は、Window を定義した XAML になります。

```
<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="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Window タグが、WPF のウィンドウを表します。そしてタイトルに MainWindow と設定しており、高さや幅が 350 と 525 に設定されていることが確認できます。Window の中には Grid というタグが定義されています。App.xaml と同様に x:Class という属性でコードビハイン드의クラスが指定されています。HelloWorld.MainWindow クラスのコードを以下に示します。

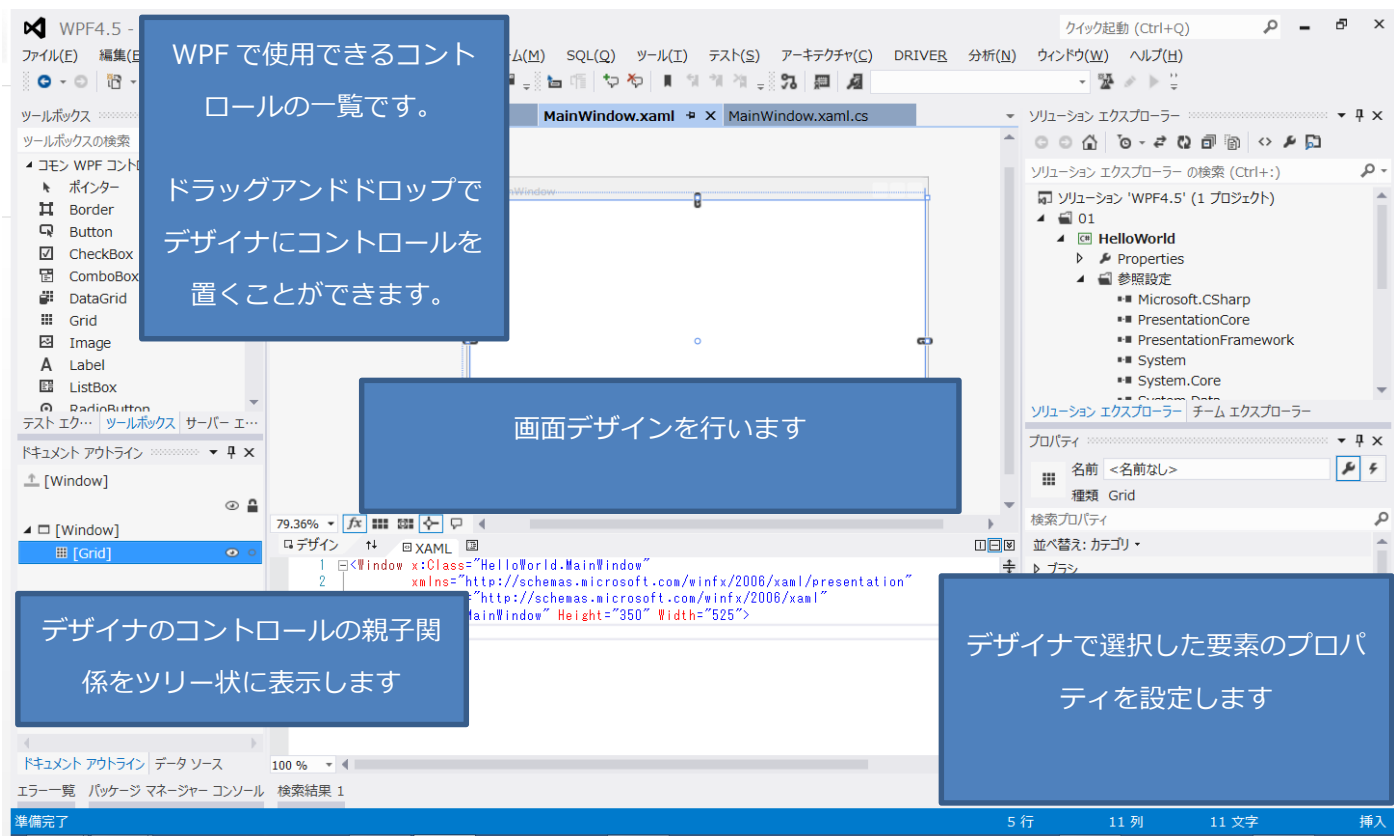

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace HelloWorld
{
    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

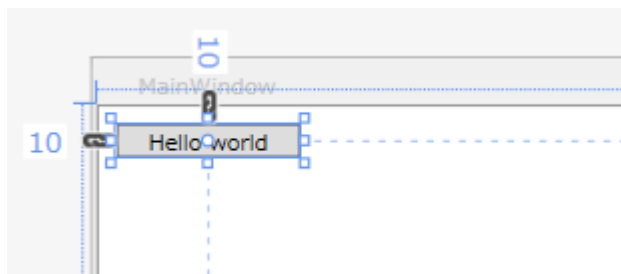
コンストラクタで呼び出されている `InitializeComponent` メソッドは、XAML で定義された情報を使用するために必須のメソッドです。このメソッドの呼び出しを忘れると、XAML で定義した情報が使用できなくなるので気を付けてください。

2.2.3. デザイナーによる画面の設計

Visual Studio 2012 には、WPF アプリケーションの画面デザインを行うためのデザイナーがついています。主にツールボックスとデザイナーとドキュメントアウトラインとプロパティを使用します。



ツールボックスからボタンを画面にドラッグアンドドロップして、位置と大きさを調整して以下のように画面に配置します。



ボタンのプロパティを以下のように設定しています。

プロパティ名	説明
x>Name	コントロールをコードビハインドから利用するための変数名。プロパティの名前か、ドキュメント アウトラインで対象をダブルクリックすることで設定可能。

	helloWorldButton と設定しています。
Content	表示文字列。Hello world と設定しています。

その他の HorizontalAlignment プロパティなどはデザイナー上などで自動的に設定されたものなので、ここでは割愛します。

2.2.4. イベントハンドラの追加とコードの記述

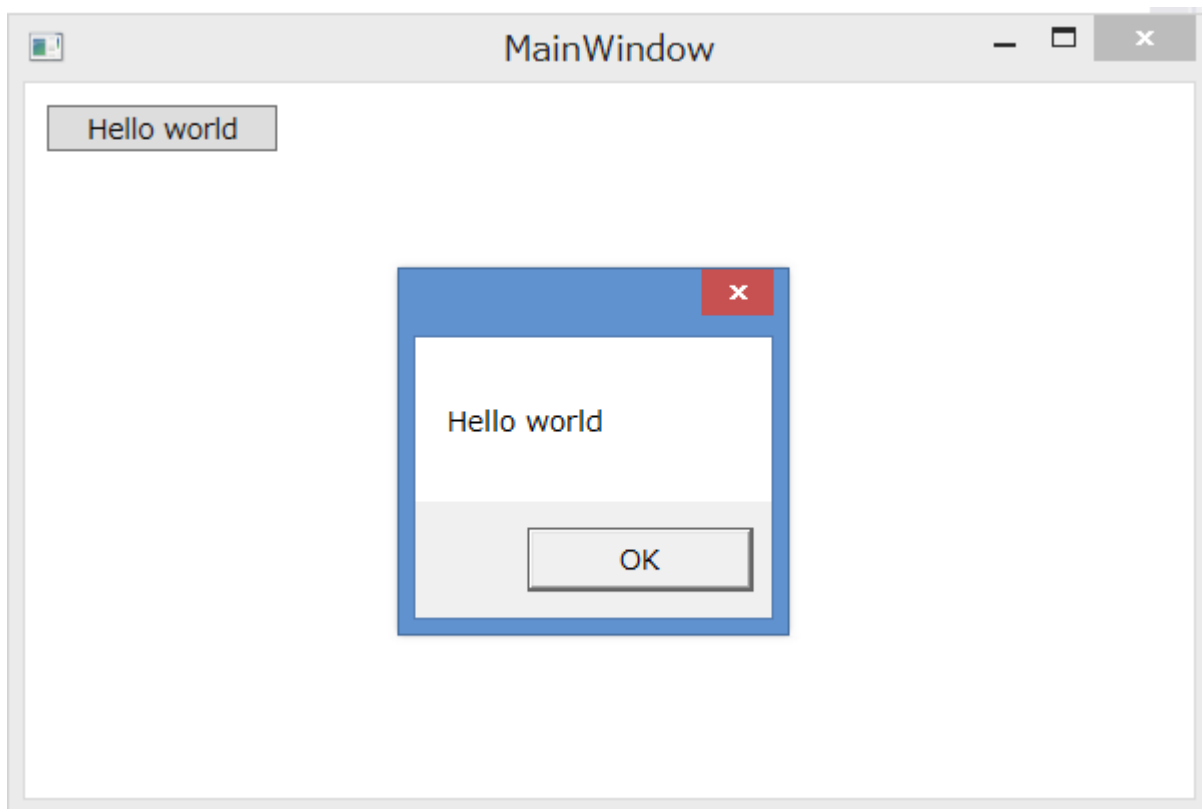
作成したボタンをダブルクリックすると、ボタンのクリックイベントが作成されます。プロパティのイベントからも Windows Form アプリと同じ要領でイベントを作成できます。helloWorldButton_Click というメソッドが作成されるので以下のようにメッセージボックスを表示するコードを追加してください。

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

    private void helloWorldButton_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("Hello world");
    }
}
```

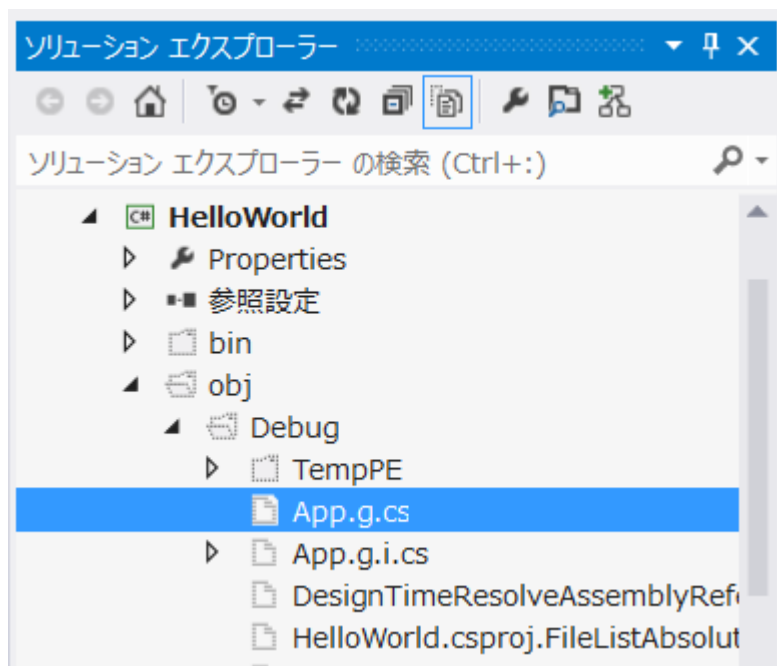
2.2.5. コンパイルして実行

アプリケーションを実行すると、以下のようにボタンの置いてある画面が表示されます。ボタンを押すとメッセージボックスが表示されます。



2.2.6. Main メソッドはどこにいった？

Hello world を作る手順の中で App クラスが WPF における Main メソッドを持つエントリーポイントのようなクラスであるという説明を行いました。これについてもう少し詳しく説明したいと思います。App.xaml と App.xaml.cs がコンパイルされる際に、以下のようなコードがコンパイラによって生成されます。このコードを見るには、ソリューションエクスプローラですべてのファイルを表示するように設定して「obj→Debug→App.g.cs」というコードを開きます。



```

namespace HelloWorld {
    /// <summary>
    /// App
    /// </summary>
    public partial class App : System.Windows.Application {
        /// <summary>
        /// InitializeComponent
        /// </summary>
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
        public void InitializeComponent() {
            #line 4 "..\¥..¥App.xaml"
            this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
            #line default
            #line hidden
        }

        /// <summary>
        /// Application Entry Point.
        /// </summary>
        [System.STAThreadAttribute()]
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
        public static void Main() {
            HelloWorld.App app = new HelloWorld.App();
            app.InitializeComponent();
            app.Run();
        }
    }
}

```

このように、WPF アプリケーションでは Main メソッドはコンパイラによって生成されています。

2.3. 全て C# で Hello world

XAML と C# を使って Hello world アプリケーションを作成しました。ここでは、この Hello world アプリケーションを C# のみで作成します。通常は、画面は XAML で記述しますし XAML で記述することを推奨します。ただ、XAML で書けることは、ほぼ全て C# で記述できます。

C# でクラス ライブラリのプロジェクトを新規作成します。ここでは CodeHelloWorld という名前で作成しました。参照設定に WPF で必要な以下の 4 つの参照を追加します。

Windows Presentation Foundation 4.5 入門

- PresentationCore
- PresentationFramework
- WindowsBase
- System.Xaml

MainWindow という名前のクラスを作成して、Window クラスを継承させます。

```
namespace CodeHelloWorld
{
    using System.Windows;

    class MainWindow : Window
    {
    }
}
```

Hello world アプリケーションで作成した画面をコードで組み立てます。InitializeComponent というメソッド内で Window 内のコントロールを組み立てています。基本的に XAML で設定している内容と 1 対 1 に対応していることが確認できます。

```
namespace CodeHelloWorld
{
    using System.Windows;
    using System.Windows.Controls;

    class MainWindow : Window
    {
        private Button helloWorldButton;

        private void InitializeComponent()
        {
            // Windowのプロパティの設定
            this.Title = "MainWindow";
            this.Height = 350;
            this.Width = 525;

            // Buttonの作成
            this.helloWorldButton = new Button
            {
                Content = "Hello world",
                HorizontalAlignment = HorizontalAlignment.Left,
                VerticalAlignment = VerticalAlignment.Top,
                Margin = new Thickness(10, 10, 0, 0),
                Width = 100
            };
            this.helloWorldButton.Click += helloWorldButton_Click;

            // Gridの作成
            var grid = new Grid();
            grid.Children.Add(this.helloWorldButton);

            // gridをWindowに設定
            this.Content = grid;
        }

        public MainWindow()
        {
            this.InitializeComponent();
        }

        private void helloWorldButton_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Hello world");
        }
    }
}
```


MainWindow クラスが出来たので App クラスを作成します。XAML を使った App クラスでは StartupUri で開始時に表示する Window の XAML の URI を指定していましたが、ここでは XAML を使っていないので MainWindow の表示を App クラスの Startup イベントで明示的に行っています。

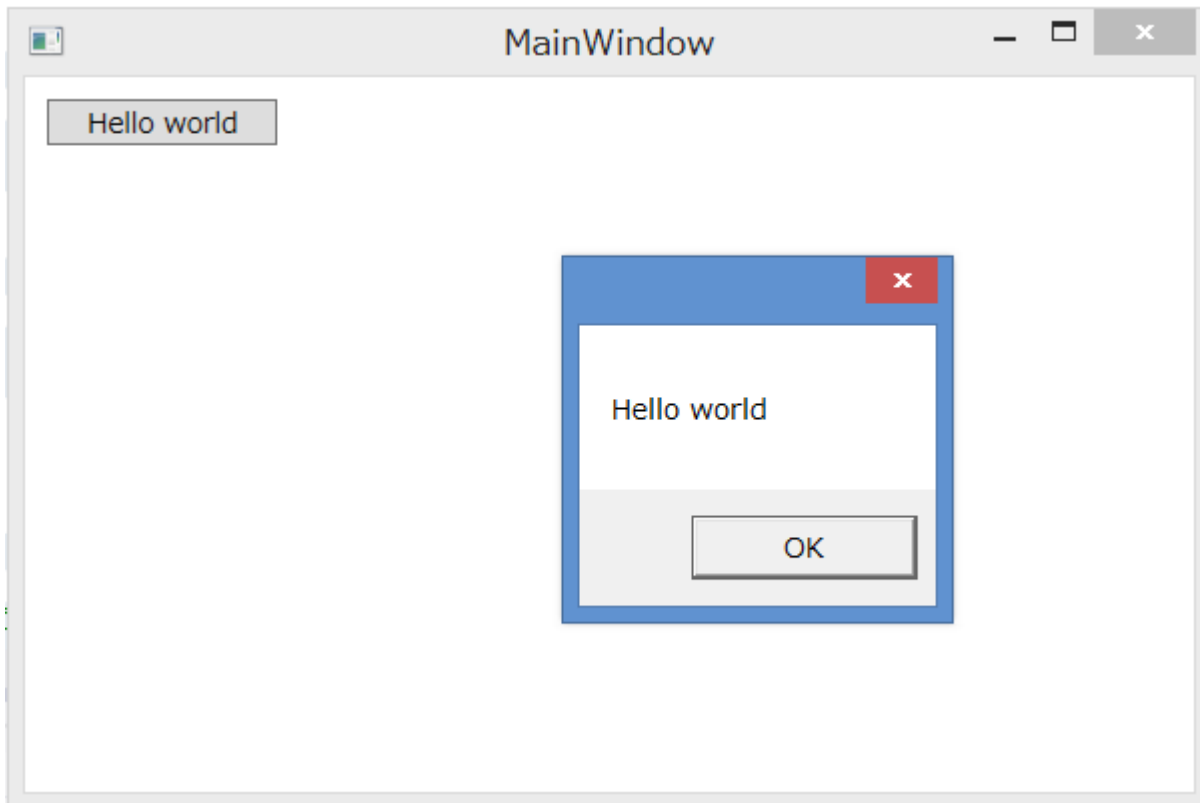
```
namespace CodeHelloWorld
{
    using System;
    using System.Windows;

    class App : Application
    {
        private void InitializeComponent()
        {
            // StartupUri は使えないので Startup イベントを使う
            this.Startup += App_Startup;
        }

        private void App_Startup(object sender, StartupEventArgs e)
        {
            // ウィンドウを作成して表示させるコードを明示的に書く
            var w = new MainWindow();
            w.Show();
        }

        [STAThread]
        public static void Main(string[] args)
        {
            // App クラスを作成して初期化して実行
            var app = new App();
            app.InitializeComponent();
            app.Run();
        }
    }
}
```

一通りのクラスが出来たので、プロジェクトのプロパティをクラスライブラリから Windows アプリケーションに変更して実行します。実行すると、XAML を使った時とおなじ見た目と動作のアプリケーションが起動します。



ここで伝えたかったのは、C#のコードでも XAML でも同じように WPF のアプリケーションが作れるという点です。そのことを知ったうえで、XAML の特性を理解し、どういつときに XAML で記述し、どういつときに C# で記述すべきなのかということを考えることが必要だということを意識して今後を読み進めてください。

2.4. WPF を構成するものを考えてみる

XAML と C# で作成するケースと、C# だけで作成するケースの Hello world を 2 つ作成しました。その過程で説明しましたが、XAML で記述できることは、ほぼ全て C# でも記述できます。その理由について WPF の構成するものを交えて説明します。

2.4.1. WPF はクラスライブラリ？

WPF = クラスライブラリというのは言い過ぎかもしれませんが、WPF は PresentationFramework と PresentationCore と WindowsBase の 3 つのアセンブリに入っているクラスから構成されています。WPF は、通常の CLR のクラスと同じように親をたどっていくと最終的に System.Object にたどり着くクラス群から構成されています。その中に、ボタンを表す Button クラスやウィンドウを表す Window クラスやボタンなどのコントロールの表示位置を決める Grid クラスなど様々なものが含まれています。基本的に、これらのクラスをインスタンス化してプロパティを設定して繋いでいくことで、WPF の画面は作成できます。

2.4.2. XAML は何？

WPF の画面を全て C# で記述できるということは、XAML は不要なのでは？ という疑問がわいてきます。確かに XAML が無くても WPF アプリケーションの作成は出来ます。XAML は、C# で記述するよりもオブジェクトのプロパティの設定や複雑なオブジェクトの組み立てを宣言的に記述できるという点で C# より優れています。

XAML は、オブジェクトのインスタンス化という領域に特化したドメイン固有言語という見方が出来ます。画面の構築は、基本的に画面を構成するオブジェクトのインスタンス化が主な仕事になります。C# で記述した Hello world の画面構築のコードをもう一度以下に示します。

```
private void InitializeComponent()
{
    // Window のプロパティの設定
    this.Title = "MainWindow";
    this.Height = 350;
    this.Width = 525;

    // Button の作成
    this.helloWorldButton = new Button
    {
        Content = "Hello world",
        HorizontalAlignment = HorizontalAlignment.Left,
        VerticalAlignment = VerticalAlignment.Top,
        Margin = new Thickness(10, 10, 0, 0),
        Width = 100
    };
    this.helloWorldButton.Click += helloWorldButton_Click;

    // Grid の作成
    var grid = new Grid();
    grid.Children.Add(this.helloWorldButton);

    // grid を Window に設定
    this.Content = grid;
}
```

プロパティの設定とオブジェクトの組み立てしか行っていません。同じ画面を構築するための XAML のコードをもう一度以下に示します。

```
<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="350" Width="525">
    <Grid>
        <Button x:Name="helloWorldButton" Content="Hello world" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" Width="100" Click="helloWorldButton_Click"/>
    </Grid>
</Window>
```

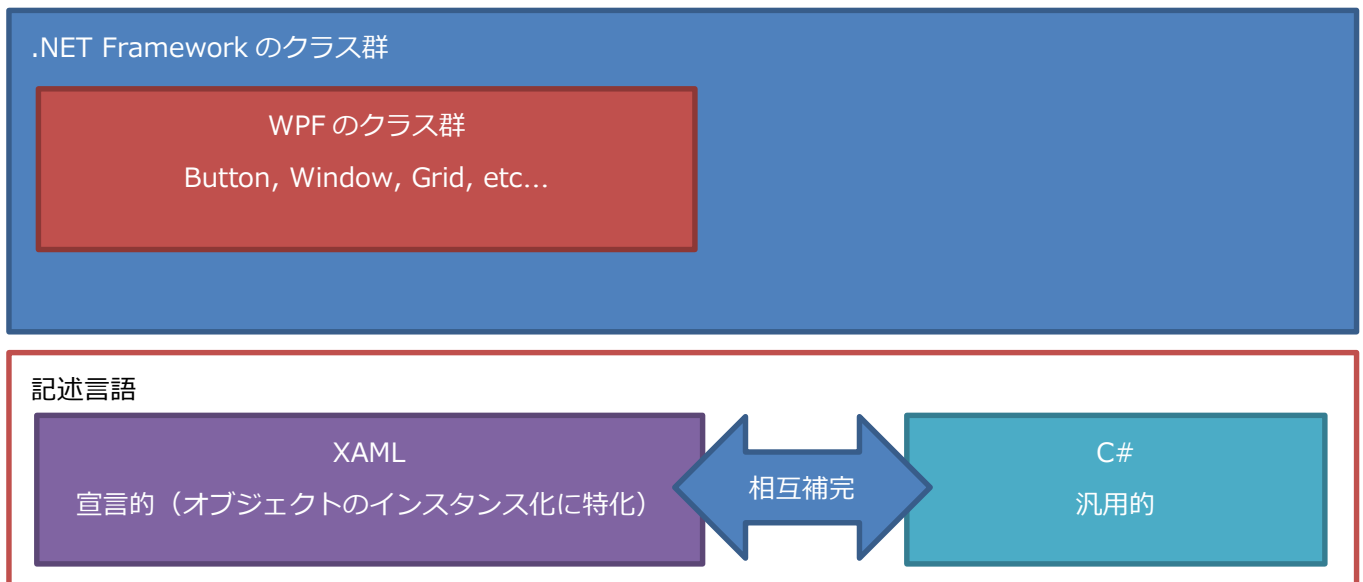
Window の下に Grid があり Grid の下に Button があるというオブジェクトの構造を端的に記述できているのはどちらになるでしょうか？個人的な主観になりますが私は XAML に軍配があがると思います。もし、画面構築に複雑な計算ロジックが含まれるようなケース（この場合も大体は WPF に用意されているレイアウトの仕組みでカバーできることが多いです）の場合は C# で記述したほうが有利かもしれません。

2.4.3. WPF を構成するもののまとめ

ということで、WPF は以下のものから構成されていることがわかりました。

- 膨大な数のクラス群
- 画面を構築するための XAML

XAML は、オブジェクトを組み立てるためのものなので、どのようなクラスが WPF にあるのかを理解し、それを XAML で組み立てるという考えで理解していくといいと思います。



2.5. WPF のコンセプト

WPF の全体像と、プログラミングモデルの説明したので、ここで WPF がどのようなコンセプトで作られているのか説明したいと思います。その後、コンセプトを実現するための重要な機能について紹介します。

WPF のコンセプトは MSDN によると「UI、メディア、およびドキュメントを組み込んだ Windows スマート クライアントの豊富なユーザー エクスペリエンスを構築するための、統一されたプログラミング モデルを開発者に提供すること」です。また、ASP.NET などに取り入れられたテンプレートを使った柔軟なレイアウトの構築や、CSS のように見た目の定義を共通化する方法などの Web アプリケーションを開発する上での優れた仕組みなども取り込まれています。

私の感想として、WPF は当時の GUI を開発するためのプラットフォームのいい点や反省点をマイクロソフトが本気で検討して実装しなおしたテクノロジーだと思います。その結果、その後の UI を開発するためのプラットフォームは Silverlight や Windows Phone や Windows ストア アプリでも WPF と同じ XAML と C# による開発が主になっています。WPF を学習するということは、マイクロソフトの提供するプラットフォーム上での UI の開発をするうえで WPF のノウハウを活用できるという点でもおすすめです。（細かい差異はたくさんありますが…）

このようなコンセプトを実現するためのキーとなる部分として、以下の 3 つをここで簡単に紹介します。

- コンテンツモデルとデータテンプレート
- スタイル
- データバインディング

ここでは、こんなことが出来るんだという雰囲気をつかんでもらうことを目的としています。詳細な説明は、後半で行っているので、そちらを参照してください。

2.5.1. コンテンツモデル

WPF では、単一の要素を表示するコントロールとして ContentControl というものが定義されています。このコントロールは、Button や Label などの多くのコントロールの親クラスです。ContentControl には Content という名前の object 型のプロパティが定義されていて、そこに設定されたクラスの型に応じて表示方法が切り替わります。表示ロジックは以下のようになっています。

- ContentTemplate にデータテンプレートが設定されている場合は、それを使って表示する。
- UIElement 型(Button や Rectangle などのコントロール)の場合はそのまま表示する。
- Content プロパティに設定された型に対してデータテンプレートが定義されている場合は、それを使って表示する。

- UIElement 型へ変換する TypeConverter がある場合は、それを使って UIElement 型に変換して表示する。
- string 型へ変換する TypeConverter がある場合は、それを使って文字列に変換して TextBlock の Text プロパティに設定して表示する。
- ToString メソッドの呼び出し結果を TextBlock の Text プロパティに設定して表示する。

ここで特に重要なのは、Content プロパティにコントロールが設定された場合は、そのまま表示できることと、その他のオブジェクトが設定された場合はデータテンプレートという仕組みを使って表示されることです。このシンプルな仕組みを理解することが WPF でデータを画面に表示する際にとっても重要になります。

UIElement を表示する例

Button コントロールを使って Content プロパティの表示を確認してみます。まずは、コントロールを設定した場合です。XAML を以下に示します。

```
<Button HorizontalAlignment="Left" VerticalAlignment="Top">
    <StackPanel Orientation="Horizontal" Margin="5">
        <TextBlock Text="button" />
        <Image Source="btn.png" Stretch="None" />
        <TextBlock Text="button" />
    </StackPanel>
</Button>
```

これは、コードでも同じように記述できます。

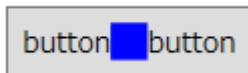
```

var panel = new StackPanel
{
    Orientation = Orientation.Horizontal,
    Margin = new Thickness(5)
};
panel.Children.Add(new TextBlock { Text = "button" });
panel.Children.Add(new Image
{
    Source = new BitmapImage(new Uri("btn.png", UriKind.Relative)),
    Stretch = Stretch.None
});
panel.Children.Add(new TextBlock { Text = "button" });

var b = new Button
{
    HorizontalAlignment = HorizontalAlignment.Left,
    VerticalAlignment = VerticalAlignment.Top,
    Content = panel
};

```

このボタンを表示すると以下ようになります。

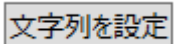


文字列の表示

ToString メソッドの呼び出し結果を表示できるので、当然文字列はそのまま表示できます。

```
<Button Content="文字列を設定" />
```

上記の XAML で以下のようなボタンが表示されます。



オブジェクトの表示

任意のクラスを設定した場合の例を示します。例えば以下のような Animal クラスがあるとします。

```
using System.Windows.Media.Imaging;

public class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }
    public BitmapImage Picture { get; set; }
}
```

このクラスを Button の Content に設定して表示してみます。コードは以下ようになります(buttonObject という名前の Button クラスの変数があると仮定)。

```
// オブジェクトを作成
var anthem = new Animal
{
    Name = "アンセム",
    Age = 9,
    Picture = new BitmapImage(new Uri("/anthem.png", UriKind.Relative))
};

// ボタンに設定
this.buttonObject.Content = anthem;
```

このボタンの表示は以下ようになります。

ContentModelSample.Animal

ToString メソッドの結果が表示されていることが確認できます。これに、Button コントロールの ContentTemplate プロパティにデータテンプレートを設定してみます。XAML を以下に示します。

```
<Button Name="buttonObject" HorizontalAlignment="Left" VerticalAlignment="Top">
    <Button.ContentTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="{Binding Name}" />
                <TextBlock Text="{Binding Age, StringFormat={} {0} 歳}" />
                <Image Source="{Binding Picture}" Stretch="None" />
            </StackPanel>
        </DataTemplate>
    </Button.ContentTemplate>
</Button>
```

テンプレートを設定すると、ToString メソッドの結果だったものが以下のような表示になります。



データテンプレートを使うと、データの表示をとて柔軟に制御できることが感じていただけると思います。

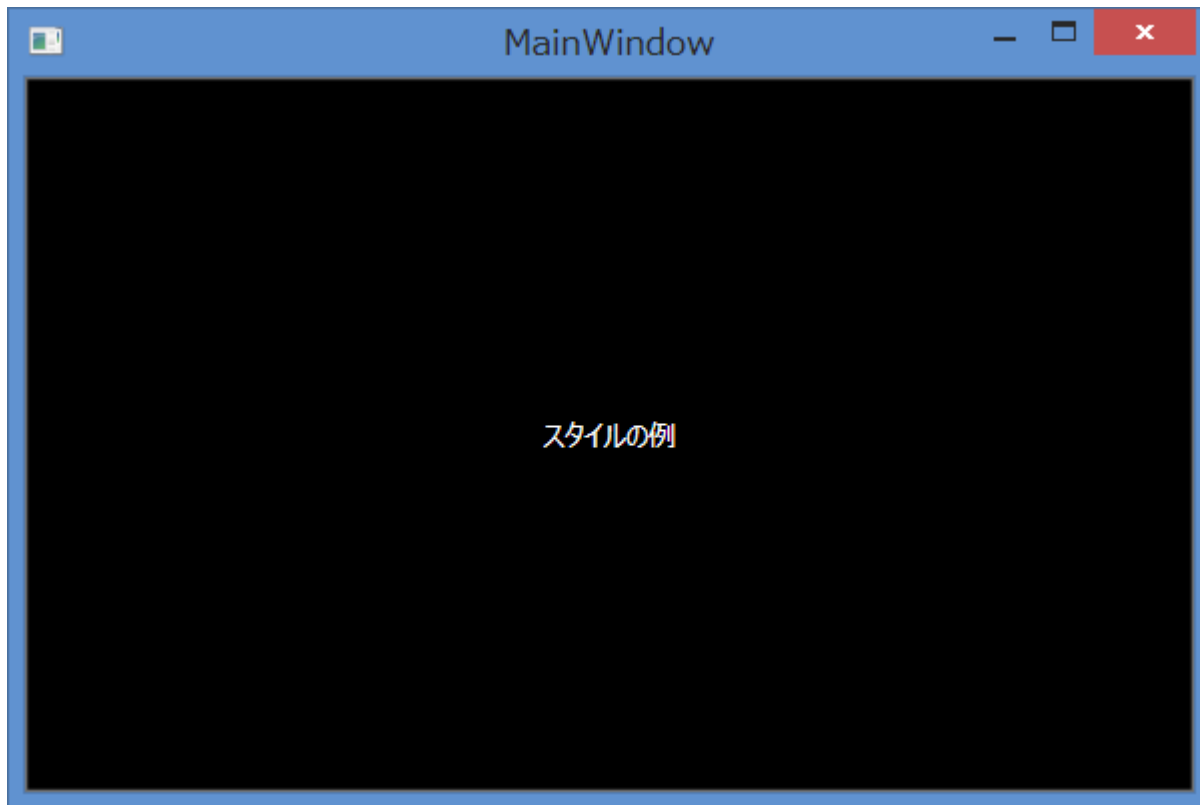
2.6. スタイル

WPF では、スタイルという仕組みを使うことでコントロールのプロパティの設定を複数のコントロールで共通化することが出来ます。こうすることで、アプリケーション全体で統一した見た目を定義することが簡単に出来るようになります。スタイルを設定するには、コントロールの Style プロパティに Style を設定します。Style は、Setter というオブジェクトを使って、どのプロパティにどんな値を設定するか指定できます。

スタイルを使って、背景色を黒、前景色を白に設定した XAML を以下に示します。

```
<Button Content="スタイルの例">
  <Button.Style>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Black" />
      <Setter Property="Foreground" Value="White" />
    </Style>
  </Button.Style>
</Button>
```

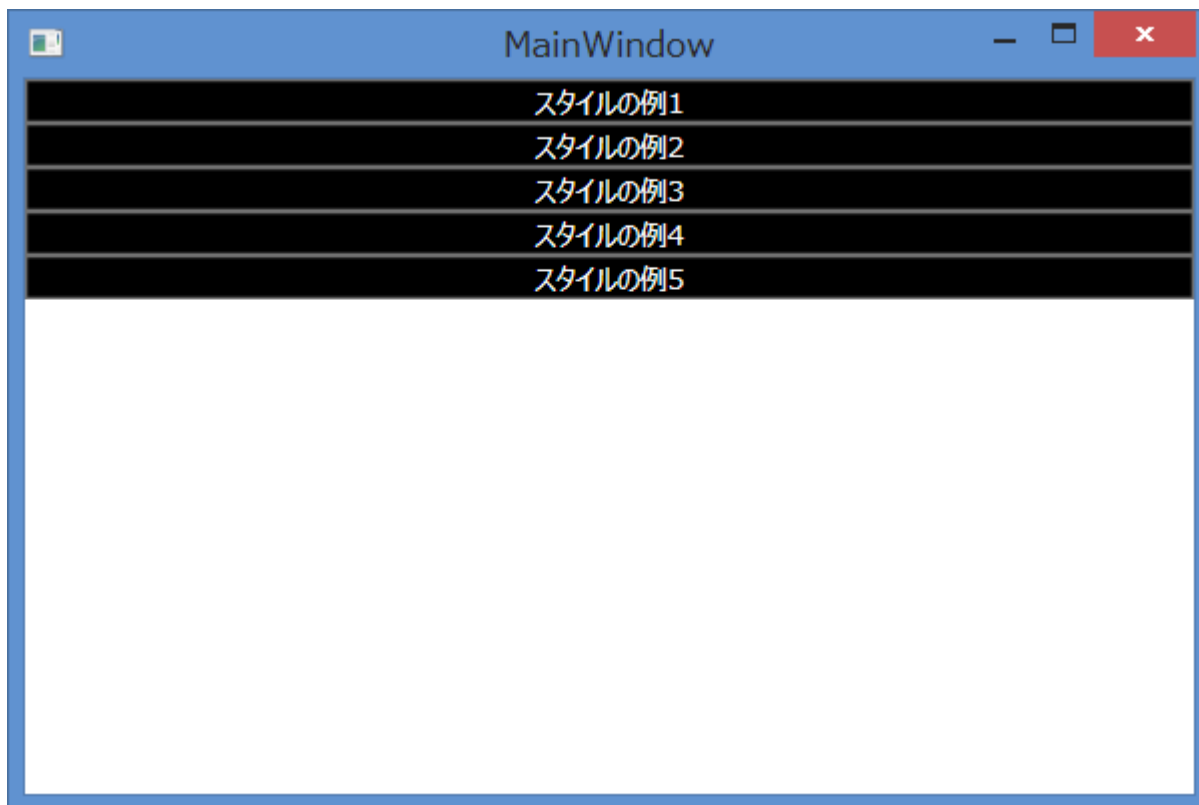
このボタンを表示すると以下ようになります。



この方法だと、1つのボタンに直接スタイルを設定しているのであまり意味はありませんが、以下のようにリソースとしてスタイルの定義を外だしにすることで複数ボタンに共通のスタイルを設定することが出来ます。XAMLを以下に示します。

```
<Window x:Class="WpfApplication4.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">
  <Window.Resources>
    <!-- スタイルをリソースに定義する -->
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Black" />
      <Setter Property="Foreground" Value="White" />
    </Style>
  </Window.Resources>
  <StackPanel>
    <!-- 画面にボタンを複数置く -->
    <Button Content="スタイルの例 1" />
    <Button Content="スタイルの例 2" />
    <Button Content="スタイルの例 3" />
    <Button Content="スタイルの例 4" />
    <Button Content="スタイルの例 5" />
  </StackPanel>
</Window>
```

先ほど Button の Style プロパティに直接設定されていた Style の定義を外に移動させています。この Window を表示すると以下ようになります。

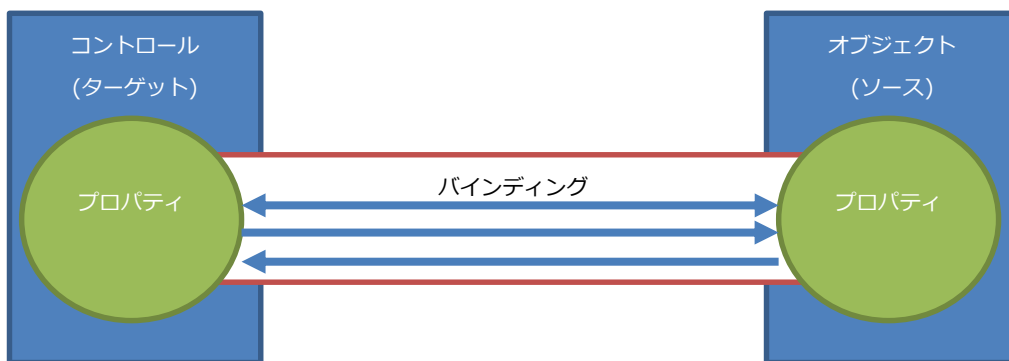


複数のコントロールにスタイルが適用されていることが確認できます。

2.7. データバインディング

最後に、データバインディングの紹介をします。データバインディングは、コントロールのプロパティと任意のオブジェクトのプロパティを同期するための仕組みです。値の同期には、双方向と一方向が指定できます。

双方向・一方向で値の同期が可能



ソースの指定方法は、いろいろな方法がありますが一番よく使うのがコントロールの DataContext プロパティにソースとなるオブジェクトを設定する方法です。DataContext プロパティは、デフォルトで親の値を継承するため Window の DataContext にソースとなるオブジェクトを設定するだけで Window 内のコントロールすべてのバインディングのソースとして設定できます。

バインディングの記述方法

バインディングは、XAML で“{”と”}”で括った記法で書くのが一般的です。この記法はマークアップ拡張といわれる XAML の記法です。Button コントロールの Content プロパティに DataContext に設定されているオブジェクトの FullName プロパティの値をバインドする XAML の例を以下に示します。

```
<Button Content="{Binding FullName}" />
```

Button を置いている Window の DataContext に以下のようにオブジェクトを設定すると Button の Content とバインドされます。

```
// DataContext に FullName プロパティを持ったオブジェクトを設定  
this.DataContext = new { FullName = “大田 一希” };
```

Window を表示させると、以下のようにバインドできていることが確認できます。

大田 一希

今回は、Button コントロールの Content プロパティとバインドしましたが TextBox コントロールのような、ユーザーの入力を受け取るコントロールとバインドすると、入力値をオブジェクトに反映することが出来ます。このように、宣言的に見た目と裏のデータを対応付け出来るため綺麗にデータと表示を分離することができます。コンテンツモデルとデータテンプレートの箇所のコード例でもバインディングを利用していましたが、データテンプレートと組み合わせることで自由自在にデータを画面に表示させることが出来るようになります。

2.8. まとめ

ここでは WPF のコンセプトと、そのコンセプトを実現するために重要だと思うコンテンツモデル・データテンプレート・スタイル・データバインディングについて簡単に紹介しました。コンテンツモデルでは、単純なデータの表示から複雑なものの表示までが、とてもシンプルに実現できることを示しました。データテンプレートでは、オブジェクトを表示するための柔軟な仕組みがあることを示しました。スタイルでは、コントロールのプロパティを共通に定義することで、一貫した見た目のアプリケーションを簡単に作れることを示しました。データバインディングでは、コントロールのプロパティと、オブジェクトのプロパティを強力に結びつけることを示しました。

これらの機能は、WPF の膨大な機能のほんの一握りにすぎませんが、これだけで従来のデスクトップアプリケーションの開発では困難だった表現や、データと見た目を綺麗に分離したアプリケーションの開発が簡単に実現できることを感じ取って頂けたと思います。ここから先は、引き続き WPF の各機能の詳細について説明していきます。

3. XAML

ここまで何度か登場してきた XAML について詳しく見て行こうと思います。XAML は、Extensible Application Markup Language の略で、MSDN によると「宣言的アプリケーション プログラミングで使用するマークアップ言語」と定義されています。XAML で主に使用する構文として以下のものがあります。

- オブジェクト要素
- XAML 名前空間
- オブジェクト要素のプロパティ
 - 属性構文
 - プロパティ要素の構文
- コレクション構文

- コンテンツ構文

1 つずつ順番に見ていきます。

3.1. オブジェクト要素と XAML 名前空間

オブジェクト要素とは、名前の通りクラス名に対応するものです。XML のタグに対応します。Hello world アプリケーションでは<Window />といったタグや<Grid />といったタグがこれにあたります。XAML 名前空間は CLR の名前空間のようなもので、そのタグに紐づくクラスがどの名前空間にいるのかを定義します。WPF では規定の名前空間として xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" が割り当てられています。その他に xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" という名前空間も割り当てられています。前者が WPF の基本的なクラスが所属するの名前空間にマッピングされていて、後者が XAML 言語組み込みの機能を提供します。

XAML 名前空間は、規定の名前空間の他に自分で作成したクラスに割り当てることができます。そのときの名前空間の書式は xmlns:プレフィックス="clr-namespace:名前空間;assembly=アセンブリ名" になります。例として、CustomXaml という名前のコンソールアプリケーションを作成して、WPF に必要なアセンブリを追加したプロジェクトに以下のような Person という名前のクラスを作成します。

```
namespace CustomXaml
{
    using System;

    public class Person
    {
        public Person()
        {
            this.Birthday = DateTime.Now;
        }

        public DateTime Birthday { get; private set; }
    }
}
```

このクラスを XAML 名前空間とオブジェクト要素を使って定義した XAML は以下のようになります。

```
<p:Person xmlns:p="clr-namespace:CustomXaml;assembly=CustomXaml" />
```

この XAML を Person.xaml という名前で作成して、埋め込まれたリソースに設定して読み込んで Person クラスのインスタンスを取得してみます。XAML を読み込むには、System.Windows.Markup.XamlReader クラスを使います。Load メソッドに Stream を渡すと XAML をパースした結果のオブジェクトを返します。今回の Person.xaml は、Person クラスを定義しているので、Person クラスのインスタンスが返ってくるはずです。読み込むコードを以下に示します。

```
namespace CustomXaml
{
    using System;
    using System.Windows.Markup;

    class Program
    {
        public static void Main(string[] args)
        {
            // アセンブリから対象の XAML のストリームを取得
            var s = typeof(Program).Assembly.GetManifestResourceStream("CustomXaml.Person.xaml");
            // パース
            var p = XamlReader.Load(s) as Person;
            // プロパティを表示してみる
            Console.WriteLine("p.Birthday = {0}", p.Birthday);
        }
    }
}
```

このプログラムを実行すると以下のように表示されます。（表示結果は実行した時間によってかわります）

```
p.Birthday = 2013/01/03 13:23:04
```

XAML をパースすることで、Person クラスのインスタンスが作られていることがわかります。このことから、XAML のオブジェクト要素がクラス名に、XAML 名前空間がクラスの名前空間に対応するということがわかると思います。

3.2. オブジェクト要素のプロパティ

XAML でオブジェクト要素を使ってオブジェクトを構築することができることがわかりました。次に、オブジェクト要素のプロパティの設定方法を紹介します。オブジェクト要素のプロパティの設定方法は「属性構文」と「プロパティ要素の構文」の 2 通りがあります。まずは属性構文について説明します。

属性構文は、名前の通り XML の属性としてプロパティを定義する方法です。例えば、先ほどの Person クラスに FullName と Salary という 2 つのプロパティを以下のように追加したとします。

```
namespace CustomXaml
{
    using System;

    public class Person
    {
        public Person()
        {
            this.Birthday = DateTime.Now;
        }

        public DateTime Birthday { get; private set; }

        // 名前と給料を追加
        public string FullName { get; set; }
        public int Salary { get; set; }
    }
}
```

この 2 つのプロパティを属性構文を使って指定すると以下ようになります。

```
<p:Person xmlns:p="clr-namespace:CustomXaml;assembly=CustomXaml"
    FullName="田中 太郎"
    Salary="300000"/>
```

XAML を読み込んで表示するプログラムに追記をして FullName と Salary も表示するように変更します。

```

namespace CustomXaml
{
    using System;
    using System.Windows.Markup;

    class Program
    {
        public static void Main(string[] args)
        {
            // アセンブリから対象の XAML のストリームを取得
            var s = typeof(Program).Assembly.GetManifestResourceStream("CustomXaml.Person.xaml");
            // パース
            var p = XamlReader.Load(s) as Person;
            // プロパティを表示してみる
            Console.WriteLine("p.FullName = {0}, p.Salary = {1}, p.Birthday = {2}",
                p.FullName,
                p.Salary,
                p.Birthday);
        }
    }
}

```

実行すると、以下のような結果が表示されます。

```
p.FullName = 田中 太郎, p.Salary = 300000, p.Birthday = 2013/01/03 15:53:04
```

続行するには何かキーを押してください ...

もう 1 つの「プロパティ要素の構文」では、プロパティを特殊な命名規約に従ったタグとして記述できます。

FullName と Salary を「プロパティ要素の構文」で設定した XAML を以下に示します。

```

<p:Person xmlns:p="clr-namespace:CustomXaml;assembly=CustomXaml">
    <p:Person.FullName>
        田中 太郎
    </p:Person.FullName>
    <p:Person.Salary>
        300000
    </p:Person.Salary>
</p:Person>

```

このように「プロパティ要素の構文」ではプロパティを<クラス名.プロパティ名>という命名規約のタグとして指定できます。今回の例のように単純なものではメリットは出ないですが、プロパティの型がオブジェクトの場合、この記法が役に立ちます。例えば Person クラスに Father、Mother という Person 型のプロパティがあった場合「属性

構文」では XAML で Father と Mother に値を設定することができません。「プロパティ要素の構文」があることで、XAML で複雑なオブジェクトを組み立てることが出来るようになっています。

Person 型の Father と Mother を追加した Person クラスの定義を以下に示します。

```
namespace CustomXaml
{
    using System;

    public class Person
    {
        public Person()
        {
            this.Birthday = DateTime.Now;
        }

        public DateTime Birthday { get; private set; }

        // 名前と給料を追加
        public string FullName { get; set; }
        public int Salary { get; set; }

        // 父親と母親
        public Person Father { get; set; }
        public Person Mother { get; set; }
    }
}
```

「オブジェクト要素の構文」を使って Father と Mother を設定している XAML を以下に示します。

```
<p:Person xmlns:p="clr-namespace:CustomXaml;assembly=CustomXaml"
    FullName="田中 太郎"
    Salary="300000">
    <!-- オブジェクト要素の構文でプロパティにオブジェクトを設定している例 -->
    <p:Person.Father>
        <p:Person FullName="田中 父" />
    </p:Person.Father>
    <p:Person.Mother>
        <p:Person FullName="田中 母" />
    </p:Person.Mother>
</p:Person>
```

この XAML を読み込んで父親と母親の名前を表示するプログラムを以下に示します。

```

namespace CustomXaml
{
    using System;
    using System.Windows.Markup;

    class Program
    {
        public static void Main(string[] args)
        {
            // アセンブリから対象の XAML のストリームを取得
            var s = typeof(Program).Assembly.GetManifestResourceStream("CustomXaml.Person.xaml");
            // パース
            var p = XamlReader.Load(s) as Person;
            // プロパティを表示してみる
            Console.WriteLine("FullName = {0}, Father.FullName = {1}, Mother.FullName = {2}",
                p.FullName,
                p.Father.FullName,
                p.Mother.FullName);
        }
    }
}

```

実行すると、以下のように Father プロパティと Mother プロパティに値が設定できていることが確認できます。

```
FullName = 田中 太郎, Father.FullName = 田中 父, Mother.FullName = 田中 母
```

続行するには何かキーを押してください ...

3.3. コレクション構文

XAML では、コレクションのプロパティを簡単に記述するためのコレクション構文が用意されています。具体的には、コレクションのプロパティを設定する際に、コレクション型を明に指定せずに、コレクションの要素を複数指定します。具体例を以下に示します。

コレクション型のプロパティの Children プロパティを持った Item という型を定義します。

```

namespace CollectionXaml
{
    using System.Collections.Generic;
    using System.Collections.ObjectModel;

    public class Item
    {
        public Item()
        {
            this.Children = new ItemCollection();
        }

        public string Id { get; set; }
        // コレクション型のプロパティ
        public ItemCollection Children { get; set; }
    }

    public class ItemCollection : Collection<Item> { }
}

```

Children プロパティに要素を 3 つ設定した XAML は以下ようになります。

```

<Item xmlns="clr-namespace:CollectionXaml;assembly=CollectionXaml"
    Id="item1">
    <Item.Children>
        <!--<ItemCollection>-->
            <Item Id="item1-1" />
            <Item Id="item1-2" />
            <Item Id="item1-3" />
        <!--</ItemCollection>-->
    </Item.Children>
</Item>

```

ここでコメントアウトしているように<Item.Children>～</Item.Children>の設定の箇所で<ItemCollection>の設定を省略できる点がコレクション構文の優れたところです。このように省略すると、XAML をパースする段階で自動的に Children プロパティからコレクションが取得され Item が追加されます。明示的に<ItemCollection>タグを使ってコレクションを設定しても、XAML をパースして得られるものは同じですが、この場合は Children プロパティの書き込みが許可されていなければいけません。

この XAML を読み込んで Item の内容を表示するプログラムを以下に示します。

```

namespace CollectionXaml
{
    using System;
    using System.Windows.Markup;

    class Program
    {
        static void Main(string[] args)
        {
            var s = typeof(Program).Assembly.GetManifestResourceStream("CollectionXaml.Item.xaml");
            var item = XamlReader.Load(s) as Item;

            // Itemの内容を表示
            Console.WriteLine(item.Id);
            foreach (var i in item.Children)
            {
                Console.WriteLine("  {0}", i.Id);
            }
        }
    }
}

```

このコードを実行すると以下のように表示されます。

```

item1

    item1-1

    item1-2

    item1-3

```

コレクション構文によって、設定した項目が取得できていることが確認できます。

3.4. コンテンツ構文

XAML では、基本的にプロパティを明示して、値を設定しますが XAML でマークアップするクラスに `System.Windows.Markup.ContentPropertyAttribute` 属性でコンテンツプロパティが指定されている場合に限りプロパティ名を省略して書くことが出来ます。例えば WPF のボタンなどでは `Content` という名前のプロパティがコンテンツプロパティとして指定されているので以下のように、プロパティ名を省略して XAML を記述できます。

```

<!-- 省略したもの -->
<Button>Hello world</Button>

<!-- 省略してない書き方 -->
<Button>
    <Button.Content>Hello world</Button.Content>
</Button>

```

例えば、コレクション構文の例で示した Item クラスの Children プロパティをコンテンツプロパティとして設定するには以下のように Item クラスに属性を設定します。

```

namespace CollectionXaml
{
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Windows.Markup;

    // Children プロパティをコンテンツプロパティとして指定
    [ContentProperty("Children")]
    public class Item
    {
        public Item()
        {
            this.Children = new ItemCollection();
        }

        public string Id { get; set; }
        public ItemCollection Children { get; set; }
    }

    public class ItemCollection : Collection<Item> { }
}

```

このようにすると、Children プロパティの指定を省略できるようになるので、XAML が以下のように簡潔になります。

```

<Item xmlns="clr-namespace:CollectionXaml;assembly=CollectionXaml"
    Id="item1">
    <Item Id="item1-1" />
    <Item Id="item1-2" />
    <Item Id="item1-3" />
</Item>

```

XAML を読み込んで表示するコードと、実行結果はコレクション構文で示した内容と同じため省略します。

3.5. マークアップ拡張

XAML は、XML をベースにして作られた言語なので、複雑な構造をもったオブジェクトでもタグを入れ子にしていくことで柔軟に定義できます。しかし、XML は書く人にとっては冗長でちょっとした内容でも記述量が跳ね上がるといった問題点もあります。しかも、それがよく書くものだったときには少しくんざりしてしまいます。XAML では、マークアップ拡張という機能を使うことによって、本来は大量の XML を書かなければいけないところを簡潔に記述できるようにする機能が提供されています。また、マークアップ拡張を使って XML で記述できないような値を取得して設定することもできます。

マークアップ拡張は、XAML の属性の値を指定するときに{ではじまり}で終わる形で記述します。こうすることで、System.Windows.Markup.MarkupExtension から継承したクラスに値の生成を委譲することが出来ます。WPF では組み込みで{Binding Path=...}や{StaticResource ...}や{DynamicResource ...}など様々な種類のマークアップ拡張が定義されています。これらを使うことで、XAML の記述を簡潔に行ったり、プロパティに設定する値を特殊な方法で取得することが出来るようになります。

例えば、以下のような Item という名前のクラスの Id というプロパティに XAML から毎回ユニークになるような値を設定しないといけないような場合に、マークアップ拡張を使うことができます。マークアップ拡張と Item クラスのコードを以下に示します。


```

namespace CollectionXaml
{
    using System;
    using System.Windows.Markup;

    public class Item
    {
        public string Id { get; set; }
    }

    // Idを提供するマークアップ拡張
    public class IdProviderExtension : MarkupExtension
    {
        // Idのプリフィックス
        public string Prefix { get; set; }

        // 値を提供するロジックを記述する
        public override object ProvideValue(System.IServiceProvider serviceProvider)
        {
            return Prefix + Guid.NewGuid().ToString();
        }
    }
}

```

IdProviderExtension というクラスがマークアップ拡張のクラスになります。{IdProvider Prefix=hoge}のように使います。実際に Item クラスの Id プロパティに指定した XAML は以下のようになります。

```

<Item xmlns="clr-namespace:MarkupExtensionSample;assembly=MarkupExtensionSample"
    Id="{IdProvider Prefix=item-}" />

```

この XAML を 2 回読み込んで Id の値を表示してみます。

```

namespace MarkupExtensionSample
{
    using System;
    using System.Windows.Markup;

    class Program
    {
        static void Main(string[] args)
        {
            // XAML を読み込んで Id を表示
            var item = XamlReader.Load(
                typeof(Program).Assembly.GetManifestResourceStream("MarkupExtensionSample.Item.xaml")) as
s Item;

            Console.WriteLine(item.Id);

            // 再度 XAML を読み込んで Id を表示
            var item2 = XamlReader.Load(
                typeof(Program).Assembly.GetManifestResourceStream("MarkupExtensionSample.Item.xaml")) as
s Item;

            Console.WriteLine(item2.Id);
        }
    }
}

```

実行すると、Id の値が毎回ことになっていることが確認できます。

```
item-574cb4ed-4ec4-46ce-9e99-76f09b7545b7
```

```
item-d56c2d5e-f608-4ccc-8702-dffa18f366a9
```

3.5.1. ProvideValue メソッドの IServiceProvider って何者？

簡単なマークアップ拡張を作るときには利用しませんが、WPF が提供している Binding や StaticResource などのような 1 つのマークアップ拡張に閉じた範囲で値の設定ができないような複雑なマークアップ拡張を作るときには ProvideValue メソッドの引数の IServiceProvider を使用します。IServiceProvider の GetService メソッドを使って、様々な関連情報にアクセスできるクラスが取得できます。どのようなクラスが取得できるかは、MSDN を参照してください。

MarkupExtension.ProvideValue メソッド

<http://msdn.microsoft.com/ja-jp/library/system.windows.markup.markupextension.providevalue.aspx>

3.6. 添付プロパティ

添付プロパティは「型名.プロパティ名」といった形で指定するプロパティです。通常のプロパティと異なる点は、プロパティを設定する型と添付プロパティを指定するときの型名が必ずしも一致しなくても良いという点です。添付プロパティは、主に親要素が子要素に追加で情報を与えるのに利用されます。

具体例としては、コントロールの要素を配置するためのレイアウトの情報があげられます。たとえばボタンを左から 10px, 上から 20px の場所に置くといった場合にボタンに Left や Top というプロパティを付けるというのが一般的な考えになると思います。しかし、これでは絶対座標以外のレイアウトの指定方法が出てきた場合にボタンのプロパティが爆発的に増えてしまうといった問題があります。このようなときに Left や Top といった情報は上位のパネルの添付プロパティとして定義して、ボタンとは定義は切り離しつつもボタンに直接値を設定できるといったことを実現しています。添付プロパティは、XAML からコードに展開される際に「クラス名.Set 添付プロパティ名(オブジェクト, 値)」という形に展開されます。値の取得には「クラス名.Get 添付プロパティ名(オブジェクト)」の形式のメソッドを使用します。

添付プロパティについては、WPF のコントロールの基底クラスを説明する箇所で再度取り上げます。

3.7. 添付イベント

添付イベントも WPF のコントロールの規定クラスと強く紐づいている概念のため、ここでは詳細は割愛します。添付プロパティと同じように、実際にそのオブジェクトに定義されていないイベントハンドラを、設定する記法になります。

3.8. TypeConverter

XAML では、プロパティの指定に属性構文を使うと値は必ず文字列で指定します。しかし、XAML では int 型などの数字や、enum や、その他のクラス型のプロパティに対しても文字列で指定することが出来ます。これは、型コンバータという仕組みが間に入って型変換を行っているためです。普段、特別意識する必要はありませんが、文字列を指定するだけでオブジェクトを指定できるケースがある場合は、この仕組みのおかげだと頭の片隅に入れておいてください。

また、ここでは詳細に説明しませんが、独自の型コンバータを定義することもできます。独自の型を定義して XAML から簡単に文字列で指定するのが適切だと思った場合には以下の MSDN を参考にして実装を検討してみてください。

TypeConverters および XAML

<http://msdn.microsoft.com/ja-jp/library/aa970913.aspx>

3.9. その他の機能

XAML には、その他に様々な機能が定義されています。ここでは詳細は説明しませんが、今後の説明の中で出てきた段階で必要に応じて説明をしたいと思います。以下に、個人的に目を通しておいた方がよいと思うものについて MSDN のリンクを列挙します。

- XAML 名前空間 (x:) 言語機能
<http://msdn.microsoft.com/ja-jp/library/ms753327.aspx>
- WPF XAML 拡張機能 （定義されているマークアップ拡張の一覧）
<http://msdn.microsoft.com/ja-jp/library/ms747180.aspx>
- mc:Ignorable 属性
<http://msdn.microsoft.com/ja-jp/library/aa350024.aspx>

4. WPF のコントロール

WPF には、アプリケーションを構築するために必要な様々なコントロールが定義されています。MSDN のカテゴリ別のコントロールのページには以下のように分類されています。

カテゴリ別のコントロール

<http://msdn.microsoft.com/ja-jp/library/ms754204.aspx>

- レイアウト
- ボタン
- データ表示
- 日付表示および選択
- メニュー
- Selection
- Navigation
- ダイアログ ボックス
- ユーザー情報
- ドキュメント
- 入力

- メディア
- デジタル インク

ここでは、カテゴリごとにコントロールの簡単な使用方法について紹介します。各コントロールの完全なプロパティやメソッドのリストについては MSDN ライブラリを参照してください。

4.1. レイアウト

レイアウトは、配下にコントロールを 1 つ以上持ちコントロールのレイアウトを決めるコントロールのことです。代表的なものとして StackPanel や DockPanel、Grid などがあります。ここでは、いくつかのコントロールをピックアップして紹介します。

4.1.1. Border コントロール

Border コントロールは、子の周囲に境界線や背景を表示するコントロールです。主に以下のプロパティを設定して利用します。

プロパティ	説明
Thickness <code>BorderThickness { get; set; }</code>	境界の上下左右の幅を指定します。XAML では”左, 上, 右, 下”のカンマ区切りの文字列で指定できます。省略時には”10”のように 1 つ指定するだけで上下左右を全て 10 に設定できます。
Brush <code>BorderBrush { get; set; }</code>	境界を塗りつぶすためのブラシを指定します。XAML では単色の場合は Red や Blue などの色の名前で指定できます。
CornerRadius <code>CornerRadius { get; set; }</code>	角の丸みを指定できます。XAML では”左上, 右上, 右下, 左下”のカンマ区切りの文字列で指定します。
Brush <code>Background { get; set; }</code>	背景を塗りつぶすためのブラシを指定します。
Thickness <code>Padding { get; set; }</code>	境界の中と境界の間の余白を指定します。

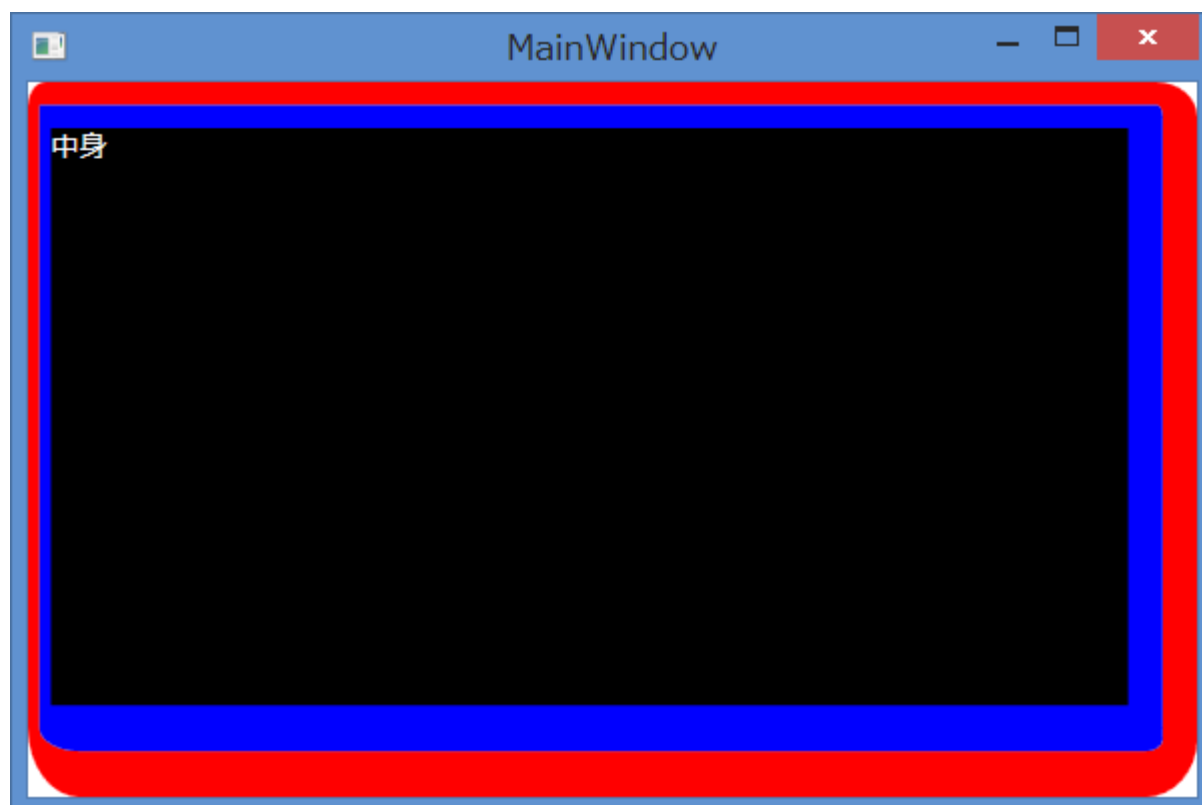
UIElement Child { get; set; }

子要素を指定します。UIElement は、WPF の画面における要素の先祖のクラスになります。このプロパティはコンテンツプロパティになります。

Border を置いた Window の XAML の例を以下に示します。

```
<Window x:Class="BorderSample.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">
    <Border
        Padding="5, 10, 15, 20"
        BorderThickness="5, 10, 15, 20"
        BorderBrush="Red"
        CornerRadius="5, 10, 15, 20"
        Background="Blue">
        <TextBlock Text="中身" Background="Black" Foreground="White" />
    </Border>
</Window>
```

Border 内の余白と Border の線と角の丸みに 5, 10, 15, 20 という値を設定しています。そして Border の色を赤に、背景を青に設定しています。Borderの中には、黒背景に白文字のテキストを配置しています。この Window を表示すると以下のようになります。



枠線の太さや Border 内の余白のとり方などを確認することで、どのようにプロパティの設定が実際の表示に反映されているか確認できます。実際に、これらのプロパティの値をいじってみて、見た目にどのように反映されるのか試してみてください。Border コントロールは、実際のアプリケーションでもコントロールを分類するための枠を作るためによく使うコントロールなので、しっかり挙動を押さえておきましょう。

4.1.2. BulletDecorator コントロール

BulletDecorator コントロールは、行頭の要素と、子要素を表示するコントロールです。正直あまり使うことはないと思います。主に以下のコントロールを設定します。

プロパティ	説明
Brush Background { get; set; }	背景色を指定します。
UIElement Bullet { get; set; }	行頭に表示する要素を指定します。
UIElement Child { get; set; }	子要素を指定します。このプロパティはコンテンツプロパティです。

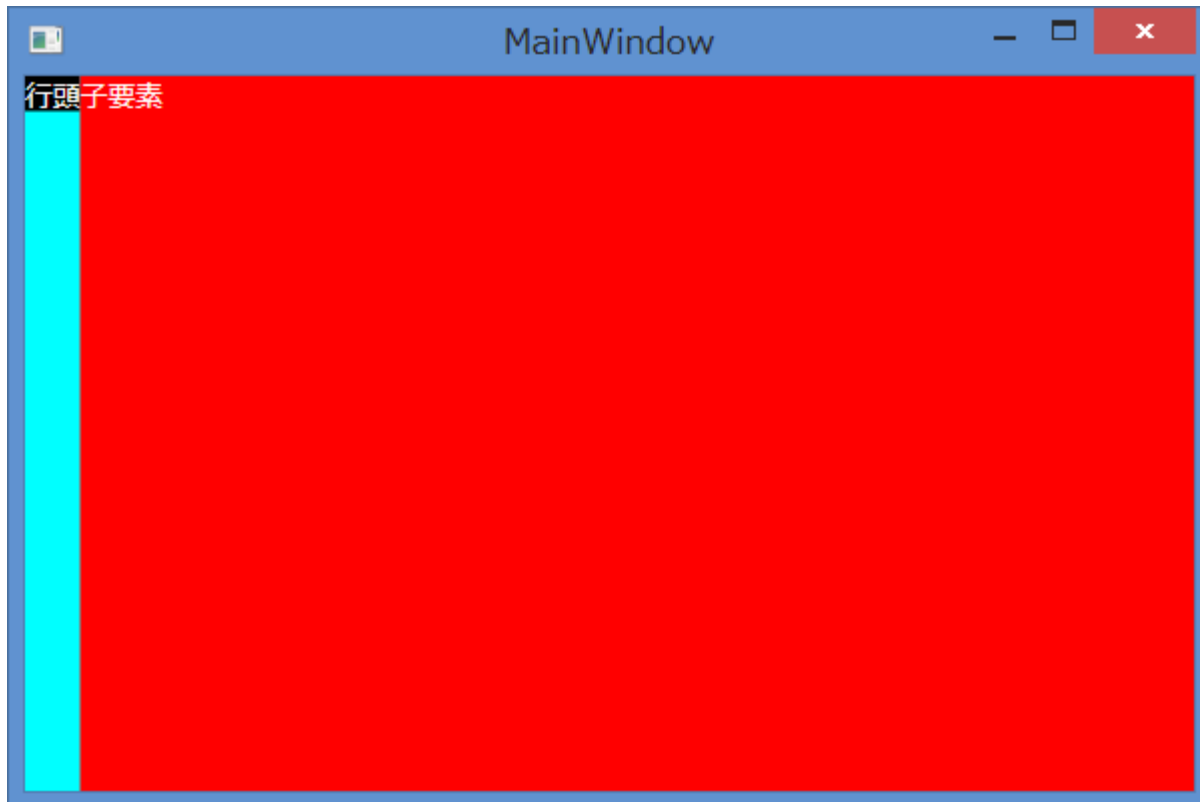
このコントロールを使用した XAML を以下に示します。

```

<Window x:Class="BulletDecoratorSample.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">
  <BulletDecorator Background="Cyan">
    <BulletDecorator.Bullet>
      <TextBlock Text="行頭" Foreground="White" Background="Black" />
    </BulletDecorator.Bullet>
    <TextBlock Text="子要素" Foreground="White" Background="Red" />
  </BulletDecorator>
</Window>

```

Bullet に行頭というテキストを、子要素に子要素というテキストを設定しています。この Window を表示すると以下ようになります。



4.1.3. Canvas コントロール

Canvas コントロールは、子要素を Canvas の中に絶対座標指定で配置できるコントロールです。これまで紹介してきた Border コントロールや BulletDecorator コントロールとは異なり、Canvas コントロールは、配下に子要素を複数持つことができます。

Canvas コントロールで使用するプロパティは以下のようなものがあります。

プロパティ	説明
Bottom 添付プロパティ	Canvas の下を起点として位置を指定します。
Left 添付プロパティ	Canvas の左を起点として位置を指定します。
Right 添付プロパティ	Canvas の右を起点として位置を指定します。
Top 添付プロパティ	Canvas の上を起点として位置を指定します。

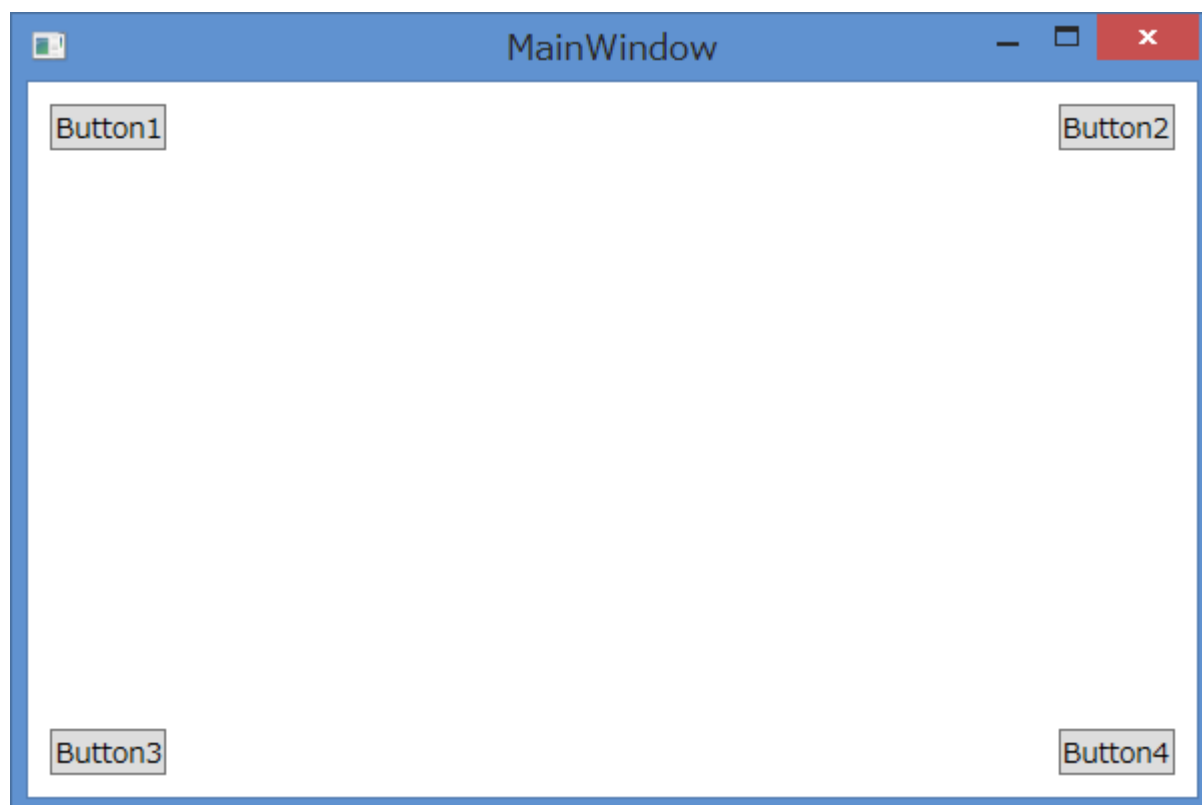
上記の添付プロパティで Top と Bottom、Left と Right が両方設定された場合は Top と Left が優先されます。

Canvas コントロールの使用例を以下に示します。

```
<Window x:Class="CanvasSample.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">
    <Canvas>
        <Button Canvas.Top="10" Canvas.Left="10" Content="Button1" />
        <Button Canvas.Top="10" Canvas.Right="10" Content="Button2" />
        <Button Canvas.Bottom="10" Canvas.Left="10" Content="Button3" />
        <Button Canvas.Bottom="10" Canvas.Right="10" Content="Button4" />
    </Canvas>
</Window>
```

Top と Left と Right と Bottom は添付プロパティなので、XAML で説明したようにクラス名.プロパティ名の形で設定を行います。このように、親要素（この場合 Canvas）に対して子要素（この場合 Button）が何かしら情報を提供するために使用します。WPF では、今回の例のようにレイアウト情報で使われることが多いです。

この Window を表示すると以下のようになります。



4.1.4. StackPanel コントロール

StackPanel コントロールは、子要素を縦方向または横方向に一系列に並べるコントロールです。StackPanel の表示領域からあふれたコントロールは表示されません。StackPanel が子コントロールを並べる際に子コントロールに、左端・右端・上端・下端・中央・領域全体に表示するかを委ねます。デフォルトでは、表示可能な領域いっぱい子コントロールを配置します。

StackPanel で使用するプロパティには以下のようなものがあります。

プロパティ	説明
Orientation <code>Orientation { get; set; }</code>	子要素を縦並びにするか、横並びにするか設定します。横並びのときは Horizontal、縦並びのときは Vertical を設定します。デフォルトは Vertical です。

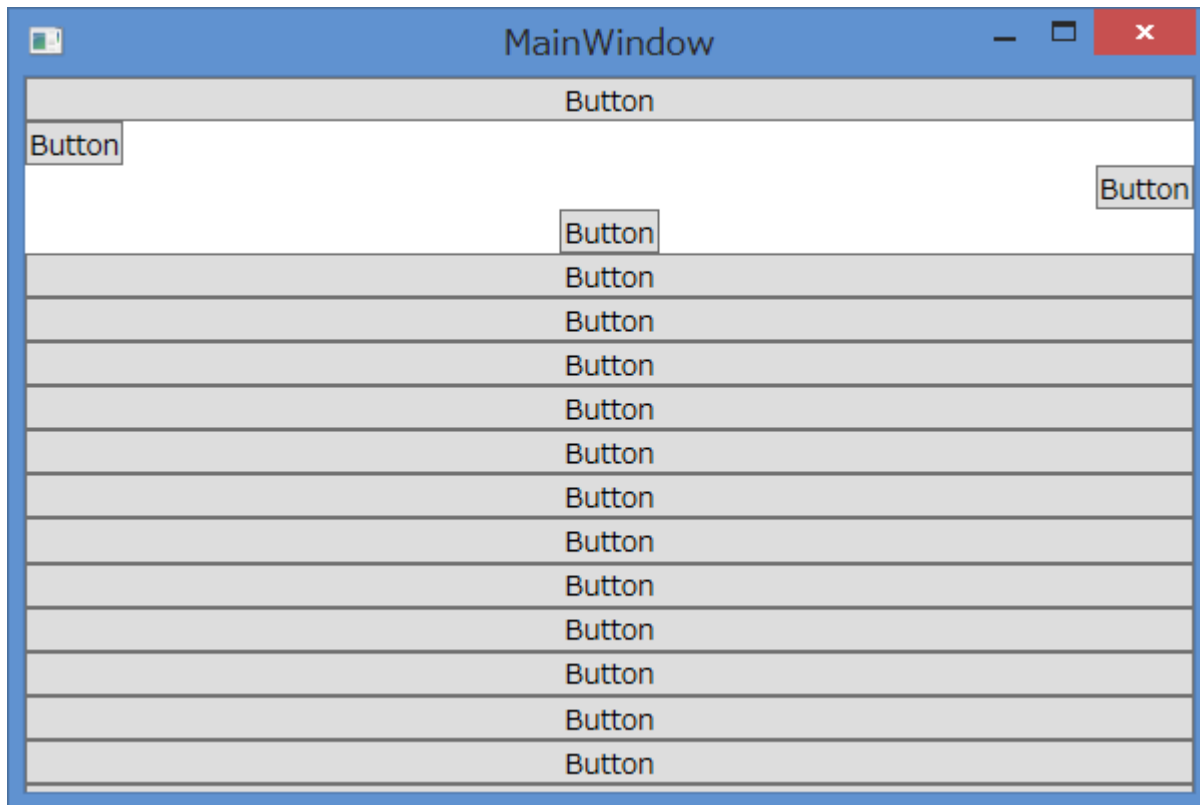
縦方向にコントロールを表示する場合の XAML の例は以下のようになります。

```

<Window x:Class="StackPanelSample01.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">
    <StackPanel>
        <Button Content="Button" />
        <!-- 左寄せ -->
        <Button Content="Button" HorizontalAlignment="Left" />
        <!-- 右寄せ -->
        <Button Content="Button" HorizontalAlignment="Right" />
        <!-- センタリング -->
        <Button Content="Button" HorizontalAlignment="Center" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
    </StackPanel>
</Window>

```

一部のボタンでは HorizontalAlignment プロパティで水平方向の配置の仕方を指定しています。この Window を表示すると以下のようになります。水平方向の表示位置を設定したボタンの表示のされかたに注目してください。



続けて Orientation を Horizontal にして水平方向に子要素を配置する場合の XAML を以下に示します。

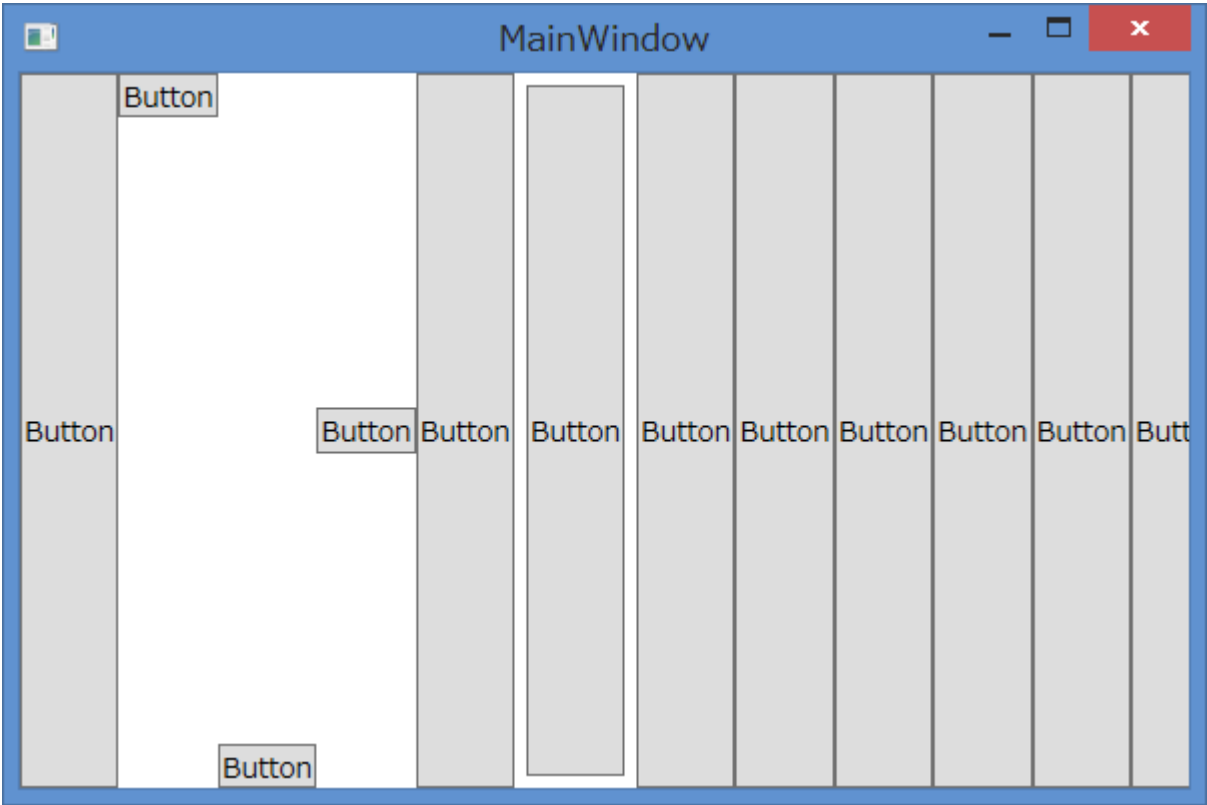
```

<Window x:Class="StackPanelSample02.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">
    <StackPanel Orientation="Horizontal">
        <Button Content="Button" />
        <!-- 上寄せ -->
        <Button Content="Button" VerticalAlignment="Top" />
        <!-- 下寄せ -->
        <Button Content="Button" VerticalAlignment="Bottom"/>
        <!-- 中央寄せ -->
        <Button Content="Button" VerticalAlignment="Center"/>
        <Button Content="Button" />
        <!-- ボタンの周囲に余白を作る -->
        <Button Content="Button" Margin="5" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
        <Button Content="Button" />
    </StackPanel>
</Window>

```

これも一部ボタンで、VerticalAlignment プロパティを指定して垂直方向の位置の調整をおこなっています。さらに Margin というプロパティを使用して、レイアウト可能な領域に対して上下左右の余白を設定しているボタンもあります。Margin プロパティは Border コントロールの Padding プロパティと同じ Thickness 型なので、"左, 上, 右, 下"のようにカンマ区切りで値を指定できます。今回の例では、上下左右に一括で 5px の余白をとりたかったので単に 5 と指定しています。

この Window の表示結果を以下に示します。



VerticalAlignment を設定しているボタンとしていないボタンの表示位置の違いと、Margin を指定しているボタンの周りに余白があることが確認できます。また、水平方向のときと同様に画面からはみ出したものは表示されないことも確認できます。

4.1.5. DockPanel コントロール

DockPanel コントロールは、コントロールを上下左右と残りの部分にわけて配置するコントロールです。一見 Explorer 風の左にツリー、右に詳細、上にメニューとツールバー、下にステータスバーといったユーザーインターフェースが簡単に作れそうですが、マウスで領域のサイズ変更などをしようとすると、とたんに難易度が跳ね上がるためスタティックなレイアウトを作るときに利用するくらいが無難だと思われます。

DockPanel は以下のようなプロパティで子要素のレイアウトを制御します。

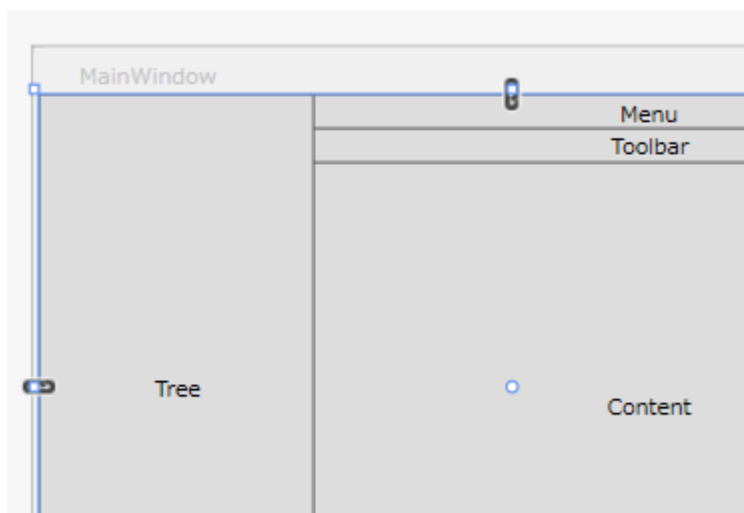
プロパティ	説明
Dock 添付プロパティ	System.Windows.Controls.Dock 列挙体で規定値は Left です。列挙体に定義されてる値は左に配置する Left、上に配置する Top、右に配置する Right、下に配置する Bottom になります。

bool LastChildFill { get; set; }	最後に追加した子を残りの余白全体に敷き詰めるか指定します。規定値は true です。
---	--

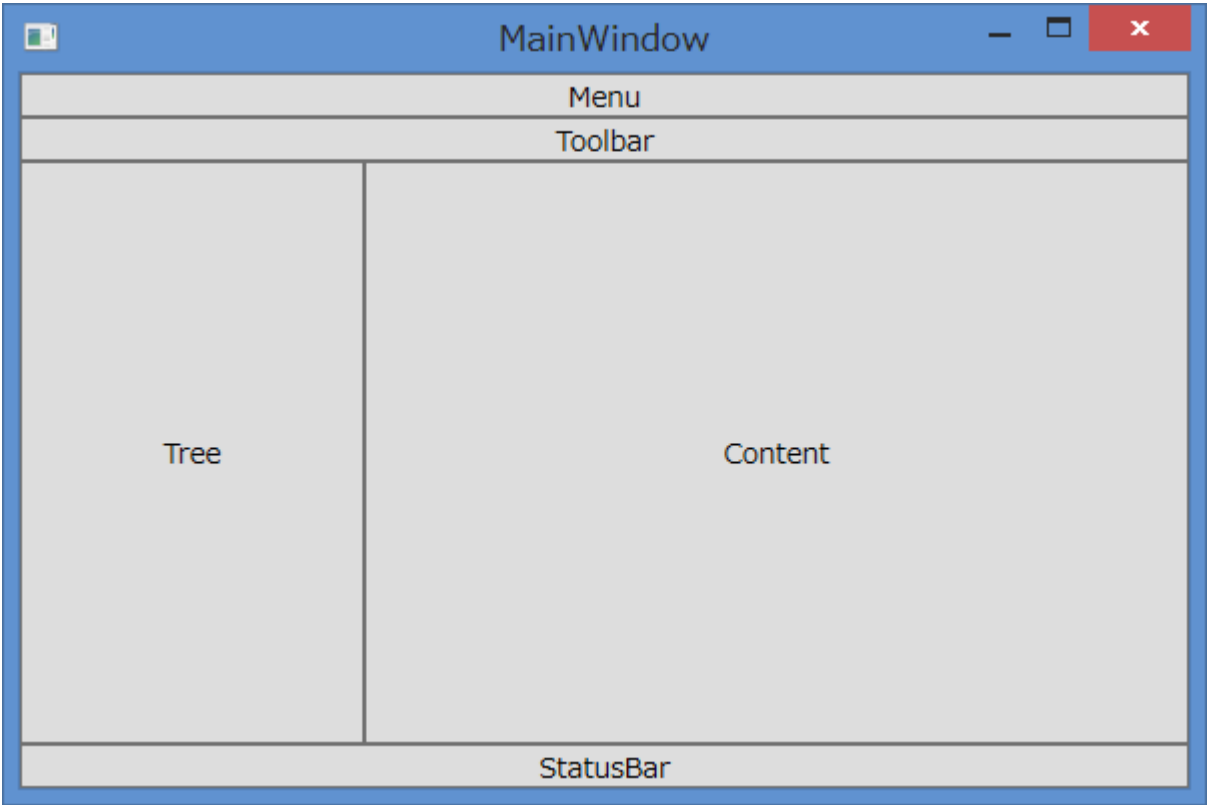
DockPanel を使ってエクスプローラーライクな画面レイアウトを作る XAML の例を以下に示します。

```
<Window x:Class="DockPanelSample.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">
    <DockPanel>
        <!-- メニューやツールバー -->
        <Button DockPanel.Dock="Top" Content="Menu" />
        <Button DockPanel.Dock="Top" Content="Toolbar" />
        <!-- ステータスバー -->
        <Button DockPanel.Dock="Bottom" Content="StatusBar" />
        <!-- ツリーが表示される場所 最低限の幅確保のため MinWidth プロパティを指定 -->
        <Button DockPanel.Dock="Left" Content="Tree" MinWidth="150" />
        <!-- エクスプローラーの右側の領域 -->
        <Button Content="Content" />
    </DockPanel>
</Window>
```

上側、下側、左側、残り全体の順番で要素を配置しています。この順番には意味があって左側に置くものを上側や下側よりも先に置くと、メニューやツールバーにあたる要素よりも先に配置されるため以下のような見た目になります。



この XAML を実行すると、以下のような結果になります。



4.1.6. WrapPanel コントロール

WrapPanel コントロールは、StackPanel コントロールと同じように横や縦に要素を並べて表示するコントロールです。StackPanel コントロールとの違いは、子要素が外にはみ出したときの挙動で StackPanel が潔く表示しなかったのに対して WrapPanel は、折り返して表示を行います。

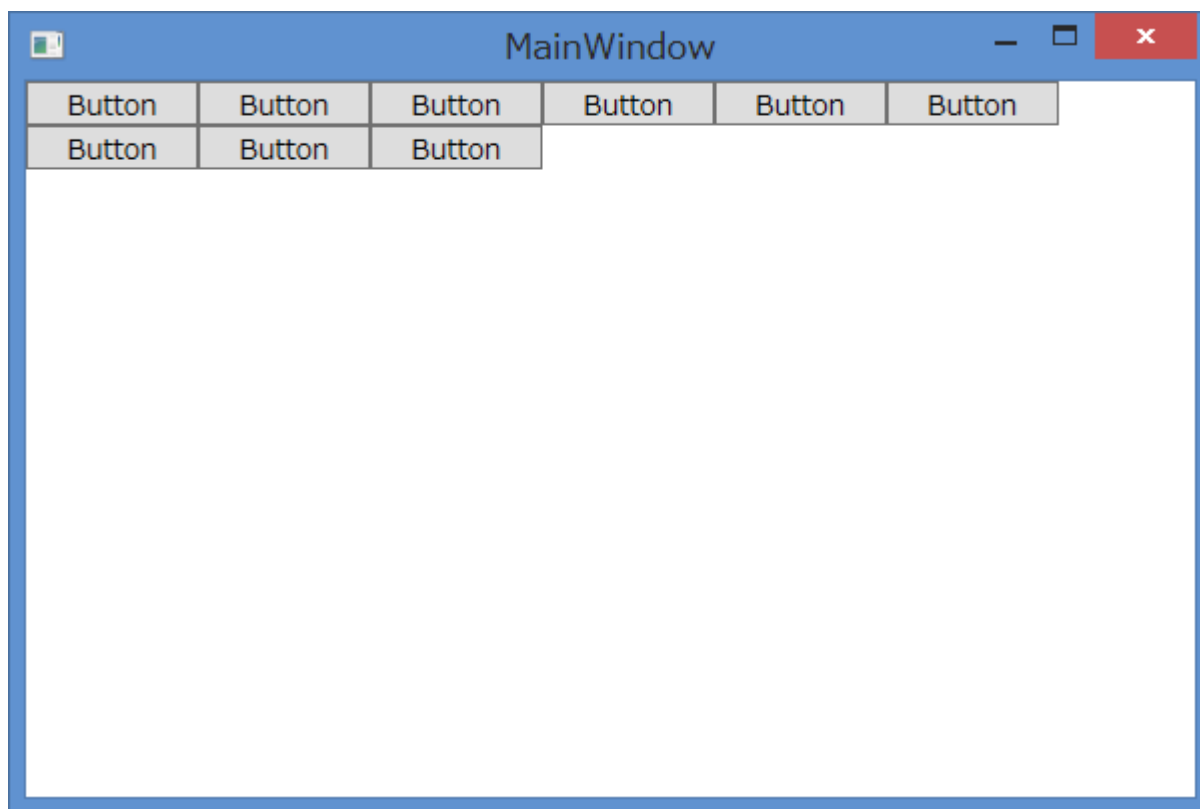
WrapPanel で使用する主なプロパティを以下に示します。

プロパティ	説明
Orientation Orientation { get; set; }	StackPanel と同様に横並びに配置する場合は Horizontal を設定し、縦並びに配置する場合は Vertical を設定します。
double ItemHeight { get; set; }	WrapPanel 内の要素の高さを設定します。NaN の場合は、子要素の高さを優先します。
double ItemWidth { get; set; }	WrapPanel 内の要素の幅を設定します。NaN の場合は、子要素の幅を優先します。

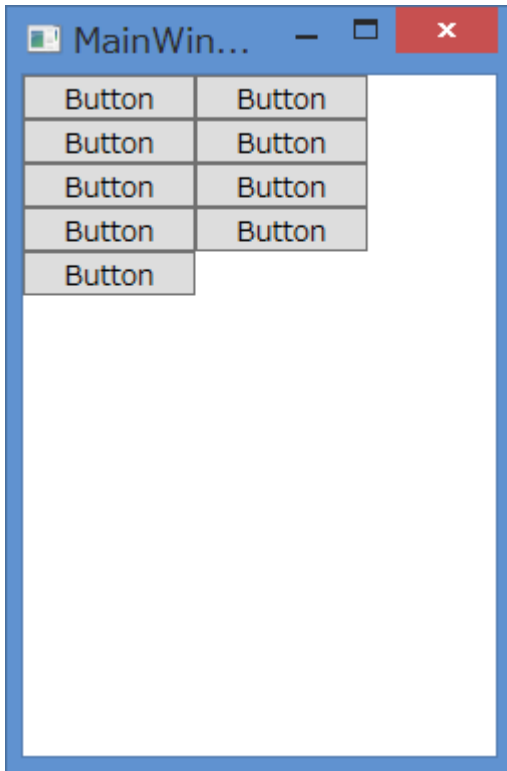
WrapPanel に幅 75px のボタンを並べる XAML を以下に示します。


```
<Window x:Class="WrapPanelSample01.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">
  <WrapPanel>
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
    <Button Content="Button" Width="75" />
  </WrapPanel>
</Window>
```

実行すると、以下のようになります。StackPanel と異なり、画面右端でボタンが折り返され 2 行に渡ってボタンが配置されます。



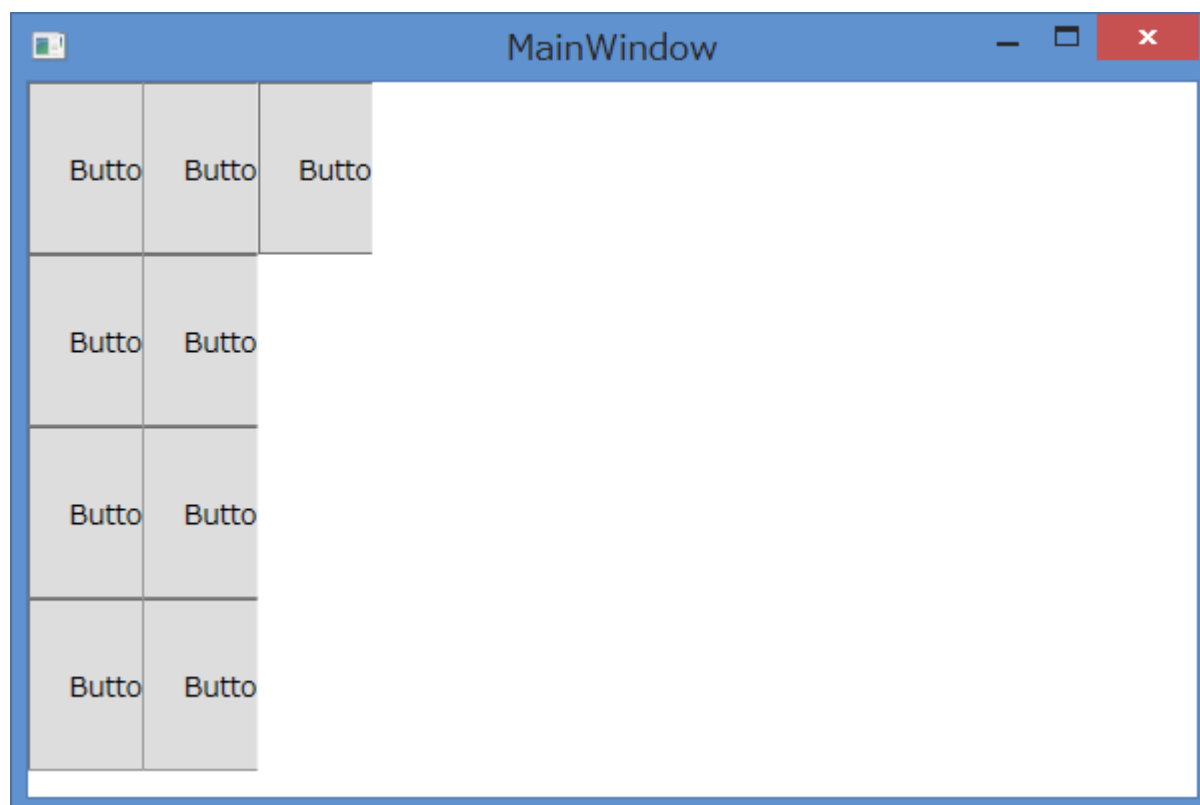
Window のサイズを変えると、そのときのサイズに最適な形に要素を再配置します。



次に、Orientation を Vertical にして ItemHeight と ItemWidth を指定した場合の XAML を以下に示します。

```
<Window x:Class="WrapPanelSample02.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">
    <WrapPanel Orientation="Vertical" ItemHeight="75" ItemWidth="50">
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
        <Button Content="Button" Width="75" />
    </WrapPanel>
</Window>
```

高さを特に指定していないボタンに ItemHeight を指定している点と、Width を 75 に指定しているボタンがある状態で ItemWidth を 50 にしている点がポイントです。実行すると、以下のようになります。



ボタンの高さが `ItemHeight` で指定した高さになっていることと、ボタンの幅が `ItemWidth` で指定した幅で描画が切られていることが確認できます。

4.1.7. ViewBox コントロール

`ViewBox` コントロールは、子要素を拡大縮小して表示するコントロールです。拡大縮小されたコントロールは、もとの動作を 100% 保った状態なので操作できます。ここら辺の動作は、WPF がベクターベースのテクノロジーである強みだといえます。

`ViewBox` コントロールで使用する主なプロパティを以下に示します。

プロパティ	説明
Stretch <code>Stretch { get; set; }</code>	拡大縮小を行う時に、どのように行うか指定します。何も行わない <code>None</code> と、領域を埋めるようにサイズを調整する <code>Fill</code> と、縦横比を維持しながら領域内に収まるようにサイズを調整する <code>Uniform</code> と、縦横比を維持しながら領域いっぱいサイズを調整する <code>UniformFill</code> があります。デフォルトは <code>Uniform</code> です。

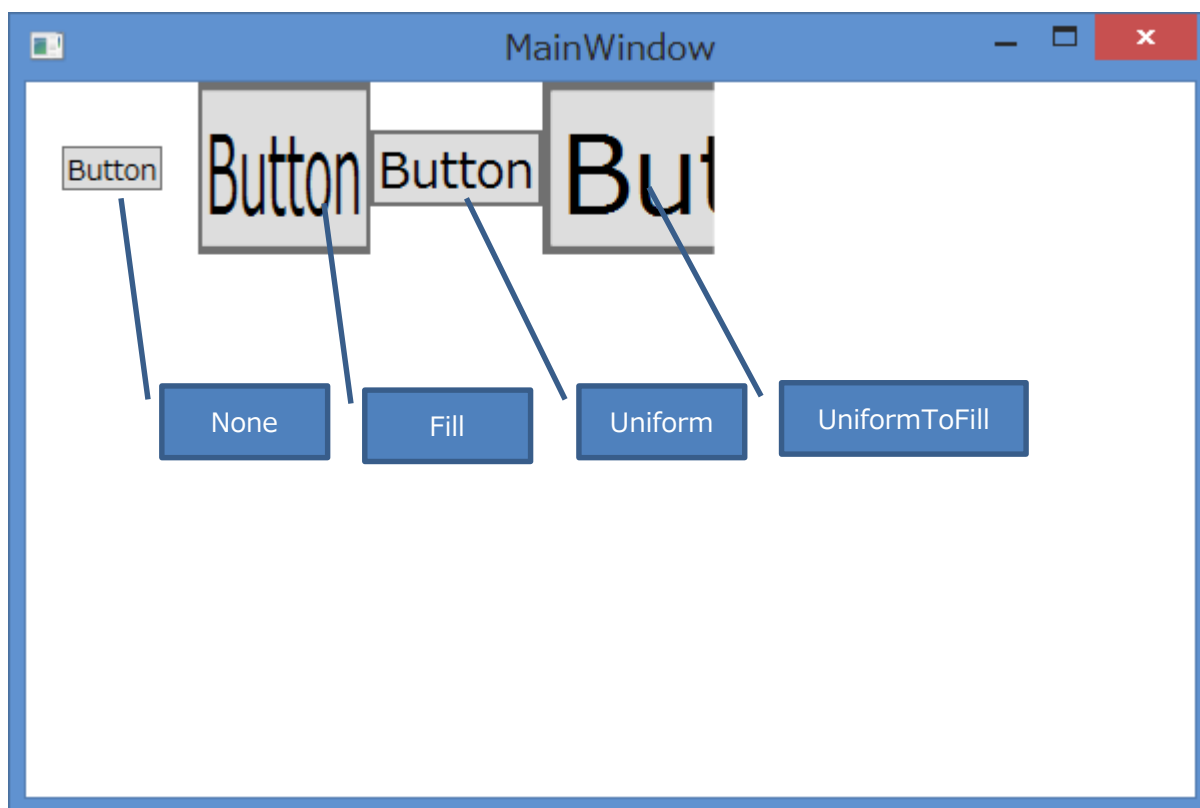
StretchDirection StretchDirection { get; set; }

拡大を行うのか縮小を行うのか、両方やるのかを指定します。拡大のみを行う UpOnly と、縮小のみを行う DownOnly と、拡大縮小を行う Both があります。デフォルトは Both です。

75px × 75px の ViewBox にボタンを置いて Stretch プロパティを変えて違いを確認します。XAML を以下に示します。

```
<Window x:Class="ViewBoxSample.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">
    <WrapPanel>
        <Viewbox Width="75" Height="75" Stretch="None">
            <Button Content="Button" />
        </Viewbox>
        <Viewbox Width="75" Height="75" Stretch="Fill">
            <Button Content="Button" />
        </Viewbox>
        <Viewbox Width="75" Height="75" Stretch="Uniform">
            <Button Content="Button" />
        </Viewbox>
        <Viewbox Width="75" Height="75" Stretch="UniformToFill">
            <Button Content="Button" />
        </Viewbox>
    </WrapPanel>
</Window>
```

これを実行すると、以下のような結果になります。



左から None, Fill, Uniform, UniformToFill になります。文章ではわかりにくかった Uniform と UniformToFill の違いですが、Uniform が領域内に収まるように拡大していて、UniformToFill が領域いっぱいになるように拡大してはみ出た部分は表示されていないことが確認できます。さらに、ボタンが動くことをマウスオーバーやクリックするなどして確認してみてください。ViewBox は使いどころが難しいですが、簡単にサイズに合わせて拡大縮小できる機能を実現する手段があるということは頭に入れておくといいでしょう。

4.1.8. ScrollViewer コントロール

ScrollViewer コントロールは、名前のとおり子要素が ScrollViewer より大きな場合にスクロールバーを出して要素を閲覧できるようにするコントロールです。ScrollViewer では、縦スクロールバー・横スクロールバーの表示方法の指定や、スクロール時に論理単位でスクロール(要素単位でスクロール)するか物理単位でスクロール(ピクセル単位でスクロール)するか指定できます。

ScrollViewer コントロールで使用する主なプロパティを以下に示します。

プロパティ	説明
bool CanContentScroll { get; set; }	スクロールを物理単位で行うか論理単位で行うか設定します。true の場合は論理スクロールで false の場合

	は物理スクロールです。デフォルトは物理スクロールです。
ScrollBarVisibility Horizontal ScrollBarVisibility { get; set; }	水平スクロールバーを表示するかどうか指定します。スクロールバーを非表示にしてサイズを ScrollViewer と同じ幅にする場合は Disabled を、スクロールバーが必要な場合にのみ表示する場合は Auto を、スクロールバーを表示しない場合は Hidden を、スクロールバーを常に表示する場合は Visible を設定します。デフォルト値は Hidden です。
ScrollBarVisibility Vertical ScrollBarVisibility { get; set; }	垂直スクロールバーを表示するかどうか指定します。デフォルト値は Visible です。
double ScrollableHeight { get; }	子要素の高さを取得します。
double ScrollableWidth { get; }	子要素の幅を取得します。
double HorizontalOffset { get; }	水平スクロールバーの位置を取得します。
double VerticalOffset { get; }	垂直スクロールバーの位置を取得します。

ScrollViewer コントロールのスクロールバーの位置を表すプロパティは、読み取り専用になっています。そのため、スクロールバーの位置の操作を行う場合はメソッドを使用します。スクロールバーの位置を制御する主なメソッドを以下に示します。

メソッド	説明
void ScrollToTop()	一番上のコンテンツまでスクロールします。
void ScrollToBottom()	一番下のコンテンツまでスクロールします。
void ScrollToLeftEnd()	左端までスクロールします。
void ScrollToRightEnd()	右端までスクロールします。
void ScrollToHorizontalOffset(double offset)	offset で指定した場所まで水平スクロールします。

void ScrollToVerticalOffset(double offset)

offset で指定した場所まで垂直スクロールします。

ScrollViewer コントロールには、この他にも行単位のスクロールを制御する Line****メソッドやページ単位のスクロールを制御する Page****メソッド(****にはスクロール方向が入ります)や、タッチ操作にどう対応するかを設定するプロパティなどがあります。詳細は、MSDN の ScrollViewer コントロールのページを確認してください。

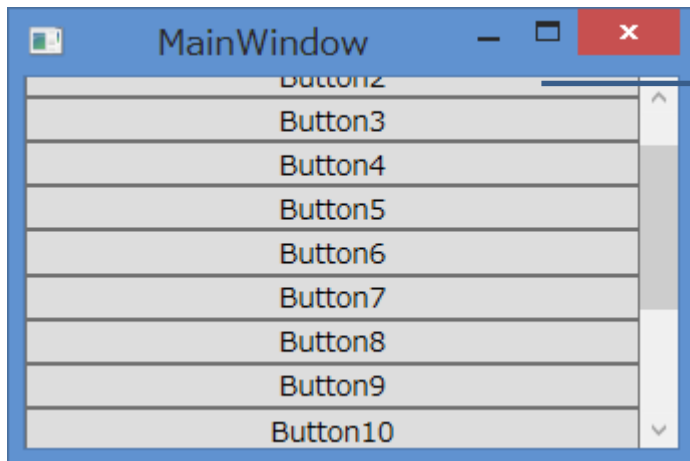
ScrollViewer コントロール

<http://msdn.microsoft.com/ja-jp/library/ms612678.aspx>

ScrollViewer コントロールを使ってスクロールする画面を作成する XAML の例を以下に示します。

```
<Window x:Class="ScrollViewerSample01.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="200" Width="300">
    <ScrollViewer>
        <StackPanel>
            <Button Content="Button1" />
            <Button Content="Button2" />
            <Button Content="Button3" />
            <Button Content="Button4" />
            <Button Content="Button5" />
            <Button Content="Button6" />
            <Button Content="Button7" />
            <Button Content="Button8" />
            <Button Content="Button9" />
            <Button Content="Button10" />
            <Button Content="Button11" />
            <Button Content="Button12" />
            <Button Content="Button13" />
            <Button Content="Button14" />
            <Button Content="Button15" />
        </StackPanel>
    </ScrollViewer>
</Window>
```

実行するとボタンが縦にならんで縦スクロールバーがある Window が表示されます。デフォルトでは物理スクロールなので、下図のようにボタンの途中でスクロールバーを止めることができます。

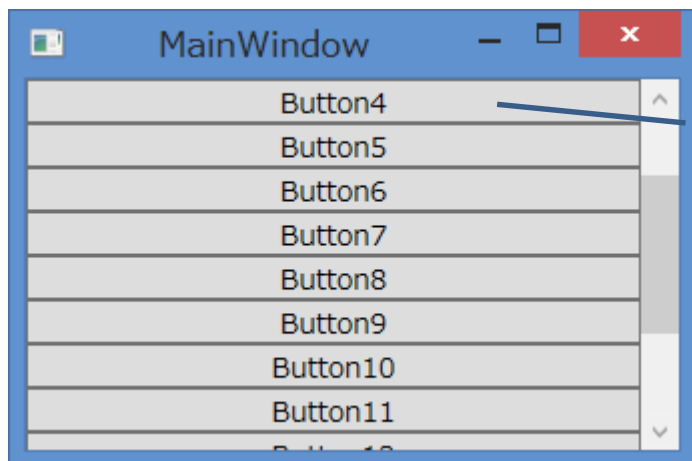


ボタンの途中でスクロールバー
を止めます。

次に、論理スクロールを有効にして縦スクロールバーの表示を Auto にした XAML を以下に示します。

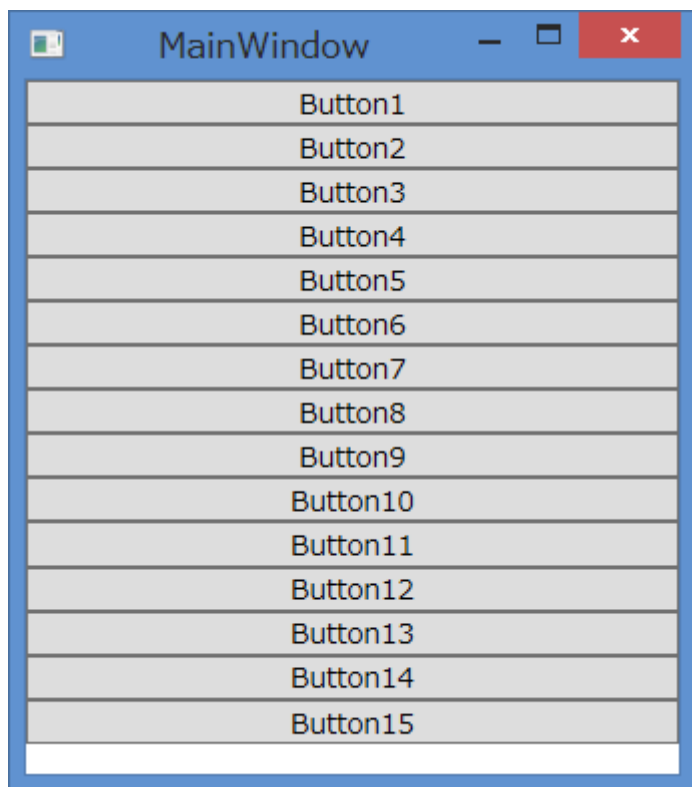
```
<Window x:Class="ScrollViewerSample02.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="300">
    <ScrollViewer VerticalScrollBarVisibility="Auto" CanContentScroll="True">
        <StackPanel>
            <Button Content="Button1" />
            <Button Content="Button2" />
            <Button Content="Button3" />
            <Button Content="Button4" />
            <Button Content="Button5" />
            <Button Content="Button6" />
            <Button Content="Button7" />
            <Button Content="Button8" />
            <Button Content="Button9" />
            <Button Content="Button10" />
            <Button Content="Button11" />
            <Button Content="Button12" />
            <Button Content="Button13" />
            <Button Content="Button14" />
            <Button Content="Button15" />
        </StackPanel>
    </ScrollViewer>
</Window>
```

実行してスクロールをすると、ボタンの途中でスクロールバーを止めることができないことが確認できます。



必ずボタンの一番上がスクロールバーの一番上とピタリとあう表示になります。

縦スクロールバーの表示を Auto にしたので、スクロールバーが必要無くなるまで Window を大きくするとスクロールバーが消えます。



次に ScrollViewer コントロールのスクロールバーの位置を制御する例を示します。XAML に ScrollViewer コントロールとスクロールバーを動かす処理を起動するボタンを置いた画面を作成します。ScrollViewer コントロールはコードビハインドから使用できるように x:Name 属性を指定しています。画面の上部に置いているボタンには Click イベントを設定しています。XAML を以下に示します。

```
<Window x:Class="ScrollViewerSample03.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="300">
    <DockPanel>
        <Button DockPanel.Dock="Top"
                Margin="5"
                Content="ScrollToHalfVerticalOffset"
                Click="ScrollToHalfVerticalOffsetButton_Click" />
        <ScrollViewer x:Name="scrollViewer">
            <StackPanel>
                <Button Content="Button1" />
                <Button Content="Button2" />
                <Button Content="Button3" />
                <Button Content="Button4" />
                <Button Content="Button5" />
                <Button Content="Button6" />
                <Button Content="Button7" />
                <Button Content="Button8" />
                <Button Content="Button9" />
                <Button Content="Button10" />
                <Button Content="Button11" />
                <Button Content="Button12" />
                <Button Content="Button13" />
                <Button Content="Button14" />
                <Button Content="Button15" />
            </StackPanel>
        </ScrollViewer>
    </DockPanel>
</Window>
```

画面上部のボタンをクリックすると、縦スクロールバーを中央に移動させるコードを以下に示します。

```

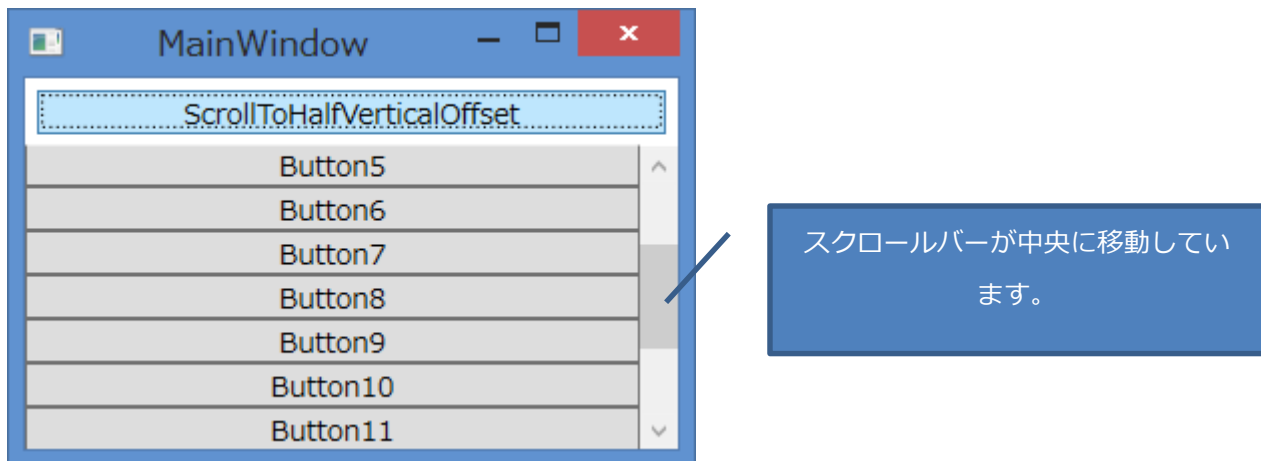
namespace ScrollViewerSample03
{
    using System.Windows;

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

        private void ScrollToHalfVerticalOffsetButton_Click(object sender, RoutedEventArgs e)
        {
            // 垂直スクロールバーの位置を真ん中に設定
            this.scrollViewer.ScrollToVerticalOffset(this.scrollViewer.ScrollableHeight / 2);
        }
    }
}

```

ScrollToVerticalOffset メソッドで縦方向のスクロールバーの位置を設定しています。スクロールバーの位置は ScrollableHeight プロパティでスクロール可能な高さを取得して 2 で割ることで、真ん中の位置を算出しています。実行してボタンを押した状態の画面を以下に示します。スクロールバーが中央にきていることが確認できます。



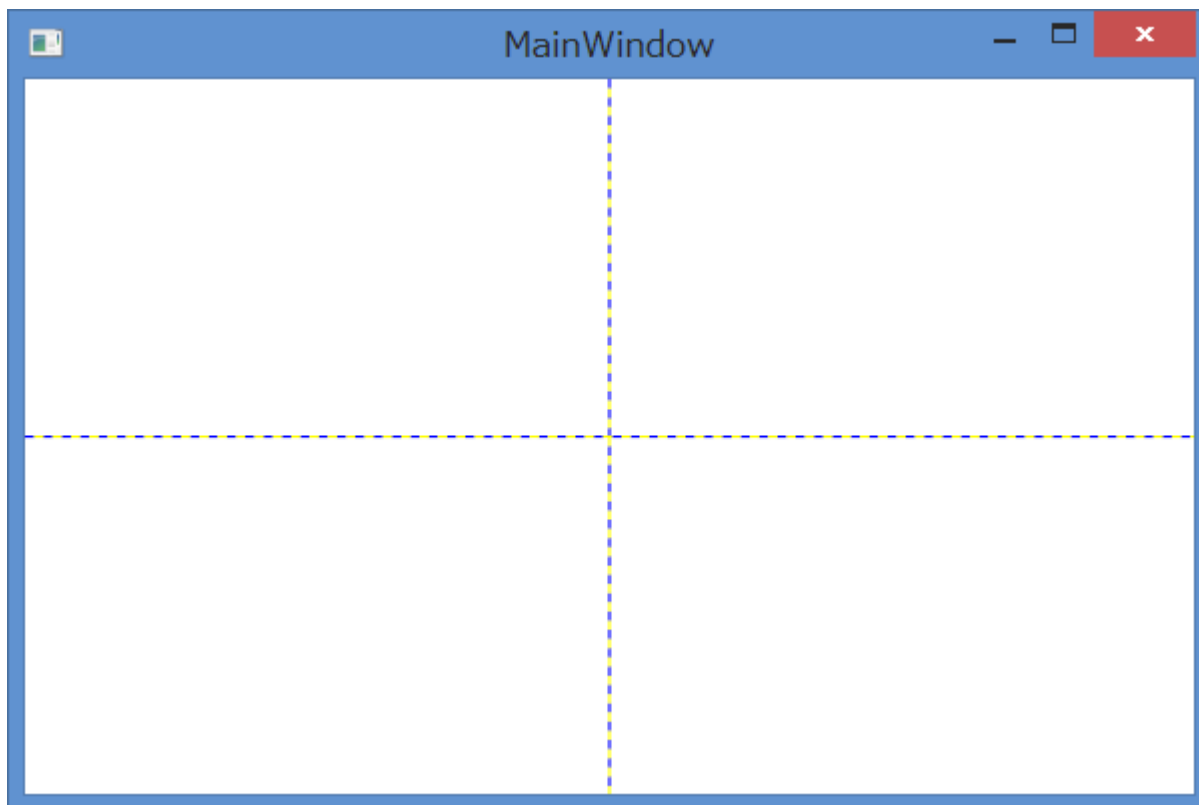
4.1.9. Grid コントロール

Grid コントロールは、テーブルレイアウトを行うための WPF のコントロールです。行と列を定義して、子要素を任意の行と列に配置できます。RowSpan や ColumnSpan を設定することで複数行や複数列にまたがって子要素を配置することが出来ます。

Grid コントロールで行を定義するには、RowDefinitions プロパティに RowDefinition クラスを設定します。列を定義するには、ColumnDefinitions プロパティに ColumnDefinition クラスを設定します。どちらもコレクション型のプロパティなので、複数の RowDefinition と ColumnDefinition が定義できます。2 行 2 列の Grid を定義する XAML は以下ようになります。

```
<Grid ShowGridLines="True">
  <!-- 行を 2 つ定義 -->
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <!-- 列を 2 つ定義 -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
</Grid>
```

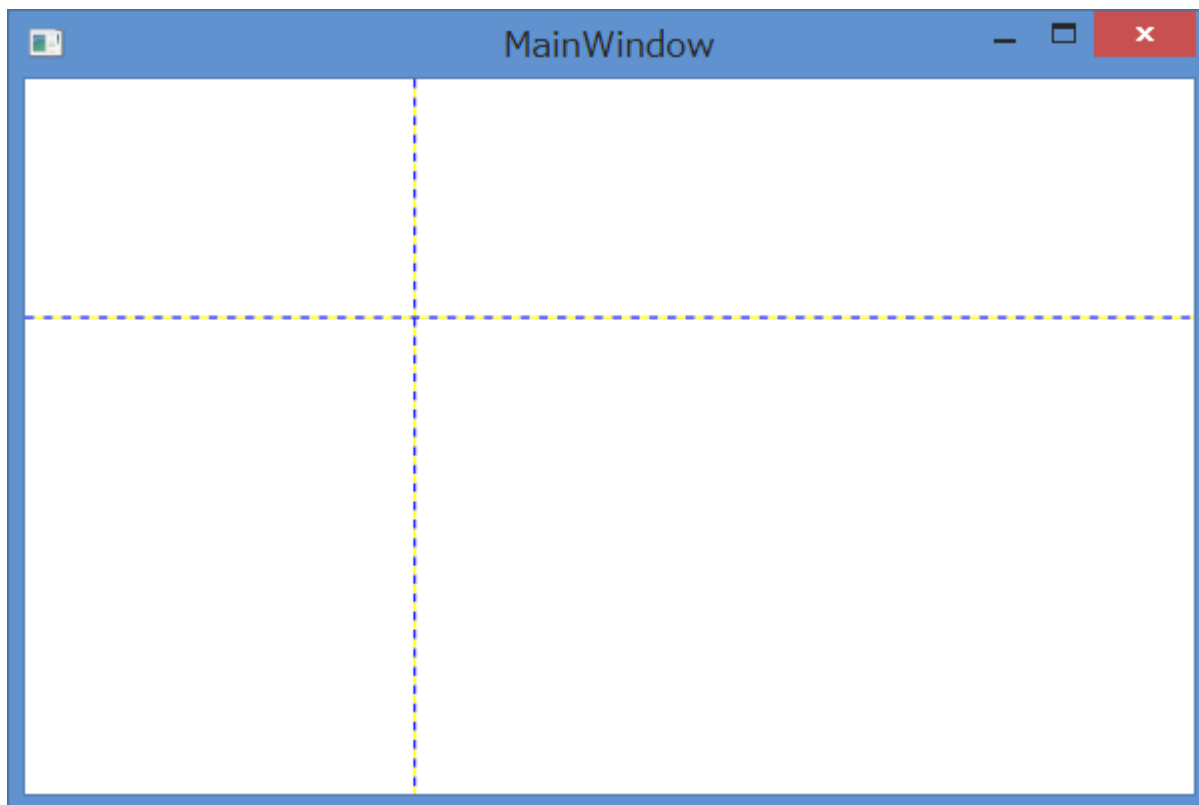
ShowGridLine プロパティは、True に設定すると行や列が定義されたことがわかるように点線を表示するプロパティです。デフォルト値は False です。通常は使用しませんが、レイアウトが意図した通りにできているか確認する際に便利です。ここでは Grid で、どのように行と列が定義されたか確認するために True に設定した状態で説明を行います。この Grid を置いた Window を表示すると以下ようになります。



デフォルトでは、RowDefinition や ColumnDefinition で定義した行や列の幅は同じ比率になります。Width や Height を設定することでこの比率を変更できます。幅や高さを比率で設定するには「数字*」という方法で記述します。1 対 2 の比率で行と列の幅を指定すると以下のような XAML になります。

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <!-- 1:2 の比率で行を定義 -->
    <RowDefinition Height="1*" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <!-- 1:2 の比率で列を定義 -->
    <ColumnDefinition Width="1*" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>
</Grid>
```

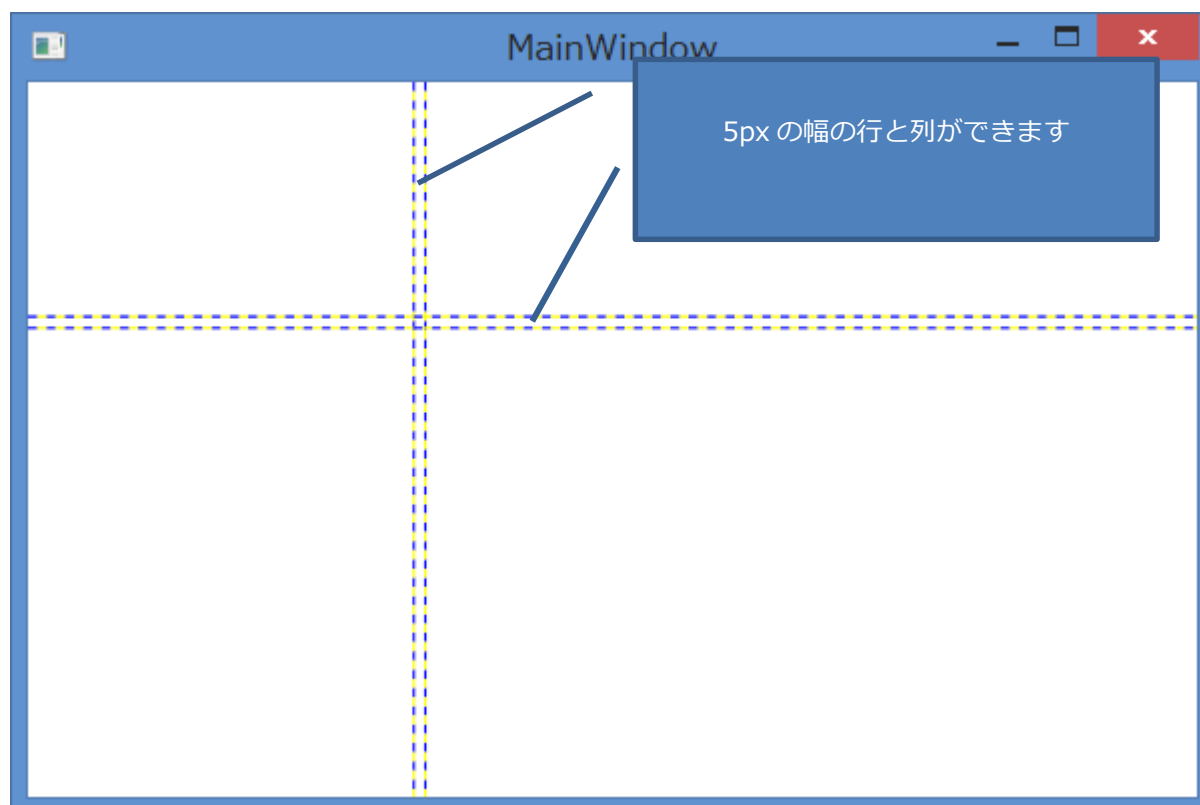
実行すると、以下のようになります。



比率での指定の他に、ピクセルで幅を明示的に指定することもできます。ピクセルで指定する場合には数字を Width や Height プロパティに設定します。設定例を以下に示します。

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="1*" />
    <!-- 幅 5px -->
    <RowDefinition Height="5" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*" />
    <!-- 幅 5px -->
    <ColumnDefinition Width="5" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>
</Grid>
```

実行すると以下のような表示になります。



RowDefinition と ColumnDefinition の高さや幅の指定方法には、その行と列に配置されてる子要素の大きさに合わせてサイズが決まる Auto という指定方法もあります。

Auto を試すためには、Grid の任意の位置に子要素を置く方法が必要になるので先に子要素を置く方法を説明します。子要素の位置を指定するには以下の添付プロパティを使用します。

プロパティ	説明
Row 添付プロパティ	Grid の何行目に置くか設定します。デフォルト値は 0 です。
Column 添付プロパティ	Grid の何列目に置くか設定します。デフォルト値は 0 です。
RowSpan 添付プロパティ	何行にわたって要素を置くか設定します。デフォルト値は 1 です。
ColumnSpan 添付プロパティ	何列にわたって要素を置くか設定します。デフォルト値は 1 です。

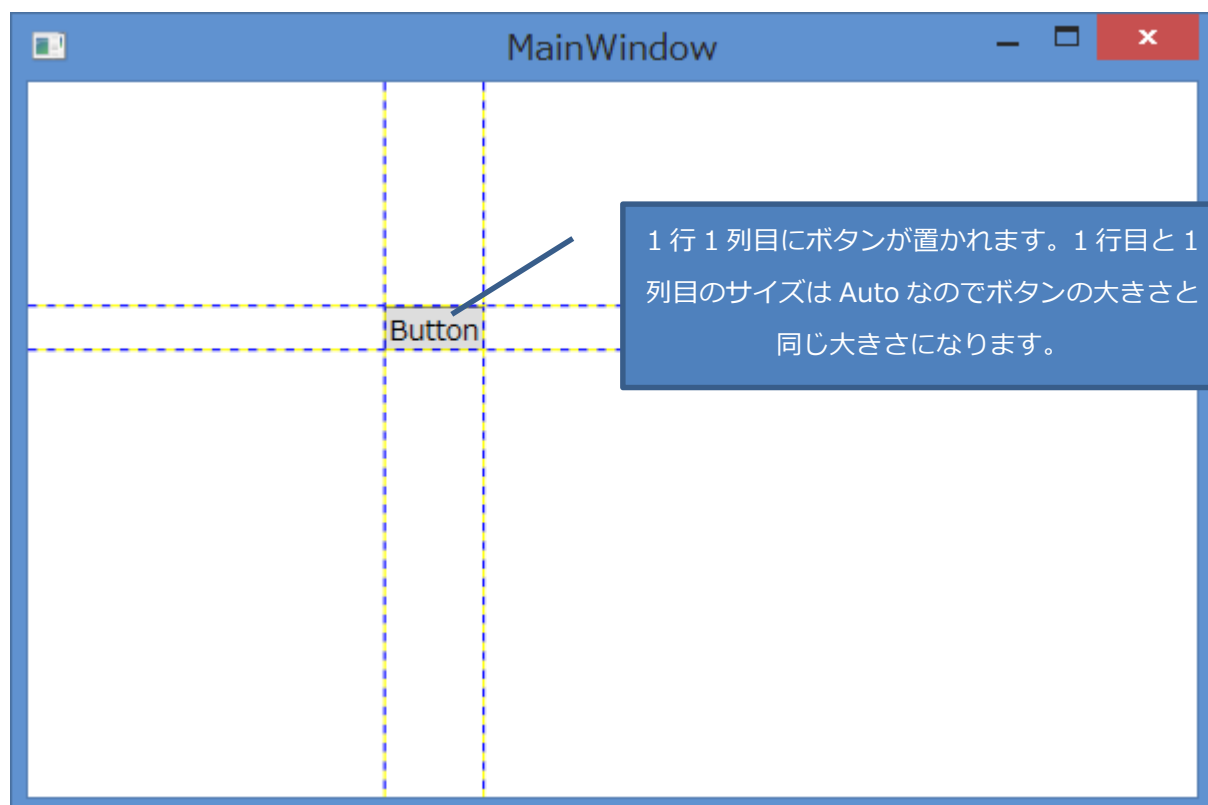
使用例を示すために以下のような 3 行 3 列の Grid を使用します。

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="1*" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>
</Grid>
```

この Grid の 1 行 1 列目に Button を置くには以下のように記述します。

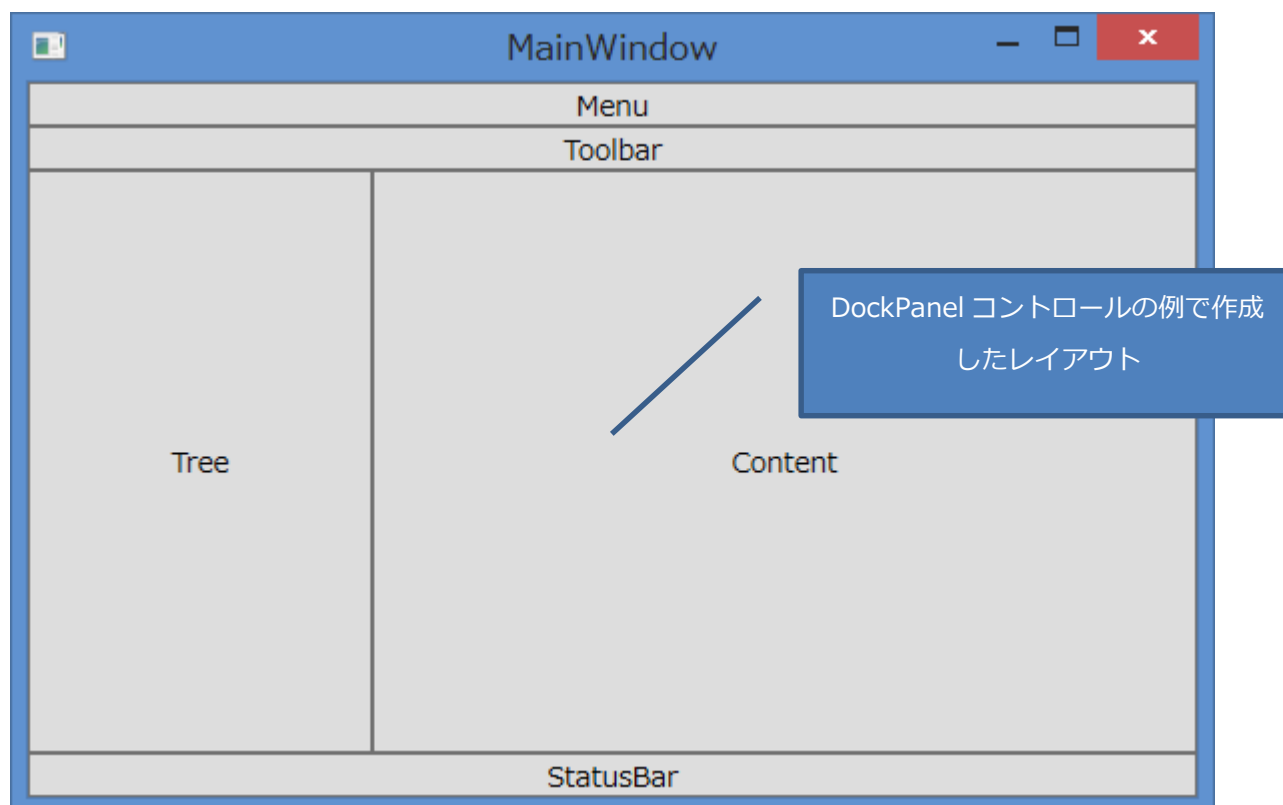
```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="1*" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>
  <!-- 1 行 1 列目に配置 -->
  <Button Content="Button" Grid.Row="1" Grid.Column="1" />
</Grid>
```

実行すると、以下のようになります。



4.1.10. Grid コントロールでのレイアウト例

Grid コントロールは、これまで説明した StackPanel や DockPanel などと比べて非常に強力なレイアウトコントロールになります。Grid コントロールでレイアウトするときは、これまでに紹介した方法を使って最終的なレイアウトを実現するのに必要な行と列の数と、それぞれに設定するサイズを決めて、そこに目的のコントロールを配置するという手順で行います。具体的な例として DockPanel コントロールで作成した下図のようなレイアウトを Grid で作成する方法について説明します。



順を追って作成していきます。このレイアウトには、Menu 用の行、Toolbar 用の行、Tree と Content 用の行、StatusBar 用の行の 4 行が必要になります。各行のサイズは Menu と Toolbar と StatusBar が子要素の高さで、Tree と Content の行が残りの部分を占有します。列に注目すると Tree 用の列と Content 用の列の 2 列が必要になります。左側の Tree の列は 150px 固定で Content の列が残りの部分を占有します。これを RowDefinition と ColumnDefinition で記述すると以下のような XAML になります。

```
<Window x:Class="GridSample01.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 ShowGridLines="True">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150" />
            <ColumnDefinition Width="*/>
        </Grid.ColumnDefinitions>
    </Grid>
</Window>
```

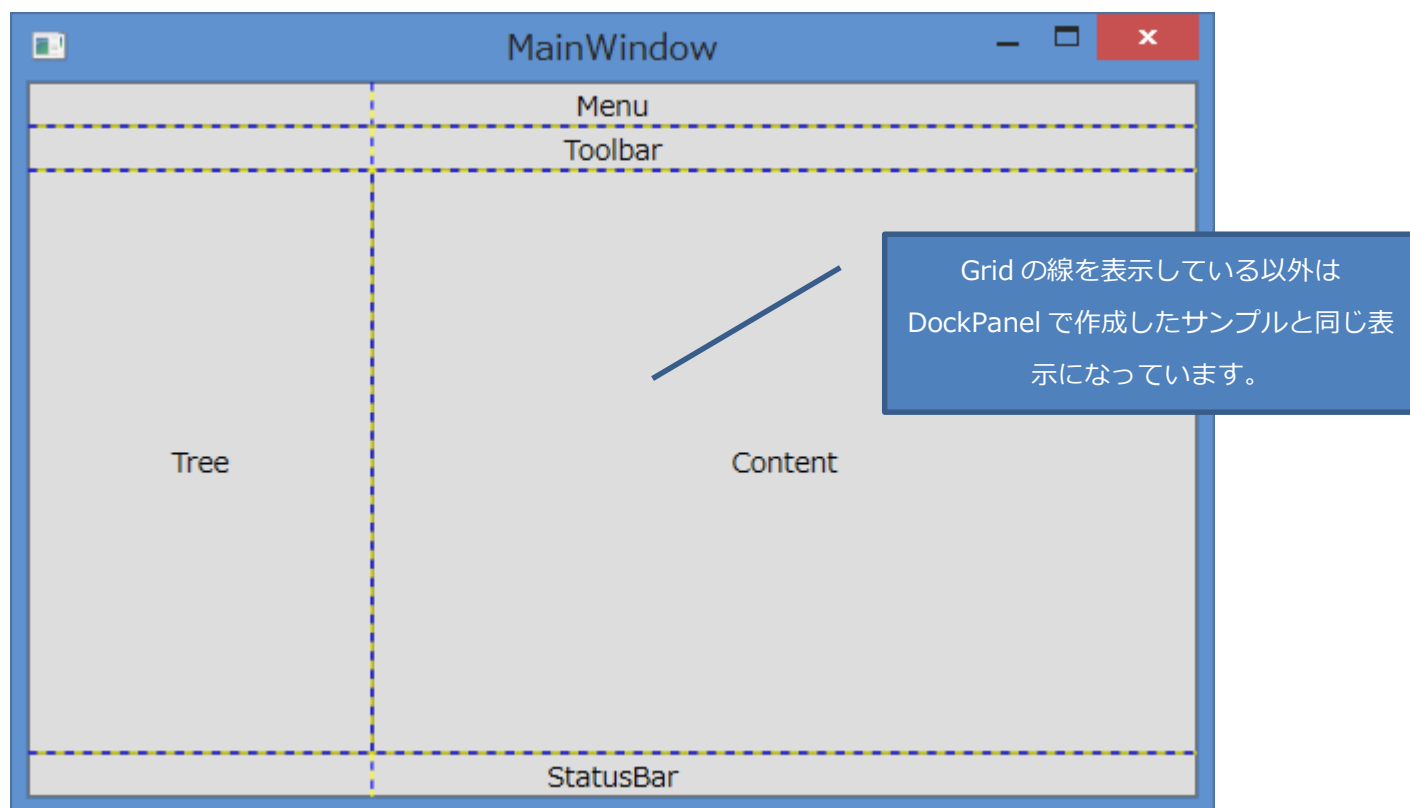
行と列の定義が出来たので子要素の Button を Grid.Row、Grid.Column、Grid.ColumnSpan、Grid.RowSpan の添付プロパティを使って置いていきます。注意する点は、Menu と Toolbar と Statusbar は 2 列に渡って配置するので Grid.ColumnSpan を 2 にする点です。ボタンを配置した Grid の XAML を以下に示します。

```

<Window x:Class="GridSample01.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 ShowGridLines="True">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="150" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <!-- メニューやツールバー -->
    <Button Grid.Row="0" Grid.ColumnSpan="2" Content="Menu" />
    <Button Grid.Row="1" Grid.ColumnSpan="2" Content="Toolbar" />
    <!-- ステータスバー -->
    <Button Grid.Row="3" Grid.ColumnSpan="2" Content="StatusBar" />
    <!-- ツリーが表示される場所 -->
    <Button Grid.Row="2" Content="Tree" />
    <!-- エクスプローラーの右側の領域 -->
    <Button Grid.Row="2" Grid.Column="1" Content="Content" />
  </Grid>
</Window>

```

この Window を表示すると、以下のようになります。DockPanel コントロールで作成した画面と同じ表示になっています。破線とボタンの位置を確認して表示内容と XAML の対応を確認してください。



4.1.11. GridSplitter コントロール

Grid コントロールの特徴の 1 つとして GridSplitter コントロールを使ったマウスでのサイズ変更への対応があります。GridSplitter コントロールを Grid コントロールの区切りに沿って配置することで、エクスプローラーのように左右（上下も可）で領域のサイズを変えることができます。

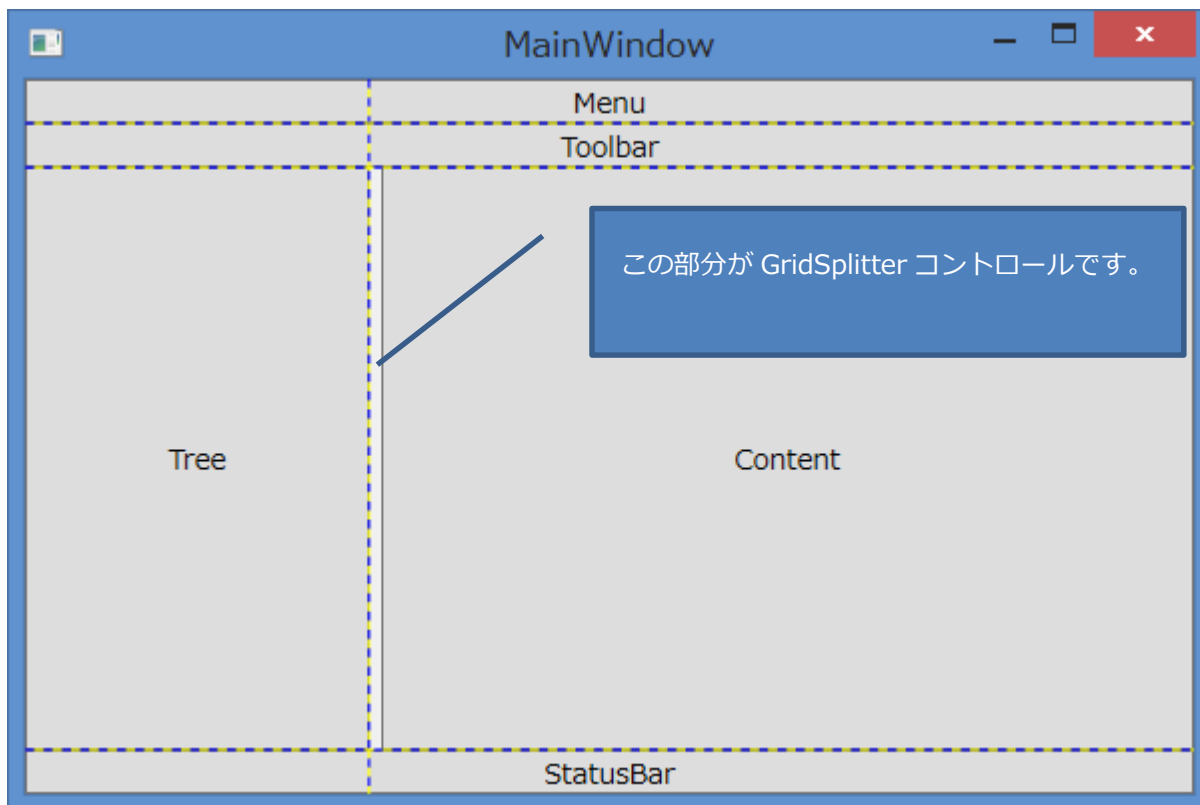
例として先ほど作成した XAML に GridSplitter コントロールを追加して Tree と Content のサイズをマウスで変更できるようにします。GridSplitter コントロールを追加した Grid コントロール部分の XAML を以下に示します。

```

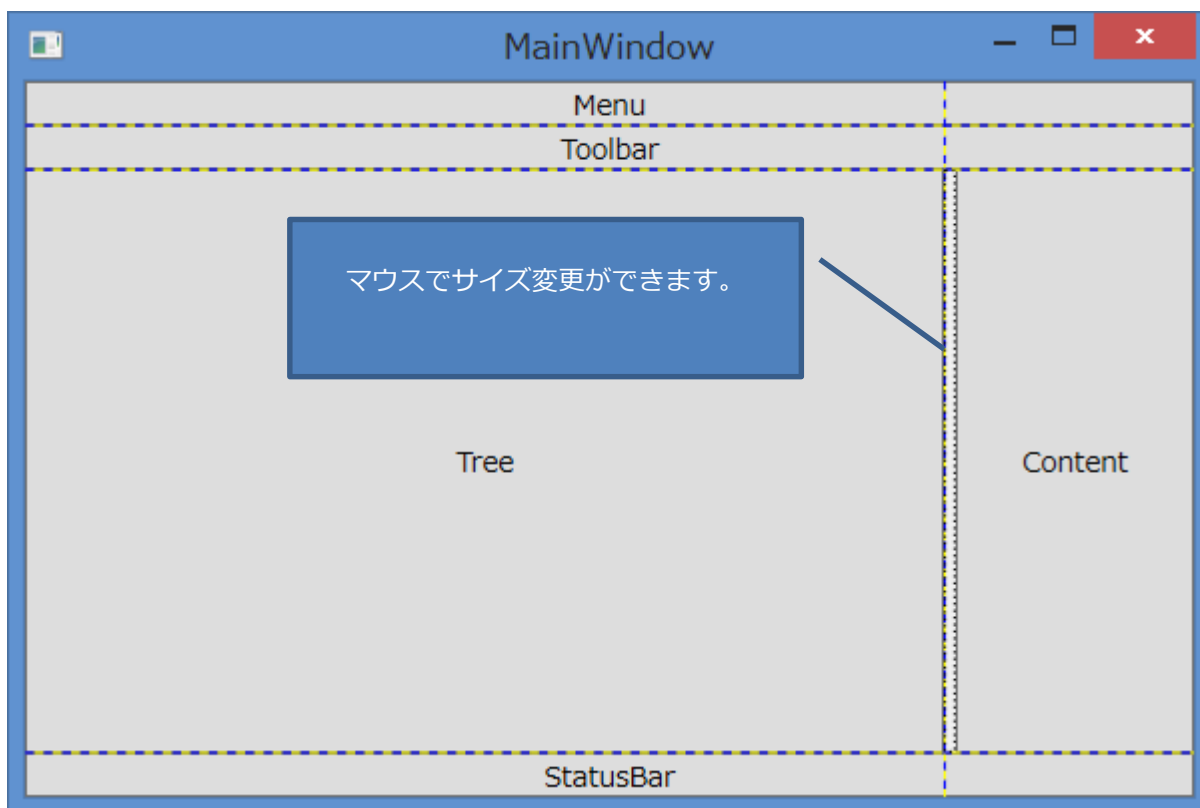
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="150" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <!-- メニューやツールバー -->
  <Button Grid.Row="0" Grid.ColumnSpan="2" Content="Menu" />
  <Button Grid.Row="1" Grid.ColumnSpan="2" Content="Toolbar" />
  <!-- ステータスバー -->
  <Button Grid.Row="3" Grid.ColumnSpan="2" Content="StatusBar" />
  <!-- ツリーが表示される場所 最低限の幅確保のため MinWidth プロパティを指定 -->
  <Button Grid.Row="2" Content="Tree" />
  <!-- エクスプローラーの右側の領域 -->
  <!-- Tree と Content のサイズを変えるための GridSplitter を配置 -->
  <GridSplitter
    Grid.Row="2" Grid.Column="1"
    HorizontalAlignment="Left" VerticalAlignment="Stretch" Width="5" />
  <!-- GridSplitter コントロールを置く余白を確保するために Margin を設定 -->
  <Button Grid.Row="2" Grid.Column="1" Content="Content" Margin="5, 0, 0, 0" />
</Grid>

```

上記の例では、Content の左に幅 5px の GridSplitter コントロールを置いています。Grid コントロールで同じセルにコントロールを置くと重ねて表示するため、重なりを防ぐために Content の左側に 5px のマージンを指定しています。この Window を表示すると以下ようになります。



Tree と Content の間にある GridSplitter コントロールをドラッグすることで以下のようにサイズ変更ができます。



4.1.12. レイアウトに影響を与えるプロパティ

ここまでレイアウトを制御する代表的なコントロールを見てきました。これらのコントロールを組み合わせることで、思った場所にコントロールを置くことができます。ここでは、WPF のほぼすべてのコントロールが共通でもつレイアウトに影響を与えるプロパティについてみていきます。レイアウトコントロールと、ここで紹介するプロパティを組み合わせることで WPF の協力なレイアウトシステムを余すことなく使うことが出来るようになります。

4.1.12.1. 水平方向・垂直方向の位置指定

まず、コントロールの水平方向・垂直方向の位置を指定するプロパティを以下に示します。

プロパティ	説明
HorizontalAlignment HorizontalAlignment { get; set; }	水平方向の配置方法を指定します。左寄せの場合は Left、右寄せの場合は Right、中央寄せの場合は Center、全体にひろげる場合は Stretch を指定します。デフォルト値は Stretch です。一部のコントロール（Label など）ではデフォルト値が変わっているものもあります。
VerticalAlignment VerticalAlignment { get; set; }	素直方向の配置方法を指定します。上寄せの場合は Top、下寄せの場合は Bottom、中央寄せの場合は Center、全体にひろげる場合は Stretch を指定します。デフォルト値は Stretch です。一部のコントロール（ComboBoxItem など）ではデフォルト値が変わっているものもあります。

4.1.12.2. サイズの指定

コントロール自身の大きさを指定するプロパティを以下に示します。

プロパティ	説明
double Width { get; set; }	コントロールの幅を設定します。デフォルト値は NaN です。
double Height { get; set; }	コントロールの高さを設定します。デフォルト値は NaN です。
double MinWidth { get; set; }	コントロールの最小の幅を設定します。デフォルト値は 0 です。

double MinHeight { get; set; }	コントロールの最少の高さを設定します。デフォルト値は 0 です。
double MaxWidth { get; set; }	コントロールの最大の幅を設定します。デフォルト値は PositiveInfinity です。
double MinHeight { get; set; }	コントロールの最大の高さを設定します。デフォルト値は PositiveInfinity です。

XAML で指定する場合は、数字以外に以下のような値を設定できます。

- 10(10px と同じ意味) : ピクセル単位で指定します。
- 10in : インチで指定します。
- 10cm : センチメートルで指定します。
- 10pt : ポイントで指定します。
- Auto(Width と Height のみ) : NaN を設定するのと同じ意味です。

幅と高さを指定すると、可能な限りその大きさに配置されます。コントロールにサイズを指定する場合は、可能な限り Min****や Max****を使って指定することをお勧めします。こうすることで、ローカライズ時に文字が切れたりレイアウトが意図しない形に崩れたりといったことを防ぐことができます。Min****や Max****を指定すると、その範囲内で適切な大きさにコントロールが表示されます。

4.1.12.3. 余白(マージン)の指定

最後に余白(マージン)を指定する Margin プロパティについて紹介します。マージンは名前の通りコントロールの周りに指定したサイズの余白をとります。XAML で指定する場合は、以下のような指定方法があります。

- 5 (数字 1 つだけの場合) : 上下左右に 5px の余白をとります
- 5, 10 (数字 2 つだけの場合) : 左右に 5px、上下に 10px の余白をとります
- 5, 10, 15, 20 (全て指定する場合) : 左に 5px、上に 10px、右に 15px、下に 20px の余白をとります

4.1.12.4. サンプルプログラム

ここで紹介したプロパティの動作を見るためのサンプルを示します。このサンプルは、3 x 3 で行と列のサイズを*に指定した Grid コントロールの各マスにコントロールを配置しています。

- 1 行目 : HorizontalAlignment プロパティと VerticalAlignment プロパティの動作確認
- 2 行目 : Width プロパティなどのサイズを指定するプロパティの動作確認
- 3 行目 : マージンを指定するプロパティの動作確認

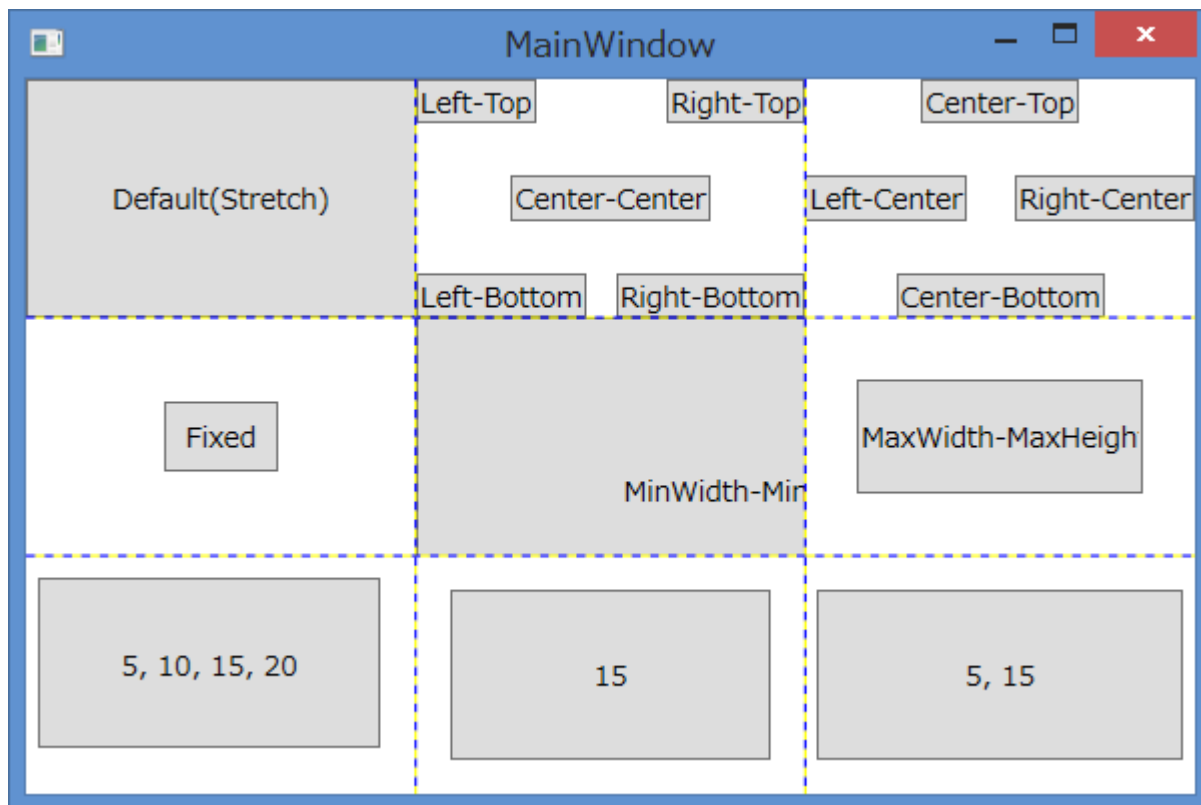
XAML を以下に示します。

```

<Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <!-- HorizontalAlignment VerticalAlignment に関する設定 -->
    <Button Grid.Row="0" Grid.Column="0" Content="Default(Stretch)" />
    <Button Grid.Row="0" Grid.Column="1" Content="Left-
Bottom" HorizontalAlignment="Left" VerticalAlignment="Bottom" />
    <Button Grid.Row="0" Grid.Column="1" Content="Left-
Top" HorizontalAlignment="Left" VerticalAlignment="Top" />
    <Button Grid.Row="0" Grid.Column="1" Content="Right-
Top" HorizontalAlignment="Right" VerticalAlignment="Top" />
    <Button Grid.Row="0" Grid.Column="1" Content="Right-
Bottom" HorizontalAlignment="Right" VerticalAlignment="Bottom" />
    <Button Grid.Row="0" Grid.Column="1" Content="Center-
Center" HorizontalAlignment="Center" VerticalAlignment="Center" />
    <Button Grid.Row="0" Grid.Column="2" Content="Left-
Center" HorizontalAlignment="Left" VerticalAlignment="Center" />
    <Button Grid.Row="0" Grid.Column="2" Content="Right-
Center" HorizontalAlignment="Right" VerticalAlignment="Center" />
    <Button Grid.Row="0" Grid.Column="2" Content="Center-
Top" HorizontalAlignment="Center" VerticalAlignment="Top" />
    <Button Grid.Row="0" Grid.Column="2" Content="Center-
Bottom" HorizontalAlignment="Center" VerticalAlignment="Bottom" />
    <!-- サイズの設定 -->
    <Button Grid.Row="1" Grid.Column="0" Content="Fixed" Width="50" Height="30" />
    <Button Grid.Row="1" Grid.Column="1" Content="MinWidth-MinHeight" MinWidth="300" MinHeight="150" />
    <Button Grid.Row="1" Grid.Column="2" Content="MaxWidth-MaxHeight" MaxWidth="125" MaxHeight="50" />
    <!-- 余白の設定 -->
    <Button Grid.Row="2" Grid.Column="0" Content="5, 10, 15, 20" Margin="5, 10, 15, 20" />
    <Button Grid.Row="2" Grid.Column="1" Content="15" Margin="15" />
    <Button Grid.Row="2" Grid.Column="2" Content="5, 15" Margin="5, 15" />
</Grid>

```

この XAML を記述した Window を表示すると以下ようになります。



MinWidth プロパティと MinHeight プロパティの例は、わかりにくいかもしれませんが Grid のセルの大きさがボタンの最小の大きさよりも小さいためボタンがはみ出しています。

4.1.12.5. WPF におけるピクセルについて

WPF でのピクセルは、物理的なピクセルではなくデバイス非依存ピクセルになります。デバイス非依存ピクセルとは、dpi(1 インチあたりのドット数)にかかわらず 1 ピクセルが 1/96 インチになります。このため、WPF では 72dpi のモニタでも 19,200dpi のプリンタでも同じサイズで描画が可能になっています。

4.2. ボタン

レイアウトコントロールの次は、もっとも基本的なユーザーがアクションを起こすときの接点となるボタンについて説明します。ボタンのコントロールには、一般的な Button コントロールと、ユーザーがボタンを押している間 Click イベントを繰り返し発行する RepeatButton の 2 種類があります。

4.2.1. Button コントロール

Button コントロールは、ユーザーのクリックやタップといった操作に対して Click イベントを発行するコントロールです。Click イベントは以下のように定義されています。

```
public event RoutedEventHandler Click;
```

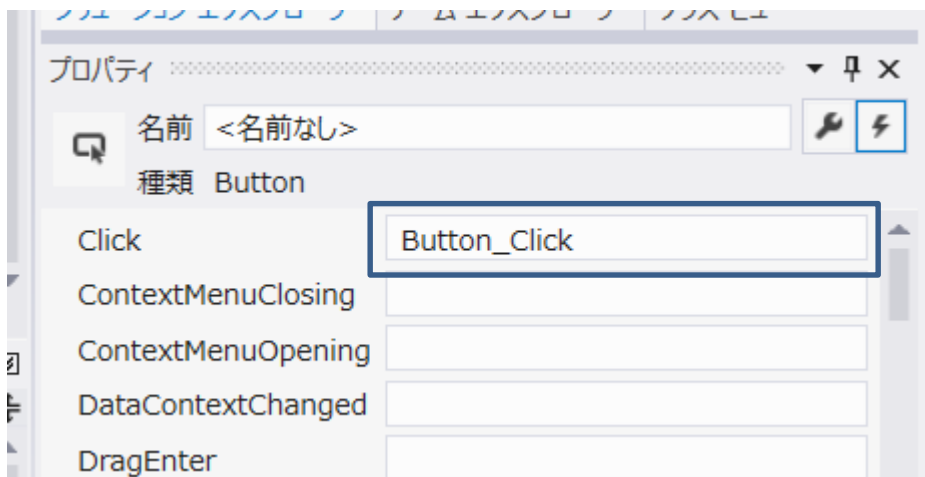
RoutedEventHandler は、以下のように定義されています。RoutedEventArgs についての詳細は WPF でのイベントの解説時に説明します。

```
public delegate void RoutedEventHandler(  
    Object sender,  
    RoutedEventArgs e  
)
```

Click イベントを講読して、ボタンがクリックされたときに Content プロパティにクリック回数を表示するプログラムを作成してみます。まず、Window にボタンを置きます。

```
<Window x:Class="ButtonSample.MainWindow"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Title="MainWindow" Height="70" Width="210">  
    <Grid>  
        <Button Content="0 回" />  
    </Grid>  
</Window>
```

ボタンの Click イベントにイベントハンドラを登録するにはプロパティウィンドウで、⚡マークを選択してイベント名の横のテキストボックスにイベントハンドラ名を入力することで対応付けができます。コードビハインドに、対応するメソッドが無い場合は自動的に生成されます。

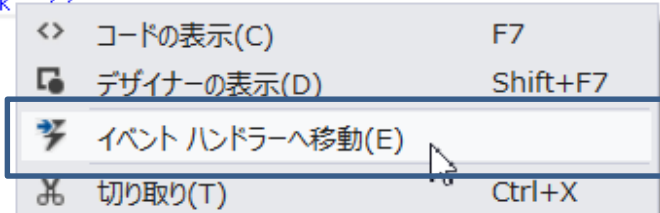


または、XAML で Click イベントに任意のメソッド名を入力して右クリックメニューの「イベント ハンドラーへ移動」を選択することでもイベントハンドラを作成できます。

```

    Title="MainWindow" Height="68.109" Width="211.691">
<Grid>
    <Button Content="0回" Click="Button_Click" />
</Grid>
</Window>

```



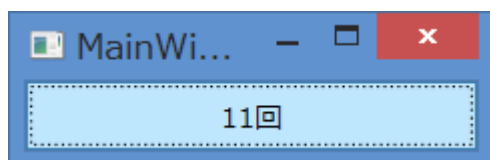
ボタンにイベントハンドラを関連付けたら、コードビハインドのイベントハンドラのメソッドに以下のようなコードを記述します。

```

// クリック回数
private int count = 0;
private void Button_Click(object sender, RoutedEventArgs e)
{
    // sender 経由でクリックイベントを発生させたボタンを取得
    var button = (Button)sender;
    // ボタンの表示を更新
    button.Content = string.Format("{0}回", ++count);
}

```

実行してボタンを 11 回押した結果を以下に示します。



4.2.2. RepeatButton コントロール

RepeatButton コントロールは、ボタンの上でマウスが押されている間、一定間隔で Click イベントを発行するボタンです。Click イベントの発行間隔は、以下のプロパティで設定します。

プロパティ	説明
int Delay { get; set; }	ボタンが押されている間に Click イベントの繰り返しが開始するまでに待つ時間（ミリ秒）を指定します。
int Interval { get; set; }	Click イベントの繰り返しの感覚（ミリ秒）を指定します。

例として、1 秒間クリックされっぱなしの場合に 2 秒間隔で Click イベントを発行する場合の XAML は以下のようになります。

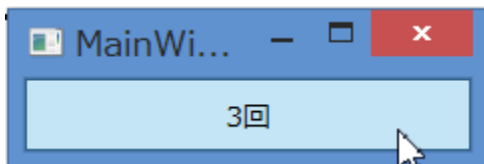
```
<Window x:Class="RepeatButtonSample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="70" Width="210">
    <Grid>
        <RepeatButton Content="0 回" Click="Button_Click"
                      Delay="1000"
                      Interval="2000" />

    </Grid>
</Window>
```

Button コントロールで示したサンプルと同じように Click イベントの処理で、イベントの発生回数をボタンに表示するロジックのコード例を以下に示します。sender を RepeatButton にキャストしている箇所が Button コントロールのコード例と異なります。

```
// クリック回数
private int count = 0;
private void Button_Click(object sender, RoutedEventArgs e)
{
    // sender 経由でクリックイベントを発生させたボタンを取得
    var button = (RepeatButton)sender;
    // ボタンの表示を更新
    button.Content = string.Format("{0} 回", ++count);
}
```

実行してボタンを押せばなしにした画面を以下に示します。



4.3. データ表示

ここでは、データの表示に使用する 3 つのコントロールについて紹介します。一般的な業務アプリケーションでは、最もよく使用するコントロールになるので、基本的な使い方をしっかりと覚えておきましょう。

4.3.1. DataGrid コントロール

WPF で表形式のデータを表示するためのコントロールです。WPF の柔軟なレイアウトシステムとテンプレートを使うことで以下のような表示を行うことができます。



WPF の DataGrid は行と列からなる格子状のエリアに、ItemsSource プロパティに設定されたコレクションの 1 要素を 1 行として表示します。1 行を表示するときに、予め定義した列の表示の設定に従いセルにデータを表示していきます。また、DataGrid にはデフォルトで表示する要素の型のプロパティから自動的に列を生成する機能があります。

4.3.1.1. 列の自動生成機能

ここでは、DataGrid の列の自動生成機能を使ってデータの表示を行います。WPF アプリケーションのプロジェクトの Window に以下のように DataGrid を置きます。

```
<Window x:Class="DataGridSample02.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>
        <DataGrid Name="dataGrid" />
    </Grid>
</Window>
```

そして、DataGrid に表示するためのデータを格納する Person クラスを作成します。

```

namespace DataGridSample02
{
    // 性別
    public enum Gender
    {
        None,
        Men,
        Women
    }

    // DataGridViewに表示するデータ
    public class Person
    {
        public string Name { get; set; }
        public Gender Gender { get; set; }
        public int Age { get; set; }
        public bool AuthMember { get; set; }
    }
}

```

string 型、enum 型、int 型、bool 型のプロパティを持っています。このクラスのリストを MainWindow クラスのコンストラクタで設定します。

```

public MainWindow()
{
    InitializeComponent();
    // 適当なデータ 100 件生成する
    var data = new ObservableCollection<Person>(
        Enumerable.Range(1, 100).Select(i => new Person
        {
            Name = "田中 太郎" + i,
            Gender = i % 2 == 0 ? Gender.Men : Gender.Women,
            Age = 20 + i % 50,
            AuthMember = i % 5 == 0
        }));
    // DataGridView に設定する
    this.dataGrid.ItemsSource = data;
}

```

ここで使用している ObservableCollection<T> 型は、DataGridView などの WPF のリストを表示するコントロールと、要素の追加・削除を同期する INotifyCollectionChanged インターフェースを実装したクラスです。性能の問題など特別な理由がない限りこのクラスを利用すると便利です。

このプログラムを実行すると、以下のような Window が表示されます。

MainWindow

Name	Gender	Age	AuthMember	
田中 太郎1	Women	21	<input type="checkbox"/>	
田中 太郎2	Men	22	<input type="checkbox"/>	
田中 太郎3	Women	23	<input type="checkbox"/>	
田中 太郎4	Men	24	<input type="checkbox"/>	
田中 太郎5	Women	25	<input checked="" type="checkbox"/>	
田中 太郎6	Men	26	<input type="checkbox"/>	
田中 太郎7	Women ▾	27	<input type="checkbox"/>	
田中 太郎8	None	28	<input type="checkbox"/>	
田中 太郎9	Men	29	<input type="checkbox"/>	
田中 太郎10	Women	30	<input checked="" type="checkbox"/>	
田中 太郎11	Women	31	<input type="checkbox"/>	
田中 太郎12	Men	32	<input type="checkbox"/>	
田中 太郎13	Women	33	<input type="checkbox"/>	
田中 太郎14	Men	34	<input type="checkbox"/>	

プロパティに対応した列が生成されていることが確認できます。string 型と int 型の列は通常のテキストを表示する列が生成され、enum 型はドロップダウンリストを表示する列が生成され、bool 型にはチェックボックスを表示する列が生成されます。単にデータを表示するだけなら、このような自動生成の機能を利用することができます。

4.3.1.2. 自動生成のカスタマイズ

DataGrid の列の自動生成は、列を自動生成するタイミングで発生する AutoGeneratingColumn イベントである程度カスタマイズすることが出来ます。AutoGeneratingColumn イベント引数の DataGridAutoGeneratingColumnEventArgs クラスには以下のようなプロパティが定義されています。

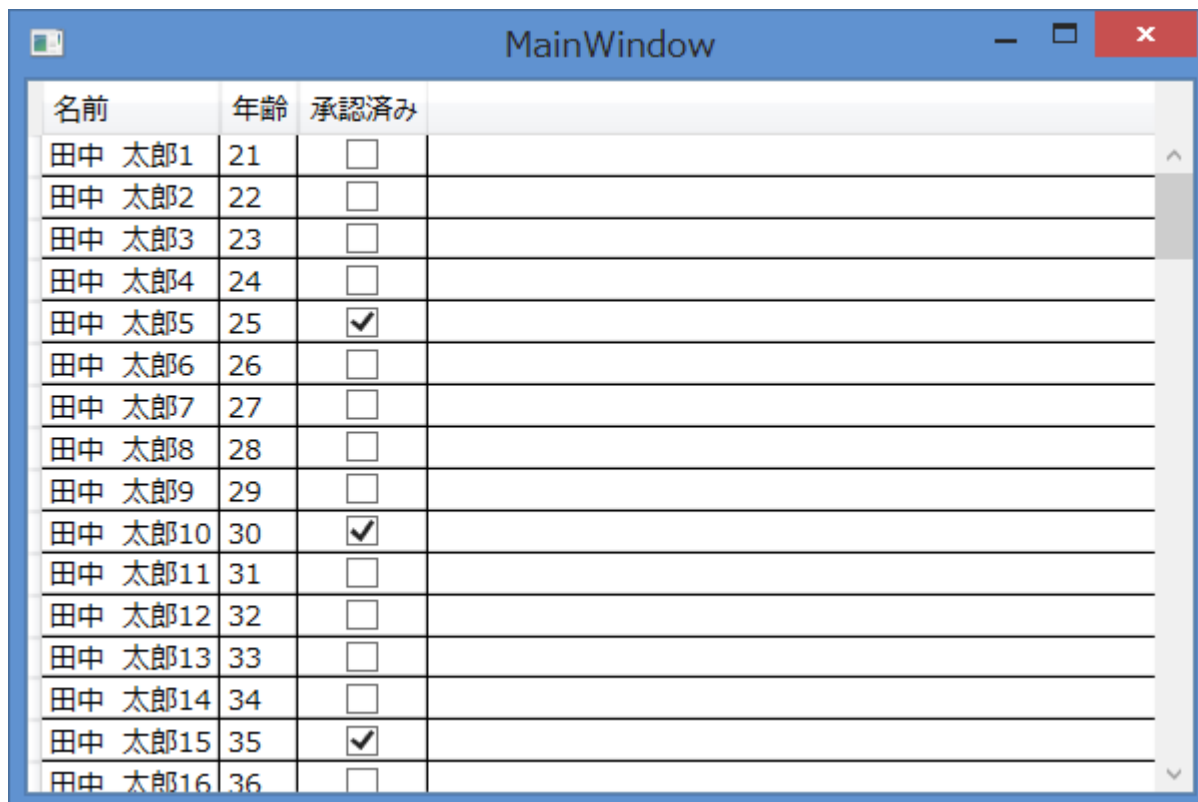
プロパティ	説明
string PropertyName { get; }	列を生成するプロパティの名前を取得します。
bool Cancel { get; set; }	列の生成をキャンセルする場合は true を設定します。
DataGridColumn Column { get; set; }	自動生成される列を取得します。また、このプロパティに自分で作成した DataGridColumn の派生クラス (http://msdn.microsoft.com/ja-

jp/library/vstudio/system.windows.controls.datagridcolumn(vs.110).aspx) を設定することで、自動生成するカラムを置き換えることができます。

自動生成のカスタマイズの例として、先ほどの自動生成のプログラムの DataGrid に AutoGeneratingColumn イベントハンドラを作成し下記のように記述しました。

```
private void dataGrid_AutoGeneratingColumn(object sender, DataGridAutoGeneratingColumnEventArgs e)
{
    // プロパティ名をもとに自動生成する列をカスタマイズします
    switch (e.PropertyName)
    {
        case "Name":
            // Name 列は最初に表示してヘッダーを名前にする
            e.Column.Header = "名前";
            e.Column.DisplayIndex = 0;
            break;
        case "Age":
            // Age プロパティは 1 番目に表示してヘッダーを年齢にする
            e.Column.Header = "年齢";
            e.Column.DisplayIndex = 1;
            break;
        case "Gender":
            // Gender プロパティは表示しない
            e.Cancel = true;
            break;
        case "AuthMember":
            // AuthMember プロパティは 2 番目に表示してヘッダーを承認済みにする
            e.Column.Header = "承認済み";
            e.Column.DisplayIndex = 2;
            break;
        default:
            throw new InvalidOperationException();
    }
}
```

上記の変更をしてプログラムを実行すると下図のような結果になります。



名前	年齢	承認済み	
田中 太郎1	21	<input type="checkbox"/>	
田中 太郎2	22	<input type="checkbox"/>	
田中 太郎3	23	<input type="checkbox"/>	
田中 太郎4	24	<input type="checkbox"/>	
田中 太郎5	25	<input checked="" type="checkbox"/>	
田中 太郎6	26	<input type="checkbox"/>	
田中 太郎7	27	<input type="checkbox"/>	
田中 太郎8	28	<input type="checkbox"/>	
田中 太郎9	29	<input type="checkbox"/>	
田中 太郎10	30	<input checked="" type="checkbox"/>	
田中 太郎11	31	<input type="checkbox"/>	
田中 太郎12	32	<input type="checkbox"/>	
田中 太郎13	33	<input type="checkbox"/>	
田中 太郎14	34	<input type="checkbox"/>	
田中 太郎15	35	<input checked="" type="checkbox"/>	
田中 太郎16	36	<input type="checkbox"/>	

カラムヘッダーがカスタマイズできていることと、Gender プロパティの列が生成されていないことが確認できます。

4.3.1.3. DataGrid で使用可能な列

DataGrid では、列の自動生成機能とイベントのカスタマイズを使うことで簡単にデータを表示することができますが、単純なケースへの対応や、属性を利用した汎用的なデータの表示機能を開発するケース以外では、これから紹介する、自分で列を定義する方法が一般的です。DataGrid で使用可能な列の中で代表的なものを以下に示します。

クラス名	説明
DataGridTextBoxColumn	Binding プロパティで指定したデータをテキストとして表示します。編集モードでは、テキストボックスを表示します。
DataGridCheckBoxColumn	Binding プロパティで指定したデータをチェックボックスとして表示します。
DataGridComboBoxColumn	ItemsSource プロパティに設定したコレクションを選択肢として表示するコンボボックスをセルに表示します。DisplayMemberPath プロパティ

	<p>や <code>SelectedValuePath</code> プロパティでコンボボックスに表示するプロパティと、選択した値として扱うプロパティを設定します。</p> <p>表示するデータは、<code>SelectedValueBinding</code> プロパティで <code>ItemsSource</code> プロパティで設定したコレクションで選択中の要素から <code>SelectedValuePath</code> で指定したプロパティの値、<code>SelectedItemBinding</code> プロパティで <code>ItemsSource</code> プロパティに設定したコレクションで選択中のものに対応づけができます。</p>
DataGridTemplateColumn	<p><code>DataTemplate</code> を使って表示内容を自由にカスタマイズできます。</p> <p><code>DataGrid</code> で指定できる列の中で一番柔軟にセル内の表示をカスタマイズできます。</p>

`DataGrid` の列は、以下のプロパティを使うことでヘッダーの内容をカスタマイズできます。

プロパティ	説明
object Header { get; set; }	列のヘッダーの値を取得または設定します。
DataTemplate HeaderTemplate { get; set; }	列のヘッダーの値を表示するためのテンプレートを取得または設定します。
Style HeaderStyle { get; set; }	列のヘッダーを表示するときに使用するスタイルを取得または設定します。スタイルを適用する型は <code>DataGridHeaderColumn</code> クラスです。

4.3.1.4. DataGrid の列の使用例

`DataGrid` の列を使って `DataGrid` にデータを表示します。まず、画面に `DataGrid` を置きます。今回は `DataGrid` の自動生成列は使用しないため、`AutoGeneratedColumns` プロパティを `False` にして自動生成機能を無効化しています。

```

<Window x:Class="DataGridSample03.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:DataGridSample03"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <DataGrid Name="dataGrid" AutoGenerateColumns="False">
            </DataGrid>
        </Grid>
    </Window>

```

次に、DataGrid に表示するためのクラスを定義します。以下のように string 型と列挙型と bool 型のプロパティを持つ Person クラスを作成します。

```

// 性別を表す列挙型
public enum Gender
{
    None,
    Men,
    Women
}

// DataGrid に表示するデータ
public class Person
{
    // 名前
    public string Name { get; set; }
    // 性別
    public Gender Gender { get; set; }
    // 認証済みユーザーかどうか
    public bool AuthMember { get; set; }
}

```

MainWindow のコンストラクタで、Person クラスのリストを DataGrid の ItemsSource プロパティに設定します。

```
// 適当なデータ 100 件生成する
var data = new ObservableCollection<Person>(
    Enumerable.Range(1, 100).Select(i => new Person
    {
        Name = "田中 太郎" + i,
        Gender = i % 2 == 0 ? Gender.Men : Gender.Women,
        AuthMember = i % 5 == 0
    }));
// DataGrid に設定する
this.dataGrid.ItemsSource = data;
```

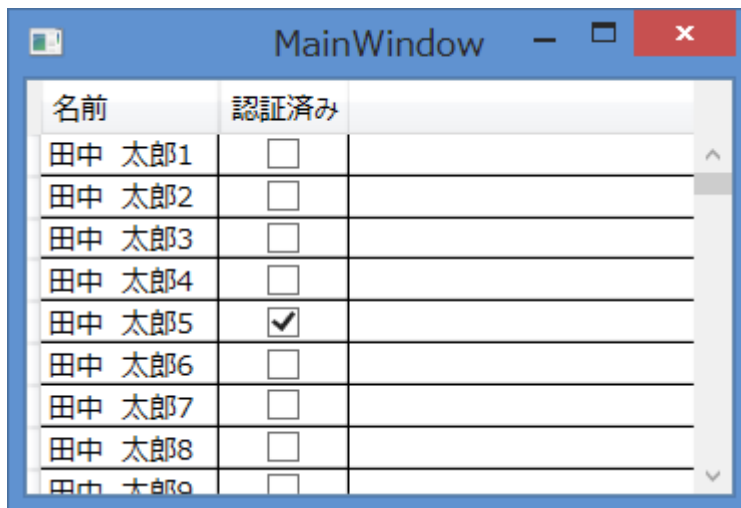
これで、表示データの準備が出来たので列を定義してデータを表示していきます。

DataGridTextBoxColumn と DataGridCheckBoxColumn の使用

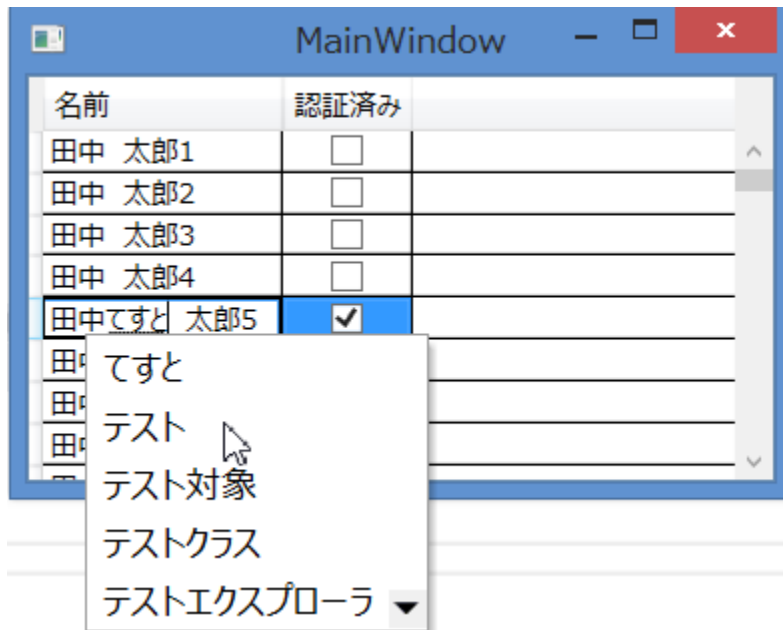
DataGrid に列を定義します。DataGrid の列は、Columns プロパティに設定します。DataGridTextBoxColumn を Name プロパティに、DataGridCheckBoxColumn を AuthMember プロパティにバインドしています。

```
<DataGrid Name="dataGrid" AutoGenerateColumns="False">
    <DataGrid.Columns>
        <DataGridTextBoxColumn Header="名前" Binding="{Binding Name}" />
        <DataGridCheckBoxColumn Header="認証済み" Binding="{Binding AuthMember}" />
    </DataGrid.Columns>
</DataGrid>
```

実行すると、以下のようになります。



セルを選択して F2 キーを押すか、選択中のセルをクリックすることでセルの編集が可能です。下図は、名前の列を編集している画像になります。



DataGridComboBoxColumn の使用

次に、Gender プロパティを DataGridComboBoxColumn を使って表示します。自動生成では、enum の値がそのまま表示されていたので、ここでは日本語ラベルを表示できるようにします。まず、ComboBox に表示するためのデータを持つクラスを作成します。ラベル用の文字列と、対応する Gender の値を持つ単純なクラスです。

```
namespace DataGridSample03
{
    // Gender を ComboBox に表示するためのクラス
    public class GenderComboBoxItem
    {
        // 表示用のラベル
        public string Label { get; set; }
        // 値
        public Gender Value { get; set; }
    }
}
```

このクラスの配列を DataGridComboBoxColumn の ItemsSource プロパティに設定して DisplayMemberPath と SelectedValuePath に Label と Value を設定します。選択した要素の値と Person クラスの Gender プロパティを紐づけたいので SelectedValueBinding プロパティに Gender とのバインディングを設定します。

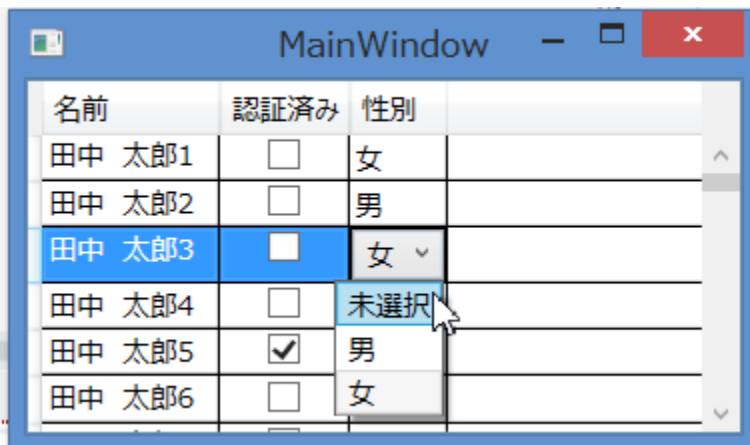
これまでに未登場の x:Array タグは Type 属性で指定したものの配列を XAML で定義するためのものです。

```

<DataGridComboBoxColumn Header="性別"
    SelectedValueBinding="{Binding Gender}"
    DisplayMemberPath="Label"
    SelectedValuePath="Value">
    <DataGridComboBoxColumn.ItemsSource>
        <x:Array Type="{x:Type local:GenderComboBoxItem}">
            <local:GenderComboBoxItem Label="未選択" Value="None" />
            <local:GenderComboBoxItem Label="男" Value="Men" />
            <local:GenderComboBoxItem Label="女" Value="Women" />
        </x:Array>
    </DataGridComboBoxColumn.ItemsSource>
</DataGridComboBoxColumn>

```

この定義を追加した DataGrid の表示を以下に示します。表示データや、編集中のドロップダウンのデータが日本語になっていることが確認できます。



DataGridTemplateColumn の使用

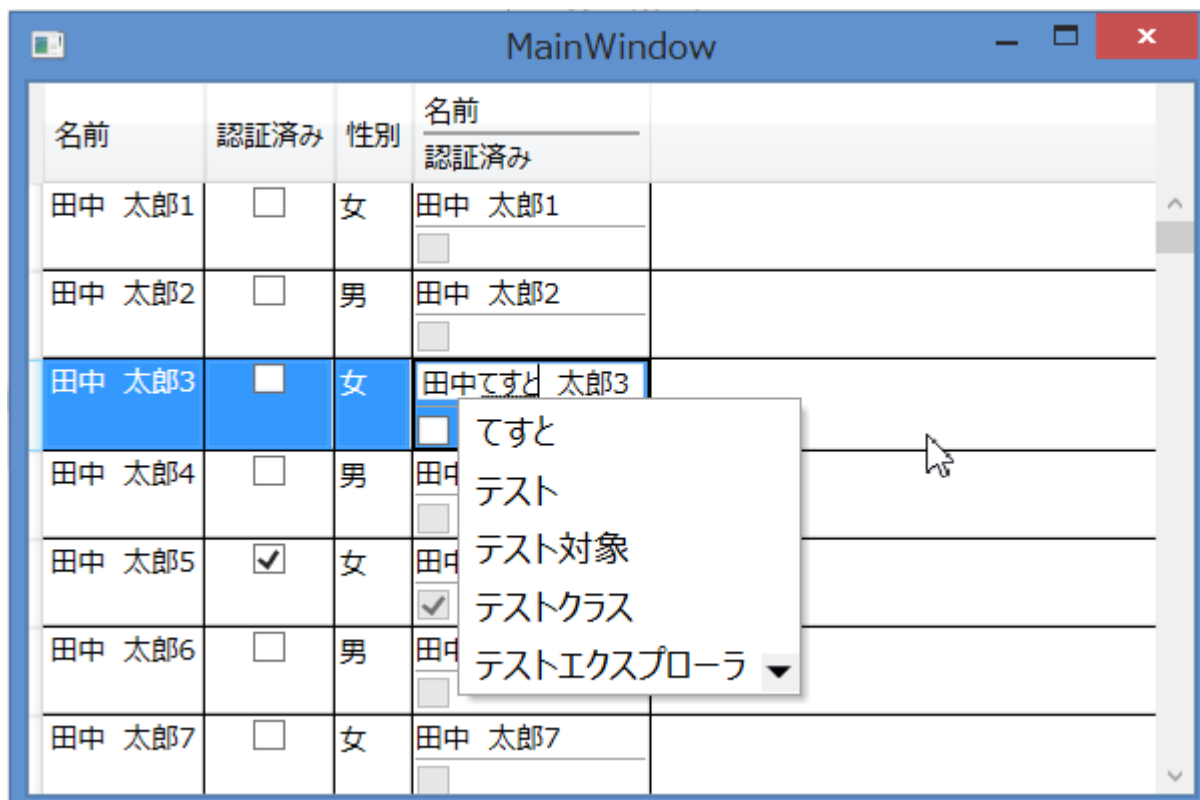
次に、セルの表示を自由にカスタマイズできる DataGridTemplateColumn を使用します。ヘッダーとセルのテンプレートに StackPanel を使って複数の項目を表示しています。HeaderStyle を使って HeaderTemplate で指定した表示内容を水平方向いっぱい指定しています。この設定がないと、ヘッダーの表示が左寄せになってしまいます。CellTemplate には表示を、CellEditingTemplate には編集用の UI を設定します。どのテンプレートも StackPanel で縦並びに複数要素を表示するように設定しています。各要素の間は Separator コントロールを使って罫線を表示しています。


```

<DataGridTemplateColumn>
  <DataGridTemplateColumn.HeaderStyle>
    <Style TargetType="DataGridColumnHeader">
      <Setter Property="HorizontalAlignment" Value="Stretch" />
    </Style>
  </DataGridTemplateColumn.HeaderStyle>
  <DataGridTemplateColumn.HeaderTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="名前" />
        <Separator />
        <TextBlock Text="認証済み" />
      </StackPanel>
    </DataTemplate>
  </DataGridTemplateColumn.HeaderTemplate>
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding Name}" />
        <Separator />
        <CheckBox IsEnabled="False" IsChecked="{Binding AuthMember}" />
      </StackPanel>
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
  <DataGridTemplateColumn.CellEditingTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBox Text="{Binding Name}" />
        <Separator />
        <CheckBox IsChecked="{Binding AuthMember}" />
      </StackPanel>
    </DataTemplate>
  </DataGridTemplateColumn.CellEditingTemplate>
</DataGridTemplateColumn>

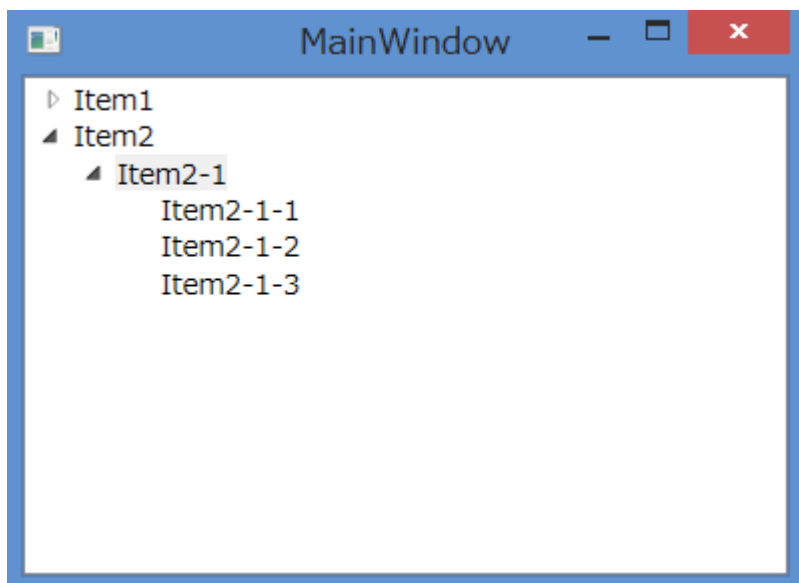
```

実行結果は以下のようになります。テンプレートを使うことで 1 つのセルに 2 行のデータを表示できていることが確認できます。



4.3.2. TreeView コントロール

TreeView コントロールは、Windows のエクスプローラーの左側のような入れ子構造のデータを表示するのに適したコントロールです。TreeView コントロールの見た目を以下に示します。



4.3.2.1. 基本的な使用方法

TreeView コントロールは、Items プロパティに TreeViewItem コントロールを設定することで木構造のデータを表示します。TreeViewItem コントロールは、Header プロパティでツリーに表示する要素を指定して、Items プロパティで TreeViewItem コントロールを子として格納します。TreeView コントロールも TreeViewItem コントロールも Items プロパティがコンテンツプロパティなので、シンプルに木構造を XAML で定義できます。以下に、上図の TreeView コントロールの XAML の定義を示します。

```
<TreeView>
  <TreeViewItem Header="Item1">
    <TreeViewItem Header="Item1-1">
      <TreeViewItem Header="Item1-1-1" />
      <TreeViewItem Header="Item1-1-2" />
      <TreeViewItem Header="Item1-1-3" />
    </TreeViewItem>
    <TreeViewItem Header="Item1-2">
      <TreeViewItem Header="Item1-2-1" />
      <TreeViewItem Header="Item1-2-2" />
    </TreeViewItem>
  </TreeViewItem>
  <TreeViewItem Header="Item2" IsExpanded="True">
    <TreeViewItem Header="Item2-1" IsExpanded="True" IsSelected="True">
      <TreeViewItem Header="Item2-1-1" />
      <TreeViewItem Header="Item2-1-2" />
      <TreeViewItem Header="Item2-1-3" />
    </TreeViewItem>
  </TreeViewItem>
</TreeView>
```

上記 XAML にあるように、IsExpanded プロパティでツリーが展開されているかどうか。IsSelected プロパティで選択中のノードを指定します。

ここまでに出てきた TreeView コントロールのプロパティを以下に示します。

プロパティ名	説明
ItemCollection Items { get; }	TreeView コントロールに表示する要素を格納するコレクションを取得します。

ここまでに出てきた TreeViewItem コントロールのプロパティを以下に示します。

プロパティ名	説明
--------	----

object Header { get; set; }	ツリーに表示する要素を取得または設定します。コンテンツモデルで紹介した表示ロジックによって要素が表示されます。
ItemsCollection Items { get; }	TreeViewItem コントロールの子要素を格納するコレクションを取得します。
bool IsExpanded { get; set; }	要素が展開されているかどうかを取得または設定します。
bool IsSelected { get; set; }	要素が選択されているかどうかを取得または設定します。

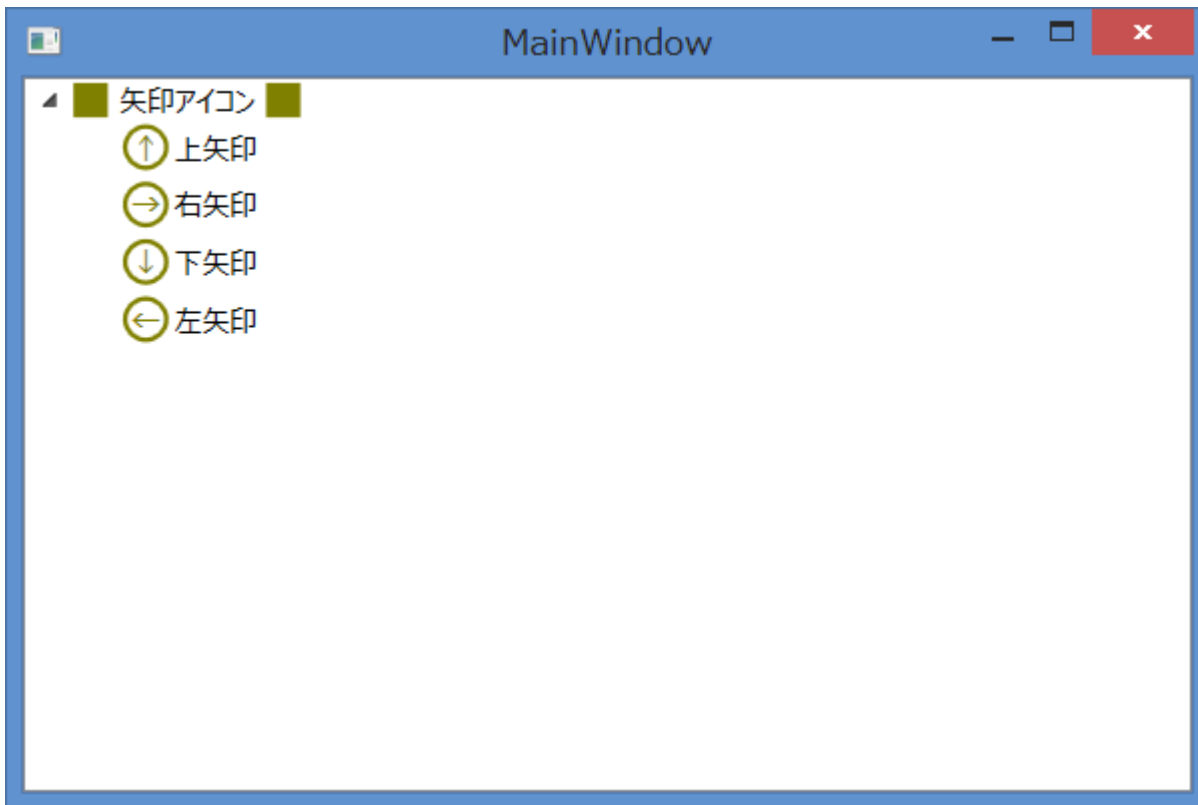
これらのプロパティを組み合わせて使うことで、静的なツリー場合は簡単に Windows Forms では表現が難しかった表現も簡単に表示することが出来ます。XAML を以下に示します。

```

<TreeView>
  <TreeViewItem IsExpanded="True">
    <TreeViewItem.Header>
      <StackPanel Orientation="Horizontal">
        <Rectangle Fill="Olive" Width="15" Height="15" />
        <TextBlock Text="矢印アイコン" Margin="5,0" />
        <Rectangle Fill="Olive" Width="15" Height="15" />
      </StackPanel>
    </TreeViewItem.Header>
  </TreeViewItem>
  <TreeViewItem>
    <TreeViewItem.Header>
      <StackPanel Orientation="Horizontal">
        <Grid Margin="2.5">
          <Ellipse Width="20" Height="20" Stroke="Olive" StrokeThickness="2" />
          <TextBlock Text="↑" HorizontalAlignment="Center"
            VerticalAlignment="Center" FontWeight="Bold" Foreground="Olive" />
        </Grid>
        <TextBlock Text="上矢印" VerticalAlignment="Center" />
      </StackPanel>
    </TreeViewItem.Header>
  </TreeViewItem>
  <TreeViewItem>
    <TreeViewItem.Header>
      <StackPanel Orientation="Horizontal">
        <Grid Margin="2.5">
          <Ellipse Width="20" Height="20" Stroke="Olive" StrokeThickness="2" />
          <TextBlock Text="→" HorizontalAlignment="Center"
            VerticalAlignment="Center" FontWeight="Bold" Foreground="Olive" />
        </Grid>
        <TextBlock Text="右矢印" VerticalAlignment="Center" />
      </StackPanel>
    </TreeViewItem.Header>
  </TreeViewItem>
  ...省略...
</TreeViewItem>
</TreeView>

```

TreeViewItem コントロールの Header プロパティに、StackPanel コントロールや Grid コントロールを使って複数のコントロールをレイアウトしています。Rectangle や Ellipse は、まだ紹介していませんが WPF で基本的な図形を表す要素です。実行結果を以下に示します。



WPF の強力な機能の 1 つである要素の合成と、TreeView コントロールを使うことで、オーナードローなどの特別な記述を行わなくても、凝った表示ができることがわかります。

4.3.2.2. TreeView の ItemTemplate

TreeView コントロールも DataGrid コントロールと同様に ItemsSource プロパティにコレクションを設定することで、任意の型のコレクションのデータを表示できます。TreeView コントロールの ItemTemplate には、木構造のデータを扱うために DataTemplate を拡張した HierarchicalDataTemplate を使用します。

HierarchicalDataTemplate は、通常の DataTemplate と同様にデータの見た目を定義するために使います。

DataTemplate と異なる点は、ItemsSource プロパティに、現在表示している要素の子にあたるものを ItemsSource プロパティに設定する点です。HierarchicalDataTemplate によるデータの見た目の定義と、ItemsSource プロパティにもとづいて TreeView コントロールの TreeViewItem コントロール組み立てられます。

HierarchicalDataTemplate の動作を確認するために、以下のような木構造をもった Person クラスを TreeView コントロールに表示してみます。

```
using System.Collections.Generic;

namespace TreeViewSample03
{
    public class Person
    {
        public string Name { get; set; }
        public List<Person> Children { get; set; }
    }
}
```

画面に、Person クラスを表示するための TreeView コントロールを置きます。

```
<TreeView Name="treeView">
</TreeView>
```

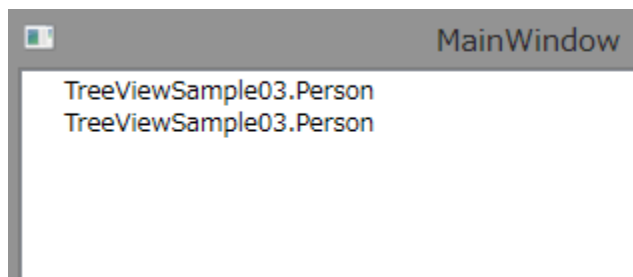
画面のコンストラクタで、この TreeView コントロールの ItemsSource プロパティに Person クラスの List を設定します。

```

public MainWindow()
{
    InitializeComponent();
    this.treeView.ItemsSource = new List<Person>
    {
        new Person
        {
            Name = "田中 太郎",
            Children = new List<Person>
            {
                new Person { Name = "田中 花子" },
                new Person { Name = "田中 一郎" },
                new Person
                {
                    Name = "木村 貫太郎",
                    Children = new List<Person>
                    {
                        new Person { Name = "木村 はな" },
                        new Person { Name = "木村 梅" },
                    }
                }
            }
        },
        new Person
        {
            Name = "田中 次郎",
            Children = new List<Person>
            {
                new Person { Name = "田中 三郎" }
            }
        }
    };
}

```

この状態で画面を表示すると、以下のように Person クラスを ToString した結果が 2 つ TreeView コントロールに表示されます。



Person クラスの Name プロパティが表示されるように TreeView コントロールの ItemTemplate プロパティを設定します。ここでは、HierarchicalDataTemplate を使用していますが、ItemsSource プロパティにを設定していないので単純に TreeView コントロールに名前が表示されるだけになります。

```
xmlns:local="clr-namespace:TreeViewSample03"

<TreeView Name="treeView">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate DataType="local:Person">
      <TextBlock Text="{Binding Name}" />
    </HierarchicalDataTemplate>
  </TreeView.ItemTemplate>
</TreeView>
```

MainWindow の属性で xmlns を使って Person クラスのある名前空間を local という名前で参照できるようにして HierarchicalDataTemplate の DataType プロパティで、扱う型が Person クラスであることを指定しています。実行すると以下のような表示になります。



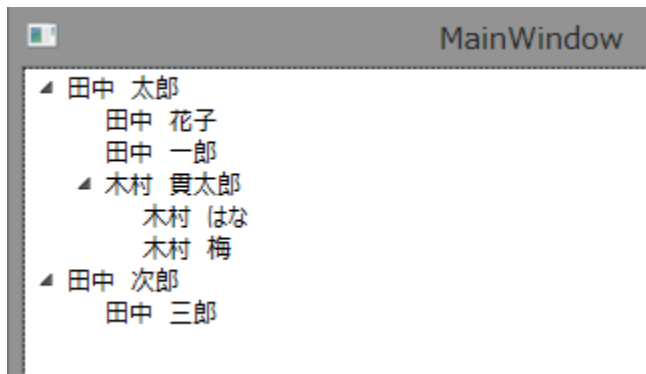
この状態では、ルート要素しか表示されないなので HierarchicalDataTemplate の ItemsSource プロパティに Person クラスの Children プロパティをバインドします。XAML を以下に示します。

```

<TreeView Name="treeView">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate DataType="local:Person" ItemsSource="{Binding Children}">
      <TextBlock Text="{Binding Name}" />
    </HierarchicalDataTemplate>
  </TreeView.ItemTemplate>
</TreeView>

```

この状態で実行すると HierarchicalDataTemplate の定義が再帰的に適用されて以下のように表示されます。



4.4. 日付表示および選択

ここでは、日付を表示したり、選択するためのコントロールを説明します。

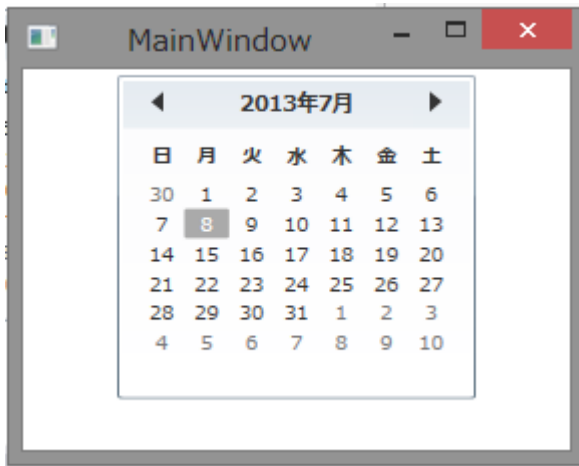
4.4.1. Calendar コントロール

Calendar コントロールは、名前の通りカレンダーを画面に表示してユーザーに日付を選択してもらうためのコントロールです。Calendar コントロールの代表的な機能を以下に示します。

1. 表示内容を 1 か月、1 年、10 年に設定できます。
2. 複数の日付（単一選択と複数選択の組み合わせ）の選択ができます。
3. 選択不可の日付を設定できます。
4. Calendar コントロールに表示される日付の範囲を設定します。
5. 今日の日付を強調表示できます。

4.4.1.1. 基本的な使い方

Calendar コントロールの基本的な使い方を示します。Window に Calendar コントロールを置くと以下のような表示になります。このように 1 月ぶんの日付が表示されます。



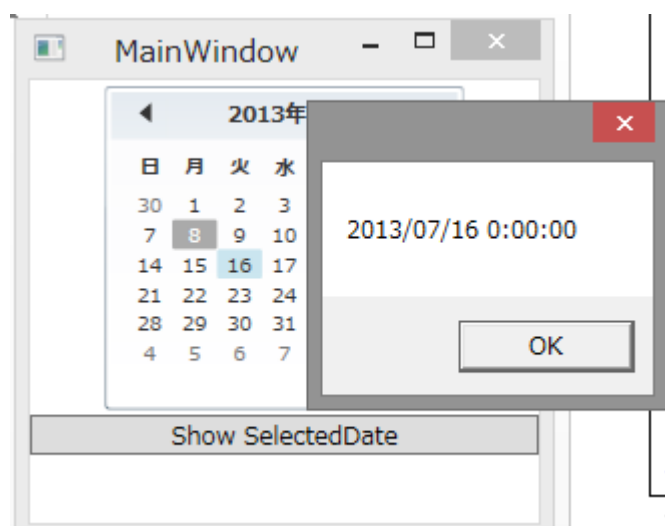
ここにボタンを置いて、選択日をメッセージボックスで表示してみます。コードは以下のようになります。

```
<Window x:Class="CalendarSample01.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.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Calendar x:Name="calendar" />
        <Button Content="Show SelectedDate" Grid.Row="1" Click="Button_Click"/>
    </Grid>
</Window>
```

ボタンのクリックイベントハンドラのコードを以下に示します。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(this.calendar.SelectedDate.ToString());
}
```

プログラムを実行して、日付を選択せずにボタンを押すと以下のように空のメッセージボックスが表示されます。SelectedDate プロパティは、DateTime? 型のため、未選択時はデータを持ちません。カレンダー内の日付を選択して、ボタンを押すと以下のように選択された日が表示されます。



4.4.1.2. 選択方法のカスタマイズ

Calendar コントロールは、以下のプロパティを設定することで、日付の選択を制御することができます。

プロパティ	説明
CalendarSelectionMode SelectionMode { get; set; }	<p>日付の選択方法を設定します。以下の値から設定できます。</p> <ul style="list-style-type: none"> ● MultipleRange 複数の範囲選択ができます。 ● None 何も選択できません。 ● SingleDate 単一の日付を選択できます。 ● SingleRange 単一の日付の範囲を選択できます。
SelectedDatesCollection SelectedDates { get; }	<p>選択された日付のコレクションです。AddRange メソッドを使って指定した日付の範囲を追加することができます。SelectionMode プロパティに MultipleRange か SingleRange が設定されているときのみ、AddRange メソッドが動きます。</p>
DateTime? SelectedDate { get; set; }	<p>単一の選択された日付を取得または設定します。</p>

このほかに、BlackoutDates プロパティを使うことで選択できない日付をカレンダーに設定できます。

プロパティ	説明
CalendarBlackoutDatesCollection BlackoutDates { get; }	<p>選択できない日付のコレクションを返します。Add メソッドで CalendarDateRange を追加することで、選択できない日付の範囲指定を行います。</p> <p>また、AddDatesInPast メソッドを呼ぶことで過去の日付を選択できないようにします。</p>

これらのプロパティを組み合わせることで、柔軟な日付選択の手段をユーザーに提供できます。以下に、今日より前の日付と、翌日から 4 日間は選択できないようにして、それ以外の日付を複数選択可能にする場合のコード例を示します。

まず、ページの XAML です。XAML では、複数選択が可能なことを SelectionMode プロパティで指定しています。

```
<Window x:Class="CalendarSample02.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.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Calendar x:Name="calendar"
            SelectionMode="MultipleRange" />
        <Button Content="Show SelectedDate" Grid.Row="1" Click="Button_Click"/>
    </Grid>
</Window>
```

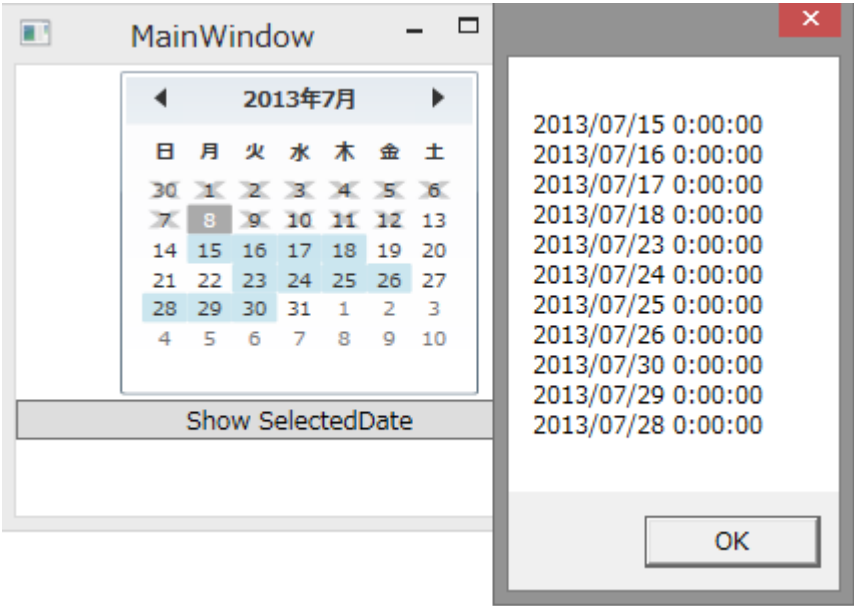
コードビハインドでは、今日より前の日付の無効化と翌日から 4 日間を無効化しています。ボタンのクリックイベントでは SelectionDates プロパティから選択された日付を改行区切りの文字列にして表示しています。

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

namespace CalendarSample02
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            // 今日より前は、選択不可能にする。
            this.calendar.BlackoutDates.AddDatesInPast();
            // 翌日から4日間も選択不可能にする
            this.calendar.BlackoutDates.Add(
                new CalendarDateRange(
                    DateTime.Today.AddDays(1),
                    DateTime.Today.AddDays(4)));
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            // 選択された日付を連結して表示
            var selected = string.Join(Environment.NewLine,
                this.calendar.SelectedDates.Select(d => d.ToString()));
            MessageBox.Show(selected);
        }
    }
}
```

実行して、何日か日付を選択してボタンをクリックした結果を以下に示します。日付を複数選択できていることと、指定した範囲の日付が選択できないように×印がついています。



4.4.1.3. 表示範囲のカスタマイズ

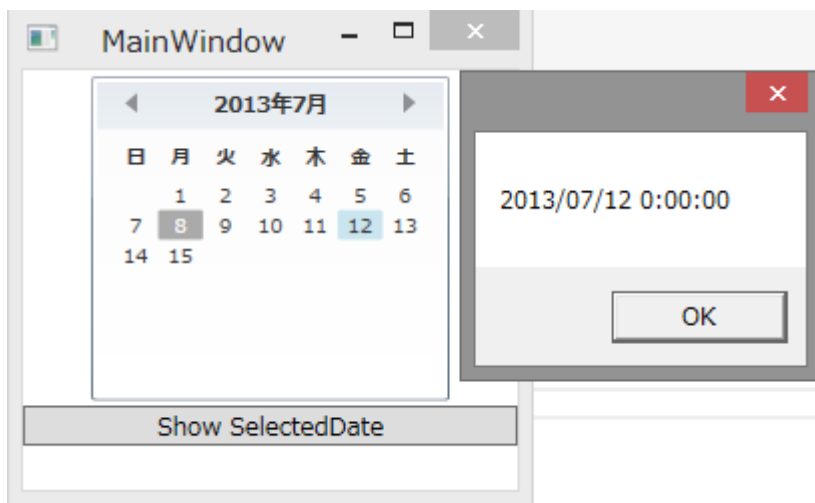
Calendar コントロールの DisplayDateStart プロパティと DisplayDateEnd プロパティを使うことで、カレンダーで選択可能な日付の範囲を指定できます。

プロパティ	説明
DateTime? DisplayDateStart { get; set; }	カレンダーの日付の範囲の最初の日付を取得または設定します。
DateTime? DisplayDateEnd { get; set; }	カレンダーの日付の範囲の最後の日付を取得または設定します。

以下に XAML で DisplayDateStart プロパティと DisplayDateEnd プロパティで 2013/07/01 から 2013/07/15 までの範囲を設定した場合の例を示します。

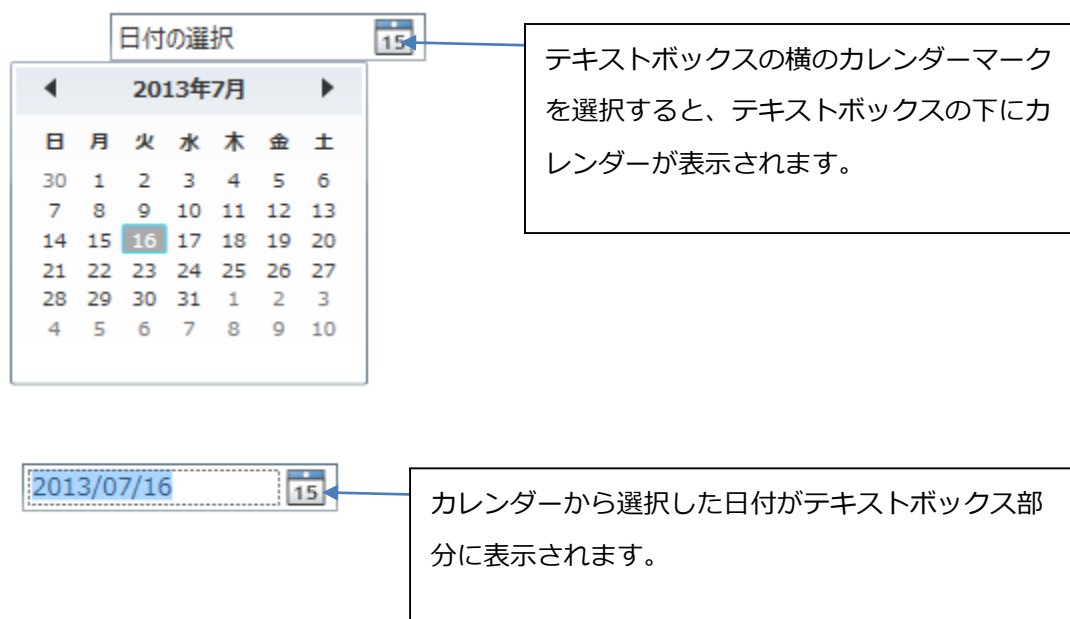
```
<Window x:Class="CalendarSample03.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Calendar x:Name="calendar"
            DisplayDateStart="2013/07/01"
            DisplayDateEnd="2013/07/15"/>
        <Button Content="Show SelectedDate" Grid.Row="1" Click="Button_Click"/>
    </Grid>
</Window>
```

実行すると以下のような結果になります。表示されている日付が指定した範囲に制限されていることが確認できます。



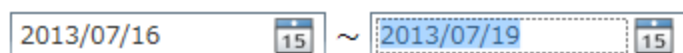
4.4.2. DatePicker コントロール

DatePicker コントロールは、TextBox コントロールと Calendar コントロールを組み合わせたようなコントロールです。ユーザーにキーボードから日付を入力してもらう方法と、カレンダーから日付を入力してもらう方法の二通りの日付の入力方法を提供します。DatePicker コントロールの見た目を以下に示します。



DatePicker コントロールには、Calendar コントロールのように日付の範囲入力をする機能はないため、DatePicker を使って日付の範囲入力をするためのインターフェースを作成するためには、2 つの DatePicker コントロールを並べて表示するなどの方法を用います。

よくある、2 つの DatePicker コントロールを使った日付の範囲入力インターフェース



4.4.2.1. 基本的な使い方

DatePicker コントロールは、日付入力時に表示させるカレンダーの表示をカスタマイズするために Calendar コントロールと同じ以下のプロパティが定義されています。

- DisplayDateStart プロパティ
- DisplayDateEnd プロパティ
- BlackoutDates プロパティ

- SelectedDate プロパティ

DatePicker コントロール固有のプロパティとして、入力された日付の表示方法をカスタマイズするための SelectedDateFormat プロパティがあります。

プロパティ	説明
DatePickerFormat SelectedDateFormat { get; set; }	DatePickerFormat 列挙体の Long と Short のどちらかを設定します。日本語ロケールの場合は Long を指定した場合には yyyy 年 M 月 d 日が表示され、Short を指定した場合には yyyy/MM/dd の書式で表示されます。

SelectedDateFormat を指定した DatePicker コントロールの例を以下に示します。


```
<Label Content="選択日付の書式指定1(Long)" />
<DatePicker
    SelectedDateFormat="Long"
    SelectedDateChanged="DatePicker_SelectedDateChanged"/>

<Label Content="選択日付の書式指定2(Short)" />
<DatePicker
    SelectedDateFormat="Short"
    SelectedDateChanged="DatePicker_SelectedDateChanged"/>
```

SelectedDateChanged イベントは選択した日付が変更したタイミングで実行されるイベントで、以下のようなコードを記載しています。画面に置いてある TextBlock に選択された日付を表示するようなコードを記載しています。

```
private void DatePicker_SelectedDateChanged(object sender, SelectionChangedEventArgs e)
{
    var datePicker = (DatePicker)sender;
    this.textBlockSelectedDate.Text = datePicker.SelectedDate.ToString();
}
```

この XAML を実行して、日付を選択すると以下のような結果になります。

選択日付の書式指定1(Long)	
2013年7月11日	
選択日付の書式指定2(Short)	
2013/07/11	

4.5. メニュー

メニューは、複数のコマンドから、ユーザーに選択をさせるためのユーザーインターフェースを提供します。

4.5.1. ContextMenu コントロール

ContextMenu コントロールは、特定のコントロールに対して固有のメニュー（主に右クリックしたときに表示されるメニュー）を提供するためのコントロールです。ContextMenu コントロールの見た目を以下に示します。



4.5.1.1. ContextMenu の基本的な使い方

ContextMenu コントロールは、コントロールの基本クラスである FrameworkElement クラスで定義されている ContextMenu プロパティに設定して利用します。ContextMenu コントロールの Items プロパティに MenuItem を設定して、メニューの項目を定義します。MenuItem コントロールは、メニューの 1 項目を表すコントロールで、Header プロパティに設定した項目を表示します。上記の使用例のようなメニューを定義する XAML は以下のようになります。

```
<Border ...省略...>
  <Border.ContextMenu>
    <ContextMenu>
      <MenuItem Header="メニュー 1" />
      <MenuItem Header="メニュー 2" />
      <MenuItem Header="メニュー 3" />
      <MenuItem Header="メニュー 4" />
    </ContextMenu>
  </Border.ContextMenu>
</Border>
```

MenuItem コントロールは、Button コントロールと同じように Click イベントがあります。Click イベントにイベントハンドラを設定することで、MenuItem が押されたときの処理を記述できます。

メニュー 1 をクリックしたときに、Hello world というメッセージを表示するには以下のように記述します。

```
<Border ...省略...>
  <Border.ContextMenu>
    <ContextMenu>
      <MenuItem Header="メニュー 1 " Click="MenuItem_Click"/>
      <MenuItem Header="メニュー 2 "/>
      <MenuItem Header="メニュー 3 "/>
      <MenuItem Header="メニュー 4 "/>
    </ContextMenu>
  </Border.ContextMenu>
</Border>
```

コードビハインドには、以下のようなメソッドを定義します。

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello world");
}
```

4.5.1.2. メニューの表示非表示の制御

ContextMenu コントロールは、基本的にマウスの右クリックで表示されますが、IsOpen プロパティを使用することで、プログラムから表示非表示を制御したり、現在表示中かどうか判断することができます。

プロパティ	説明
public bool IsOpen { get; set; }	ContextMenu コントロールが表示されているかどうかを取得または設定します。true のときに表示、false のときは非表示を表します。

4.5.1.3. メニューの階層表示

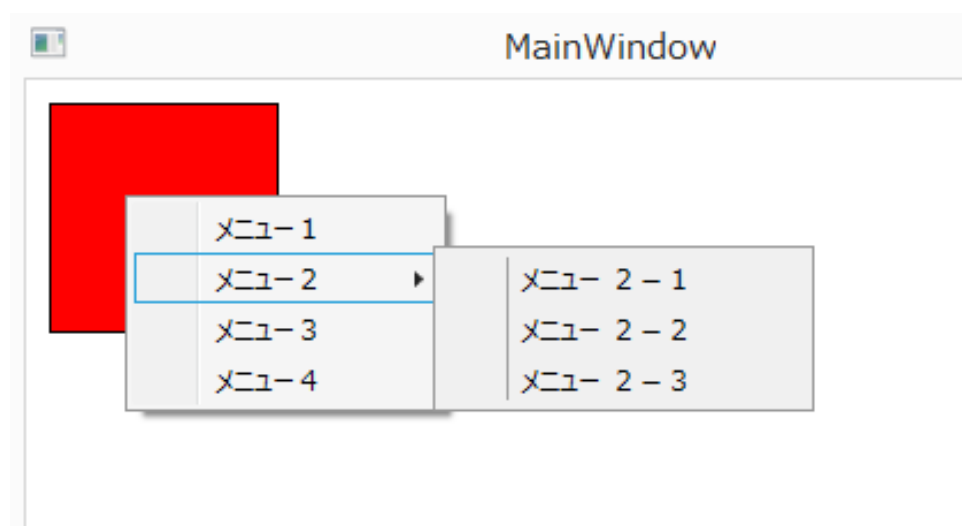
階層構造を持ったメニューを定義するには、以下のように MenuItem を入れ子に定義します。このように定義することで MenuItem の Items プロパティに MenuItem が設定されて、階層構造を持ったメニューを定義できます。

```

<ContextMenu>
  <MenuItem Header="メニュー 1" Click="MenuItem_Click"/>
  <MenuItem Header="メニュー 2">
    <MenuItem Header="メニュー 2 - 1" />
    <MenuItem Header="メニュー 2 - 2" />
    <MenuItem Header="メニュー 2 - 3" />
  </MenuItem>
  <MenuItem Header="メニュー 3"/>
  <MenuItem Header="メニュー 4"/>
</ContextMenu>

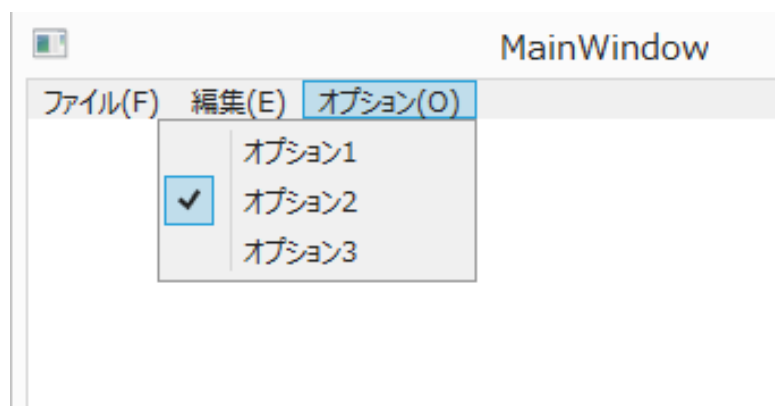
```

実行すると、以下のようにメニュー 2 に子のメニューが出来ます。



4.5.2. Menu コントロール

Menu コントロールは、Window 上部などに表示されるメニュー項目です。一般的にファイル、編集などの項目などがあります。WPF の Menu コントロールを使うと以下のようなメニューを作成することが出来ます。



4.5.2.1. 基本的な使い方

Menu コントロールも通常のコントロールと同様に、画面の好きな位置にレイアウトできます。そのため一般的なメニューの位置である画面上部に表示するためには、Grid などのレイアウトコントロールを使い画面の上部に表示されるように調整する必要があります。

画面の置き場所をレイアウトする以外は、基本的に ContextMenu コントロールと同様の方法で使うことが出来ます。上記の例のような画面を表示するための XAML を以下に示します。

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Menu>
    <MenuItem Header="_F" />
    <MenuItem Header="_E" />
    <MenuItem Header="オプション(_O)">
      <MenuItem IsCheckable="True" Header="オプション1" />
      <MenuItem IsCheckable="True" Header="オプション2" />
      <MenuItem IsCheckable="True" Header="オプション3" />
    </MenuItem>
  </Menu>
</Grid>
```

Grid コントロールを使用して、画面上部に Menu コントロールを配置しています。Menu コントロールではなく MenuItem コントロールの使い方になりますが、Header プロパティに”_F”などのように記述すると、Alt を押した後のキーボードショートカットを指定することが出来ます。

また、オプションメニューで示しているように IsCheckable プロパティを true にすることで、チェック可能なメニュー項目を作成可能です。チェックの有無の確認は IsChecked プロパティで指定可能です。

プロパティ	説明
public bool IsCheckable { get; set; }	MenuItem がチェック可能かどうかを取得または設定します。True の場合に、チェック可能となります。

```
public bool IsChecked { get; set; }
```

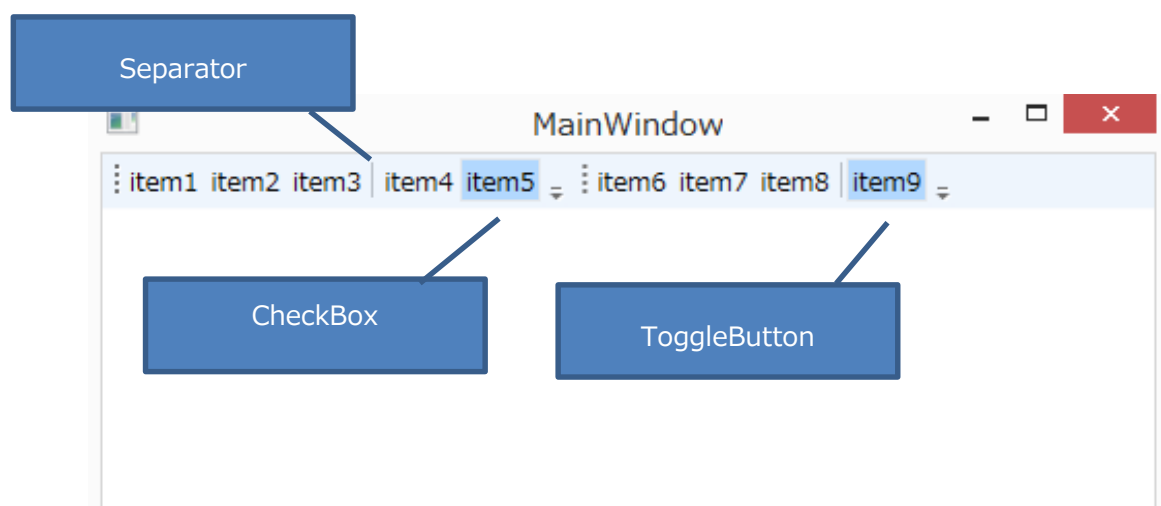
MenuItem がチェックされているかどうかを取得または設定します。チェックされている場合 True になります。

4.5.3. ToolBar コントロール

ToolBar コントロールは、名前の通り、アプリケーションのツールバーを実装するためのコントロールです。

ToolBar コントロールは、ToolBarTray コントロール内に配置する形で実装します。ToolBarTray には複数の ToolBar コントロールを設置でき、ユーザーはマウスを使って ToolBar コントロールの位置を移動させることができます。

以下に ToolBarTray コントロールに ToolBar コントロールを 2 つ置いて、その中に Button や CheckBox や ToggleButton を置いた例を以下に示します。ToolBar コントロール内の縦線は、Separator コントロールを使用しています。



CheckBox コントロールと ToggleButton コントロールは、クリックをしてチェックをつけた状態にしています。画面上で色が変わっていることが確認できます。上記の画面の XAML を以下に示します。

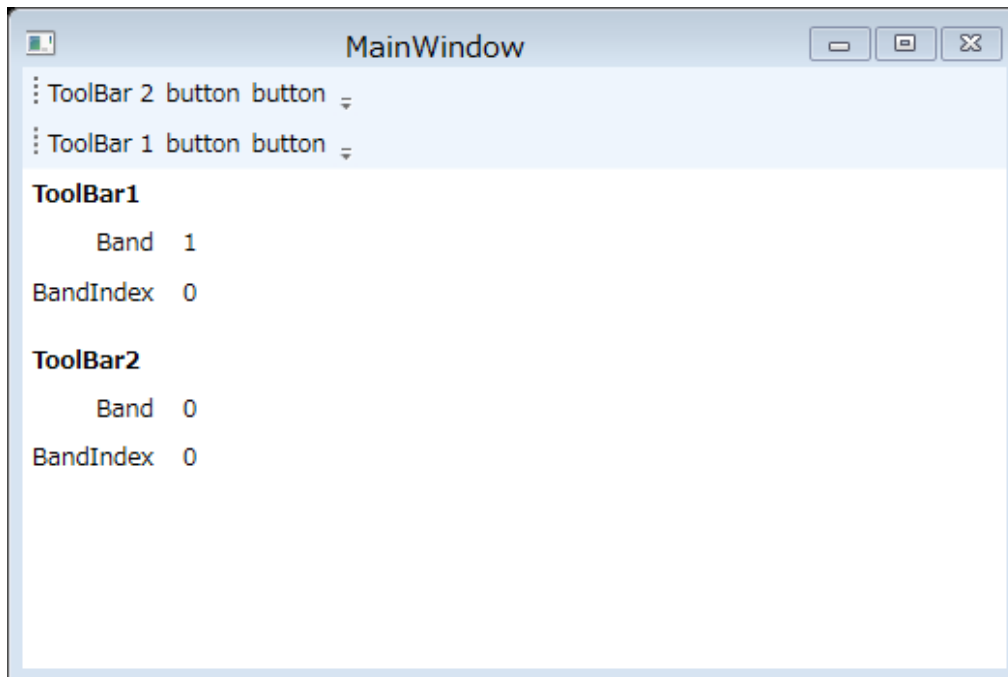
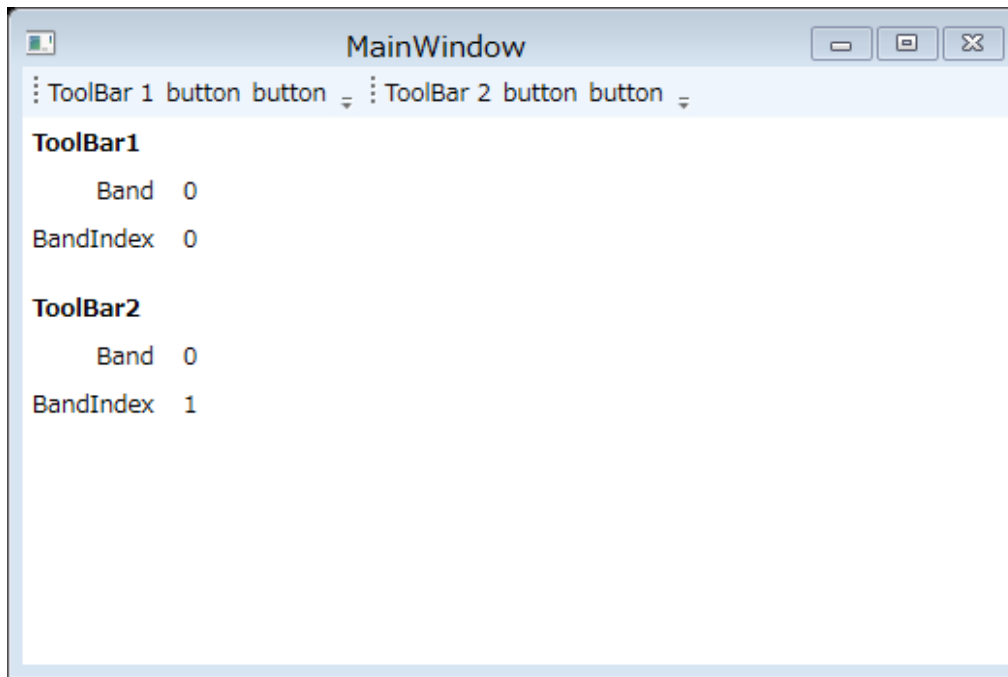
```
<ToolBarTray>
  <ToolBar>
    <Button Content="item1" />
    <Button Content="item2" />
    <Button Content="item3" />
    <Separator />
    <Button Content="item4" />
    <CheckBox Content="item5" />
  </ToolBar>
  <ToolBar>
    <Button Content="item6" />
    <Button Content="item7" />
    <Button Content="item8" />
    <Separator />
    <ToggleButton Content="item9" />
  </ToolBar>
</ToolBarTray>
```

4.5.3.1. ToolBar コントロールの位置の制御

ToolBar コントロールは、ToolBarTray コントロールに複数ある場合ユーザーによって並び替えることができます。これを制御するプロパティが、ToolBar コントロールの Band プロパティと BandIndex プロパティになります。

プロパティ	説明
public int Band { get; set; }	ToolBarTray コントロールの Orientation プロパティが Horizontal の場合に、ToolBar コントロールを何行目に表示するかを表す。 ToolBarTray コントロールの Orientation プロパティが Vertical の場合に ToolBar コントロールの何列目に表示するかを表す。
public int BandIndex { get; set; }	ToolBarTray コントロールの Orientation プロパティが Horizontal の場合に、ToolBar コントロールを何列目に表示するかを表す。 ToolBarTray コントロールの Orientation プロパティが Vertical の場合に ToolBar コントロールの何行目に表示するかを表す。

以下に ToolBar コントロールを 2 つにおいて、それぞれの Band プロパティと BandIndex プロパティを表示するプログラムの実行例を示します。ToolBar コントロールを並び替えると、それに応じて Band プロパティと BandIndex プロパティの値が変わっていることが確認できます。



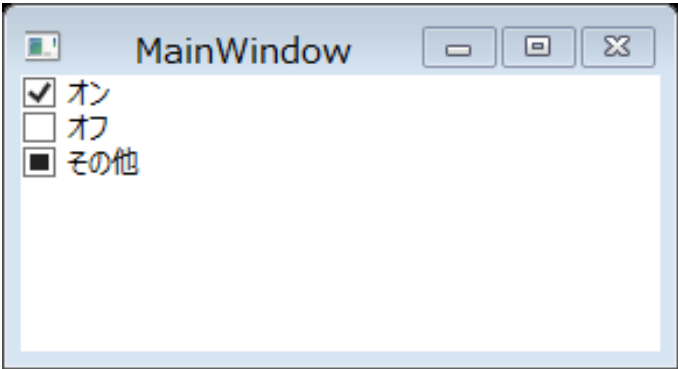
ユーザーが並び替えた ToolBar コントロールの状態を保持するには各 ToolBar コントロールの Band プロパティと BandIndex プロパティを保存・復元すればいいことがわかります。

4.6. 選択系コントロール

ここでは、1 つまたは、複数の項目を選択するために使用できるコントロールについて説明します。

4.6.1. CheckBox コントロール

CheckBox コントロールは、オン・オフ・その他の状態を表すコントロールです。一般的に以下のような見た目をしています。



上記の画面の XAML を以下に示します。

```
<StackPanel>
    <CheckBox Content="オン" IsChecked="True"/>
    <CheckBox Content="オフ" IsChecked="False"/>
    <CheckBox Content="その他" IsChecked="{x:Null}" IsThreeState="True"/>
</StackPanel>
```

CheckBox コントロールの代表的なプロパティを以下に示します。

プロパティ	説明
<code>public Nullable<bool> IsChecked { get; set; }</code>	オンの時に true、オフの時に false、その他の時に null が設定されています。
<code>public bool IsThreeState { get; set; }</code>	CheckBox がオン・オフの 2 つの状態ではなく、オン・オフ・その他の 3 つの状態を持つかどうかを表します。true のときに、その他の状態を持つようになります。デフォルト値は false です。

また、CheckBox コントロールは、選択状態が変わったことを検知するための以下のイベントも備えています。

イベント	説明
Checked	IsChecked プロパティが true になったときに呼び出されます。
Unchecked	IsChecked プロパティが false になったときに呼び出されます。
Indeterminate	IsChecked プロパティが null になったときに呼び出されます。

これらのイベントを使うことで、選択状態に応じた処理を行うことができます。

CheckBox のチェック状態に応じて TextBlock の表示を切り替えるプログラムは以下のようになります。

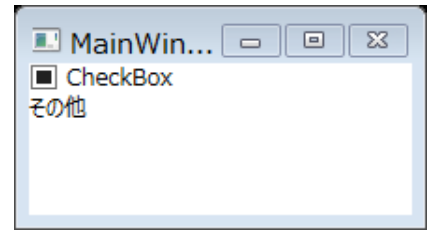
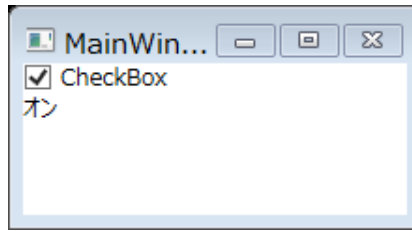
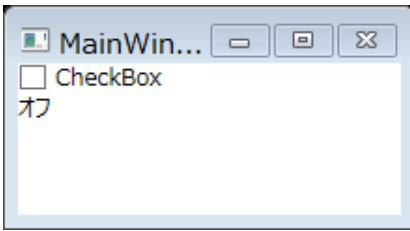
```
<StackPanel>
  <CheckBox
    IsThreeState="True"
    Content="CheckBox"
    Checked="CheckBox_Checked"
    Unchecked="CheckBox_Unchecked"
    Indeterminate="CheckBox_Indeterminate"/>
  <TextBlock x:Name="textBlock" Text="オフ"/>
</StackPanel>
```

```
private void CheckBox_Checked(object sender, RoutedEventArgs e)
{
    this.textBlock.Text = "オン";
}

private void CheckBox_Unchecked(object sender, RoutedEventArgs e)
{
    this.textBlock.Text = "オフ";
}

private void CheckBox_Indeterminate(object sender, RoutedEventArgs e)
{
    this.textBlock.Text = "その他";
}
```

実行すると以下のような結果になります。



4.6.2. ComboBox コントロール

ComboBox コントロールは、複数の選択肢の中から 1 つをユーザーに選択してもらうためのユーザーインターフェースを提供するコントロールです。オプションとして、複数の選択肢の中から 1 つを選ぶか、自由にテキストを入力する方法も提供することができます。

ComboBox の基本的な使い方は、ItemsSource プロパティにオブジェクトのコレクションを設定して、ItemTemplate で見た目を定義する方法になります。例として、以下のような Name と Age プロパティを持つ Person クラスを表示する場合のコードを示します。

```
// ComboBoxに表示するオブジェクト
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

ComboBox の XAML での定義は以下のようになります。ItemTemplate に、Person クラスの表示を定義するテンプレートを設定しています。

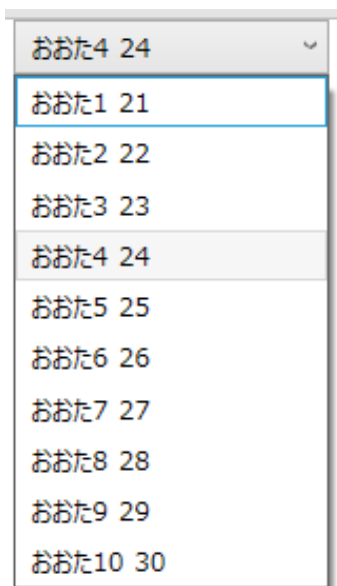
```
<ComboBox x:Name="comboBox">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Name}" Margin="2.5"/>
                <TextBlock Text="{Binding Age}" Margin="2.5"/>
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

コードビハインドのコンストラクタで、この ComboBox コントロールの ItemsSource プロパティにデータを設定します。

```
var items = Enumerable.Range(1, 10)
    .Select(i => new Person { Name = "おおた" + i, Age = 20 + i })
    .ToList();

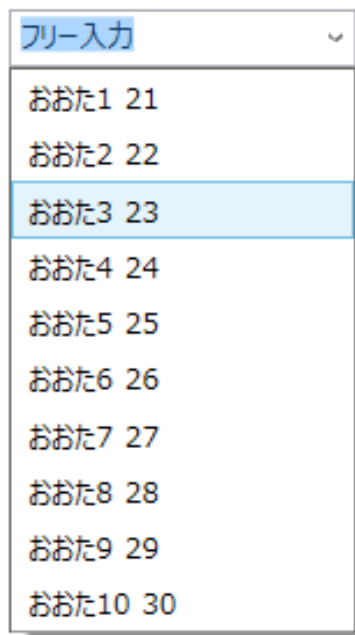
this.comboBox.ItemsSource = items;
```

この状態で実行すると、以下のように ItemTemplate に設定した見た目の通り「名前 年齢」の順番でデータが並んだ状態で表示されます。



また、IsEditable プロパティを true に設定することで、選択肢にない項目を TextBox と同じ要領で入力できるようにもなります。選択した項目の表示場所が TextBox になることで、ItemTemplate の表示ができなくなるため、TextSearch.TextPath プロパティで選択した項目の、どのプロパティの値を表示するのか指定する必要があります。（省略した場合は、ToString の結果がそのまま表示されます）

```
<ComboBox x:Name="comboBoxEditable" Grid.Column="2" Grid.Row="2" MinWidth="150"
    IsEditable="True" TextSearch.TextPath="Name">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Name}" Margin="2.5"/>
                <TextBlock Text="{Binding Age}" Margin="2.5"/>
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```



4.6.2.1. 選択項目の操作

ComboBox コントロールで選択された要素を取得するには、インデックスを取得する方法と、選択された値そのものを取得するプロパティがあります。また、Text プロパティを使うことで現在選択中の項目のテキストも取得できます。

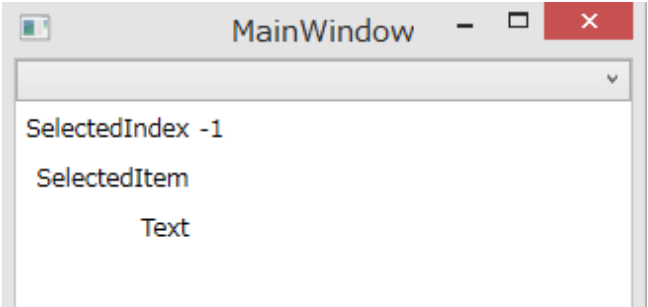
プロパティ	説明
public int SelectedIndex { get; set; }	ComboBox コントロールで現在選択中の項目のインデックスを取得または設定します。未選択時は-1 です。

public object SelectedItem { get; set; }	ComboBox コントロールで現在選択中の値を取得または設定します。
public string Text {get; set; }	ComboBox コントロールで現在のテキストを取得または設定します。

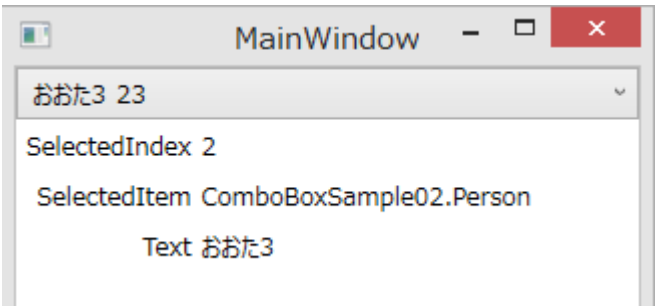
以下のように設定した、ComboBox コントロールの SelectedIndex プロパティ、SelectedItem プロパティ、Text プロパティを表示するプログラムの実行結果を示します。

```
<ComboBox x:Name="comboBox" Grid.ColumnSpan="2" TextSearch.TextPath="Name">
  <ComboBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Name}" Margin="2.5"/>
        <TextBlock Text="{Binding Age}" Margin="2.5"/>
      </StackPanel>
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>
```

実行直後は、SelectedIndex プロパティが-1 になっていることが確認できます。また、SelectedItem プロパティは null で Text プロパティは空文字になっています。



3 番目の項目を選択すると以下のような表示になります。



SelectedIndex プロパティが 3 番目を表す 2 になっている点と、SelectedItem プロパティが、選択項目の Person オブジェクトになっていることが確認できます。Text プロパティは TextSearch.TextPath 添付プロパティで Name プロパティを表示するようにしているため、Name プロパティの値が表示されていることがわかります。

4.6.3. ListBox コントロール

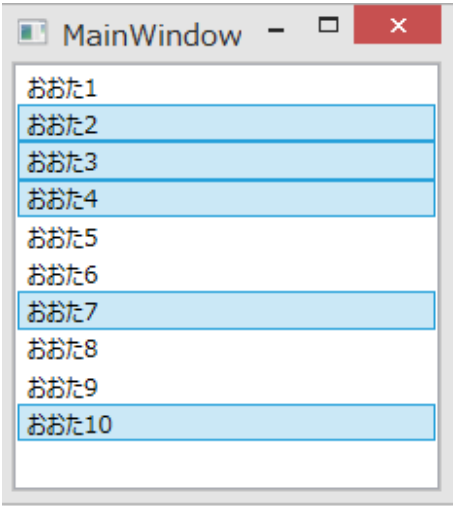
ListBox コントロールは、1 つまたは複数の項目をユーザーに選択させることが出来るコントロールです。基本的な使用法は、ComboBox コントロールと同様になります。ItemTemplate による見た目の設定、ItemsSource プロパティによる、選択項目の設定、SelectedItem プロパティや SelectedIndex プロパティによる選択項目の管理ができます。

これは、ListBox コントロールと ComboBox コントロールが、同じ Selector コントロールを継承しているためです。ここでは、ListBox コントロール固有の SelectionMode プロパティの説明のみを行います。

プロパティ	説明
public SelectionMode SelectionMode { get; set; }	<p>ユーザーがどのように ListBox コントロールの項目を選択するか取得または設定します。SelectionMode 列挙型は以下の値があります。</p> <ul style="list-style-type: none"> Extended : Shift キーを押しながら連続した複数項目を選択できる。 Multiple : Shift キーを押さなくても複数項目を選択できる。 Single : 単一項目を選択できる。

以下のように SelectionMode プロパティに、Multiple を設定したときの動作例を以下に示します。

```
<ListBox x:Name="listBox" SelectionMode="Multiple">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <!-- Name プロパティを持つPersonクラス用のテンプレート -->
      <TextBlock Text="{Binding Name}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

このように複数項目が、選択されている場合 SelectedItems プロパティで取得します。

プロパティ	説明
public IList SelectedItems { get; }	複数選択されている項目のリストを取得します。

4.6.4. RadioButton コントロール

RadioButton コントロールは、複数の選択肢の中から 1 つをユーザーに選択してもらうときに使うコントロールです。RadioButton コントロールは、デフォルトでは同じパネル（StackPanel や Grid など）にある RadioButton コントロールから、1 つだけチェックをつけることが出来ます。また、一度つけてしまったチェックは、ユーザーからは確認できないという特徴があります。チェック状態は CheckBox コントロールと同様に IsChecked プロパティで確認でき、Checked イベントでチェック状態に変更があったことをハンドリングできます。

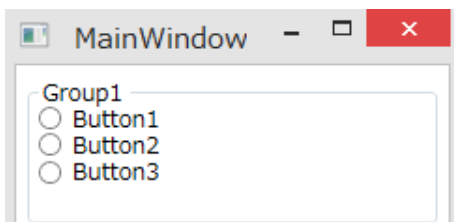
以下に RadioButton コントロールの基本的な使い方を示します。1 つの StackPanel 上に置いた RadioButton コントロールに Checked イベントのハンドラを設定しています。

```
<GroupBox Header="Group1">
  <StackPanel>
    <RadioButton Content="Button1" Checked="RadioButton_Checked"/>
    <RadioButton Content="Button2" Checked="RadioButton_Checked" />
    <RadioButton Content="Button3" Checked="RadioButton_Checked"/>
    <TextBlock x:Name="textBlockSelected" />
  </StackPanel>
</GroupBox>
```

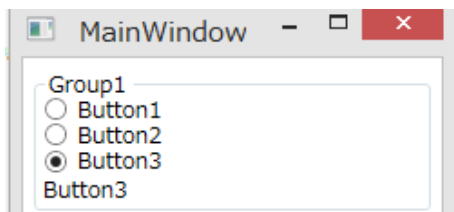
RadioButton_Checked イベントハンドラでは、sender を使って選択された RadioButton コントロールを取得して textBlockSelected に Content の内容を表示しています。

```
private void RadioButton_Checked(object sender, RoutedEventArgs e)
{
    var radioButton = (RadioButton)sender;
    this.textBlockSelected.Text = radioButton.Content.ToString();
}
```

実行すると初期状態では、どの RadioButton コントロールにもチェックは入っていません。



マウスで選択すると、選択した項目の Content が、下部に表示されます。



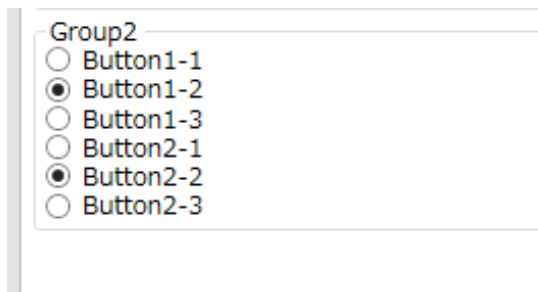
4.6.4.1. 同じパネル上で複数のグループを作成する方法

デフォルトの挙動では、同じパネルに置いた RadioButton コントロールは、1 つしか選択できません。この制限を回避するために、GroupName というプロパティがあります。GroupName に文字列を指定すると、同じ GroupName の RadioButton コントロールの中から 1 つ選択できるという挙動になります。

以下に、1 つの StackPanel 上に 2 つの異なる GroupName を指定した XAML を示します。

```
<GroupBox Header="Group2">
  <StackPanel>
    <RadioButton GroupName="group1" Content="Button1-1" />
    <RadioButton GroupName="group1" Content="Button1-2" />
    <RadioButton GroupName="group1" Content="Button1-3" />
    <RadioButton GroupName="group2" Content="Button2-1" />
    <RadioButton GroupName="group2" Content="Button2-2" />
    <RadioButton GroupName="group2" Content="Button2-3" />
  </StackPanel>
</GroupBox>
```

実行して、group1 と group2 の RadioButton コントロールをそれぞれ選択した結果を以下に示します。



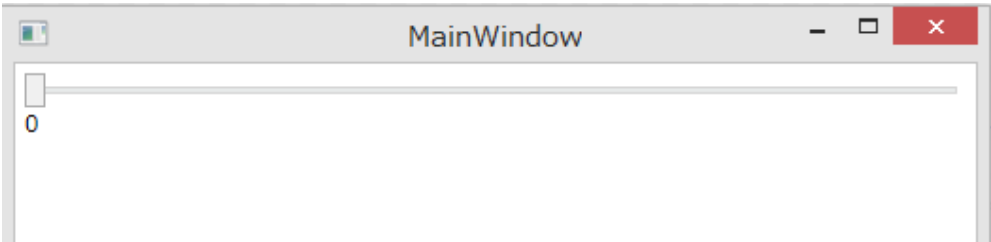
同一パネル上でも、複数の RadioButton コントロールのグループが出来ていることが確認できます。

4.6.5. Slider コントロール

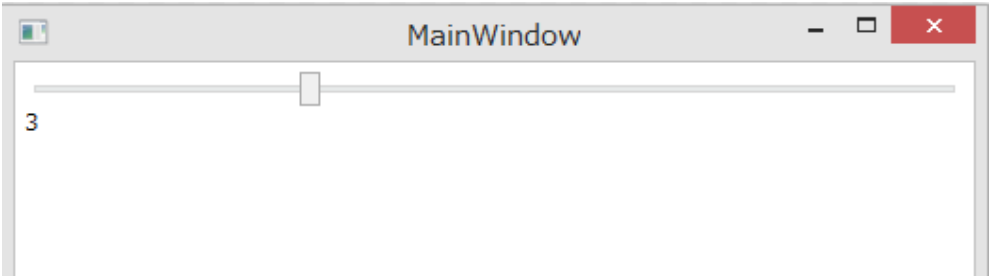
Slider コントロールは、任意の範囲の数値から 1 つをユーザーに選択させるコントロールです。デフォルトでは 0～10 の間の実数を選択する動作をします。以下のように StackPanel 上に置いただけの Slider コントロールの動作を以下に示します。

```
<StackPanel Margin="5">
  <Slider x:Name="slider" />
  <TextBlock Text="{Binding Value, ElementName=slider}" />
</StackPanel>
```

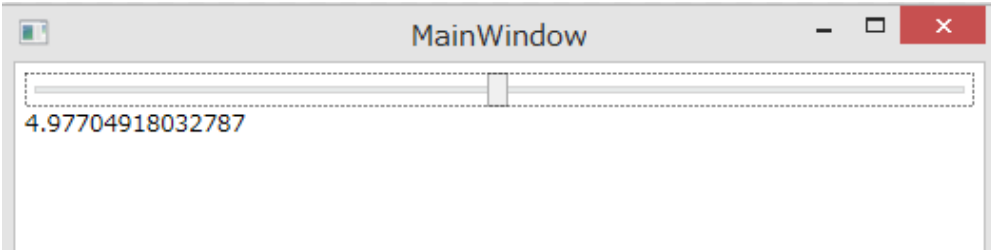
起動直後は、以下のように 0 をさしています。



スライダーコントロールの余白の線をクリックすると 1 ずつ値が増えていきます。以下の図は、3 回クリックしたときの様子です。値が 3 になっていることが確認できます。



つまみを移動することで、範囲内の任意の値を選択することもできます。



Slider コントロールの値を取得するには Value プロパティを使用します。

プロパティ	説明
<code>public double Value { get; set; }</code>	Slider コントロールの現在の値を取得または設定します。

4.6.5.1. 選択範囲のカスタマイズ

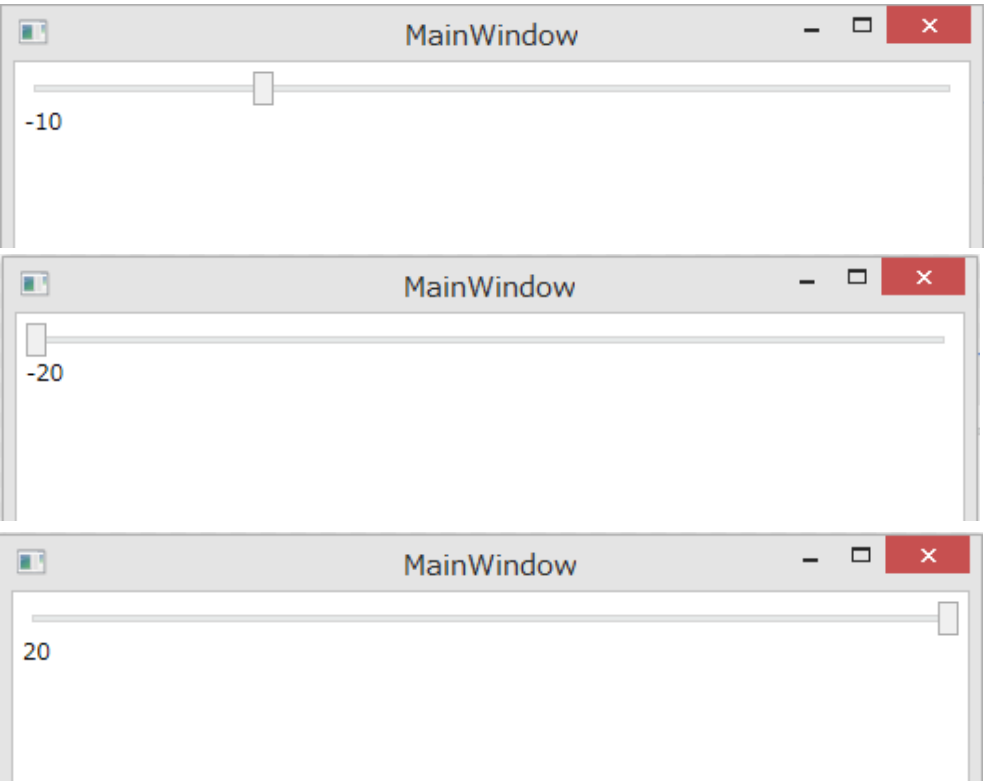
Slider コントロールは、Minimum プロパティと Maximum プロパティを使って最大値と最小値を変更することが出来ます。

プロパティ	説明
-------	----

<code>public double Minimum { get; set; }</code>	Slider コントロールの最小値を取得または設定します。
<code>public double Maximum { get; set; }</code>	Slider コントロールの最大値を取得または設定します。

以下のように XAML で Minimum プロパティ、Maximum プロパティ、Value プロパティを指定した実行結果を示します。

```
<StackPanel Margin="5">
    <Slider x:Name="slider" Minimum="-20" Maximum="20" Value="-10" />
    <TextBlock Text="{Binding Value, ElementName=slider}" />
</StackPanel>
```



初期状態で-10、最小値が-20、最大値が 20 になっていることが確認できます。

4.6.5.2. 値の増減幅の設定

Slider コントロールは、余白をクリックしたときの値の移動量（デフォルトは 1）と、矢印キーを押したときの値の移動量(デフォルトは 0.1)を設定できます。それぞれ、LargeChange プロパティと、SmallChange プロパティで指定します。

プロパティ	説明
public double LargeChange { get; set; }	マウスでつまみの横の余白の線をクリックしたときの値の変化量を取得または設定します。
public double SmallChange { get; set; }	矢印キーを押したときの値の変化量を取得または設定します。

4.6.5.3. 縦方向のスライダーと目盛り

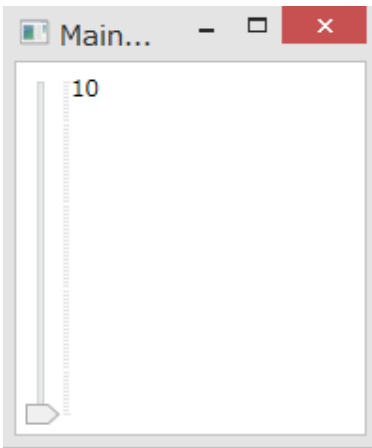
Slider コントロールは、横方向だけではなく縦方向にしたり、Slider コントロールに目盛りをつけることが出来ます。

プロパティ	説明
public Orientation Orientation {get; set; }	スライダーが横方向なのか縦方向なのかを取得または設定します。デフォルト値は Horizontal（水平）です。Vertical を設定することで縦方向にできます。
public TickPlacement TickPlacement { get; set; }	Slider コントロールの目盛りの状態を取得または設定します。 <ul style="list-style-type: none"> ● None : 目盛りなし。（デフォルト値） ● Both : 横方向のときは上下に、縦方向のときは左右に目盛りがつきます。 ● BottomRight : 横方向のときは下に、縦方向のときは右に目盛りがつきます。 ● TopLeft : 横方向のときは上に、縦方向のときは左に目盛りがつきます。

縦方向にして、目盛りをつける場合の XAML を以下に示します。

```
<StackPanel Margin="5" Orientation="Horizontal">
    <Slider x:Name="slider"
        Minimum="10"
        Maximum="100"
        SmallChange="1"
        LargeChange="10"
        TickPlacement="BottomRight"
        Orientation="Vertical"/>
    <TextBlock Text="{Binding Value, ElementName=slider}" />
</StackPanel>
```

実行すると以下のように縦方向で目盛り付きの Slider コントロールが表示されます。



4.7. ナビゲーションコントロール

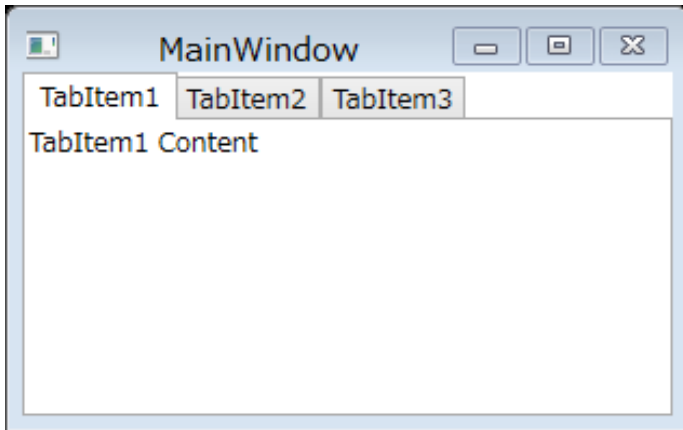
XBAP や Frame を使ったページナビゲーションは、使用頻度が低いため、ここでの説明は省略します。詳細は以下の MSDN のページを参照してください。

ナビゲーションの概要

[http://msdn.microsoft.com/ja-jp/library/ms750478\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/ms750478(v=vs.110).aspx)

4.7.1. TabControl

TabControl は、以下のようなタブで切り替えて複数のコンテンツを表示する UI を提供するためのコントロールです。



TabControl は、Items プロパティに TabItem コントロールを指定してタブを作成します。TabItem コントロールは、Header プロパティにタブのヘッダーに表示するコンテンツを設定して、Content プロパティにタブの中に表示するコンテンツを設定します。上記の画面の XAML を以下に示します。

```
<TabControl>
  <TabItem Header="TabItem1">
    <TextBlock Text="TabItem1 Content" />
  </TabItem>
  <TabItem Header="TabItem2">
    <TextBlock Text="TabItem2 Content" />
  </TabItem>
  <TabItem Header="TabItem3">
    <TextBlock Text="TabItem3 Content" />
  </TabItem>
</TabControl>
```

4.7.1.1. TabControl でコレクションを表示する

TabControl は、ComboBox コントロールや ListBox コントロールと同じ Selector コントロールを継承しています。そのため、ItemsSource プロパティにコレクションを設定すると、ItemTemplate プロパティに従って Tab のヘッダーを表示することが出来ます。Tab のコンテンツ部は、ContentTemplate プロパティに DataTemplate を指定して表示方法を定義します。

以下のような Person クラスを表示する TabControl を説明します。


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

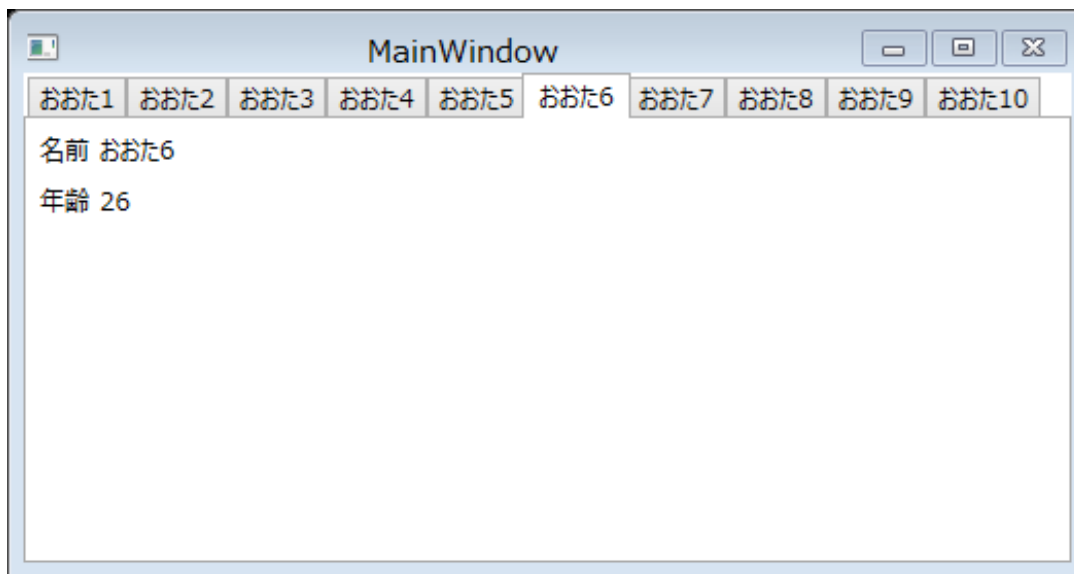
TabControl の ItemTemplate で Name を表示して、ContentTemplate で Name と Age を表示する XAML を以下に示します。

```
<TabControl x:Name="tabControl">
    <TabControl.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Name}" />
        </DataTemplate>
    </TabControl.ItemTemplate>
    <TabControl.ContentTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>
                <Label Grid.Row="0" Grid.Column="0" Content="名前" />
                <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Name}"
                    VerticalAlignment="Center" />
                <Label Grid.Row="1" Grid.Column="0" Content="年齢" />
                <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Age}"
                    VerticalAlignment="Center" />
            </Grid>
        </DataTemplate>
    </TabControl.ContentTemplate>
</TabControl>
```

コードビハインドのコンストラクタで、以下のように TabControl の ItemsSource プロパティに値を設定します。

```
public MainWindow()  
{  
    InitializeComponent();  
  
    var source = Enumerable.Range(1, 10)  
        .Select(i => new Person { Name = "おおた" + i, Age = 20 + i });  
    this.tabControl.ItemsSource = source;  
}
```

実行すると、以下のようにコレクションが Tab で表示されます。



4.8. ファイルダイアログ

WPF は、ファイルを開いたり保存するためのダイアログを提供しています。以下の 2 つのダイアログがあります。

1. SaveFileDialog
ファイルを保存するときに使用するダイアログです。
2. OpenFileDialog
ファイルを開くときに使用するダイアログです。

これらのダイアログは、主に以下のような流れで使います。

- インスタンスを生成する。

- Title プロパティと、Filter プロパティを設定する。
- ShowDialog を呼び出してダイアログを表示する。
- ダイアログの戻り値を確認して OK が押されている場合は、FileName プロパティか FileNames プロパティを使用して選択されたファイルを取得する。

プロパティの説明を以下に示します。

プロパティ	説明
public string Title { get; set; }	ファイルダイアログのタイトルに表示されるテキストを取得または設定します。
public string Filter { get; set; }	ファイルダイアログで選択できるファイルの種類を示すフィルターを取得または設定します。
public string FileName { get; set; }	ファイルダイアログで選択されたファイルのフルパスを取得または設定します。
public string[] FileNames { get; }	ファイルダイアログで選択されたすべてのファイルのフルパスを取得します。

Filter プロパティの書式について詳しく説明します。Filter プロパティは以下のような書式で指定します。

全てのファイル|*..*|ワードファイル|*.doc;*.docx

表示用のラベルと対象の拡張子を|で区切って記述します。対象の拡張子が複数ある場合は;で区切って指定します。上記の記述では、全てのの拡張子を対象とした全てのファイルと、doc と docx という拡張子を対象としたワードファイルという選択肢がファイルダイアログに表示されます。

次に、ShowDialog メソッドについて説明します。

メソッド	説明
public Nullable<bool> ShowDialog()	ファイルダイアログを表示します。OK が選択された場合は、true を返して、それ以外の場合は false を返します。

以下に、OpenFileDialog と SaveFileDialog の使用例を示します。

画面は、Button コントロールと TextBlock コントロールを並べたシンプルなものです。

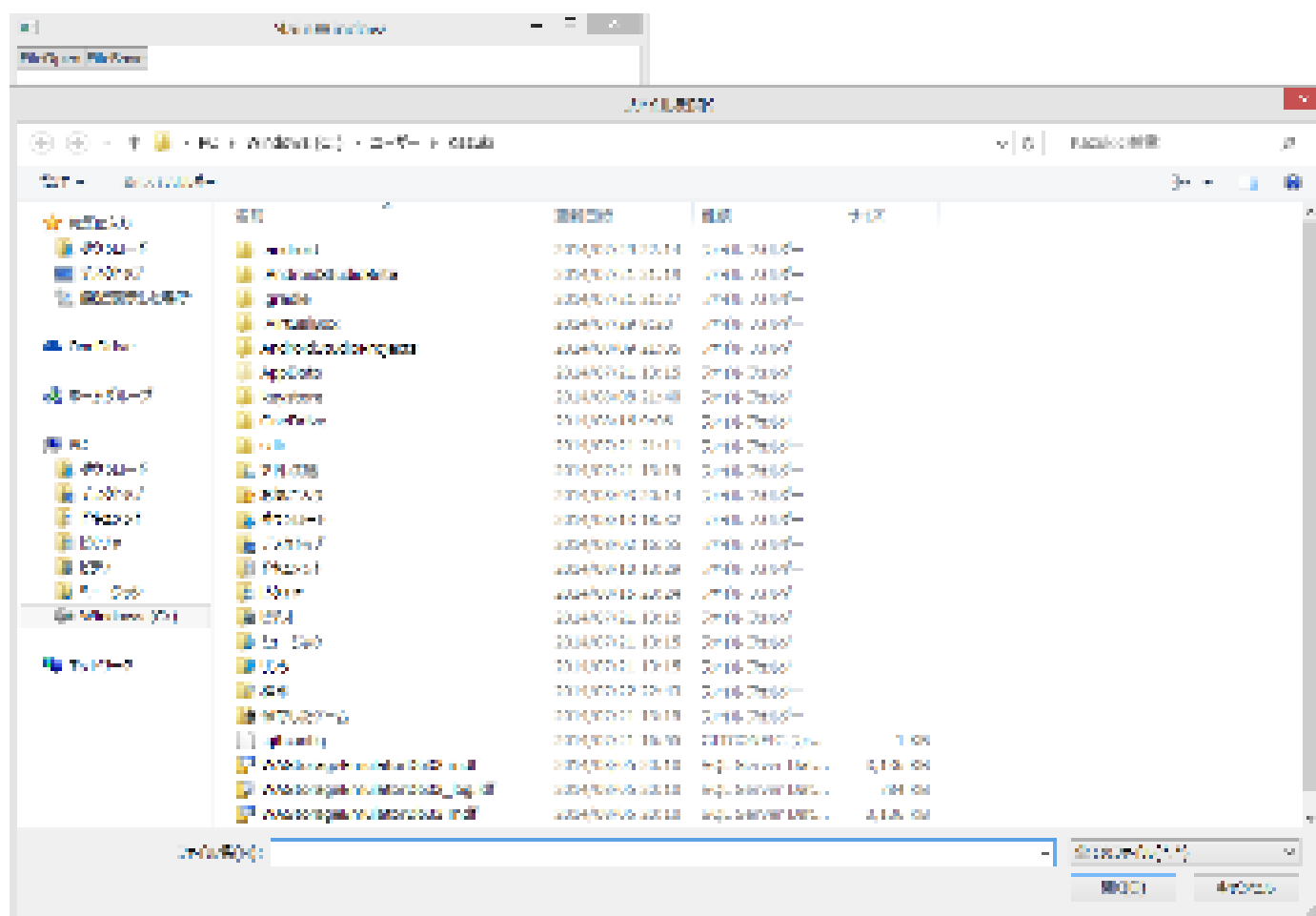
```
<StackPanel>
  <StackPanel Orientation="Horizontal">
    <Button Content="FileOpen" Click="FileOpenButton_Click" />
    <Button Content="FileSave" Click="FileSaveButton_Click" />
  </StackPanel>
  <TextBlock x:Name="textBlockFileName" />
</StackPanel>
```

コードビハインドの、Button のクリックイベントでダイアログを表示してファイル名を画面に表示しています。

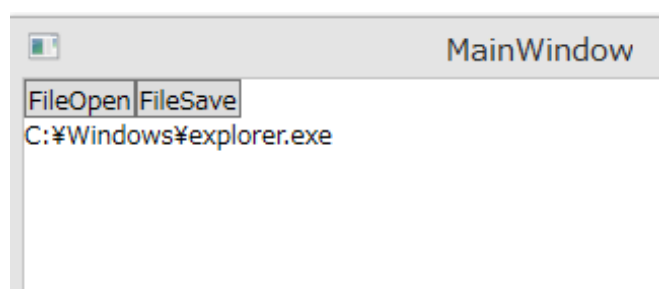
```
private void FileOpenButton_Click(object sender, RoutedEventArgs e)
{
    var dialog = new OpenFileDialog();
    dialog.Title = "ファイルを開く";
    dialog.Filter = "全てのファイル(*.*)|*.*";
    if (dialog.ShowDialog() == true)
    {
        this.textBlockFileName.Text = dialog.FileName;
    }
    else
    {
        this.textBlockFileName.Text = "キャンセルされました";
    }
}

private void FileSaveButton_Click(object sender, RoutedEventArgs e)
{
    var dialog = new SaveFileDialog();
    dialog.Title = "ファイルを保存";
    dialog.Filter = "テキストファイル|*.txt";
    if (dialog.ShowDialog() == true)
    {
        this.textBlockFileName.Text = dialog.FileName;
    }
    else
    {
        this.textBlockFileName.Text = "キャンセルされました";
    }
}
```

実行してボタンを押すと、以下のようにダイアログが表示されます。



ファイルを選択すると TextBlock に選択したファイルのフルパスが表示されます。



4.9. 情報を表示するコントロール

4.9.1. Label コントロール

Label コントロールは、コントロールに対するラベルを表示するコントロールです。Label コントロールは Button コントロールなどと同じ ContentControl を継承しているため、Content プロパティに任意の値を設定して ContentTemplate プロパティを使って、表示をカスタマイズできますが、文字列を指定するのが一般的です。

Label コントロールは、「_アルファベット」でアクセスキーを提供して、Target プロパティに設定したコントロールにフォーカスをうつす機能があります。

```
<StackPanel Margin="5">
  <Label Content="ファイル(_F)" Target="{Binding ElementName=textBox1}" />
  <TextBox x:Name="textBox1" />
  <Label Content="編集(_E)" Target="{Binding ElementName=textBox2}" />
  <TextBox x:Name="textBox2" />
</StackPanel>
```

上記の例では Alt+F で textBox1 へフォーカスが移動し、Alt+E で textBox2 へフォーカスが移動します。

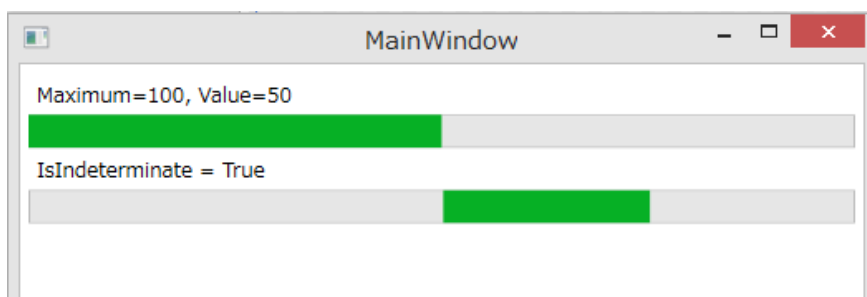
4.9.2. ProgressBar コントロール

ProgressBar コントロールは、ユーザーに処理の進捗状況を表示するためのコントロールです。ProgressBar コントロールは、Slider コントロールと同じ親クラスを持っていて、Maximum プロパティ、Value プロパティなどを使って値の範囲を指定することが出来ます。また、IsIndeterminate プロパティを true に設定することで、固定の値ではなく、何かの作業をしていることを示すアニメーションを表示することが出来ます。

以下に使用例を示します。

```
<StackPanel Margin="5">
  <Label Content="Maximum=100, Value=50" />
  <ProgressBar Maximum="100" Value="50" Height="20" />
  <Label Content="IsIndeterminate = True" />
  <ProgressBar IsIndeterminate="True" Height="20" />
</StackPanel>
```

実行すると、以下のようになります。



4.9.3. StatusBar コントロール

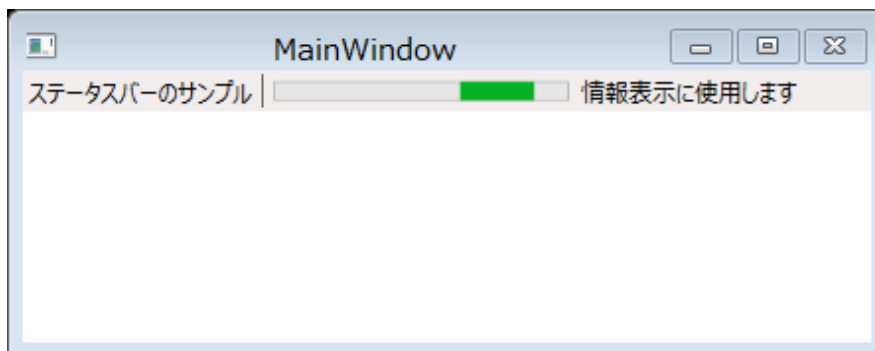
StatusBar コントロールは、水平方向に項目を並べてユーザーに情報を表示するためのコントロールです。

StatusBar コントロールは、StatusBarItem コントロールを内部に複数持ち、Separator コントロールで StatusBarItem コントロールの間を区切って表示することもできます。主に画面の下部に表示することが多いコントロールです。

以下に StatusBar コントロールの使用例を示します。

```
<StatusBar>
  <StatusBarItem>
    <TextBlock Text="ステータスバーのサンプル" />
  </StatusBarItem>
  <Separator />
  <StatusBarItem>
    <ProgressBar IsIndeterminate="True" Width="150" Height="10"/>
  </StatusBarItem>
  <StatusBarItem>
    <TextBlock Text="情報表示に使用します" />
  </StatusBarItem>
</StatusBar>
```

実行すると以下ようになります。



4.9.4. TextBlock コントロール

TextBlock コントロールは、テキストを画面に表示するためのコントロールです。基本的な使い方は Text プロパティに文字列を設定して使います。

```
<TextBlock Text="文字列を指定します" />
```


TextBlock コントロールは、Text プロパティ以外に Inlines というプロパティもあり、ここに Run や Hyperlink をつけて書式付のテキストやハイパーリンクを挿入することもできます。

```
<TextBlock>
  <Run Text="いろいろ指定できる" />
  <Hyperlink NavigateUri="http://www.bing.com">リンクできる</Hyperlink>
  <Run Foreground="Red" Text="色も付けれる" />
  <Run FontFamily="メイリオ" Text="フォントも変えれます" />
  <LineBreak />
  <Run Text="改行も入れることができます" />
</TextBlock>
```

この XAML は、以下のように表示されます。

いろいろ指定できる [リンクできる](http://www.bing.com) 色も付けれる フォントも変えれます
改行も入れることができます

4.9.5. Popup コントロール

Popup コントロールは、画面上に別ウィンドウとして項目を表示するためのコントロールです。Popup コントロールは、IsOpen プロパティを持っていて、このプロパティの値が true になったときに表示されます。Popup コントロールは、デフォルトでは、親要素の下に表示されます。Placement プロパティに Top、Bottom、Right、Left などの値を設定することで表示位置を下以外にすることが出来ます。

Button コントロールの周りに Popup コントロールが表示されるプログラムの例を以下に示します。

```

<Grid>
  <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
    <Button Content="popup button" Click="Button_Click"/>
    <Popup x:Name="popup1">
      <TextBlock Background="LightGray" Text="Bottom(Default)" />
    </Popup>
    <Popup x:Name="popup2" Placement="Left">
      <TextBlock Background="LightGray" Text="Left" />
    </Popup>
    <Popup x:Name="popup3" Placement="Top">
      <TextBlock Background="LightGray" Text="Top" />
    </Popup>
    <Popup x:Name="popup4" Placement="Right">
      <TextBlock Background="LightGray" Text="Right" />
    </Popup>
  </StackPanel>
</Grid>

```

Button を押すたびに表示、非表示を切り替えています。

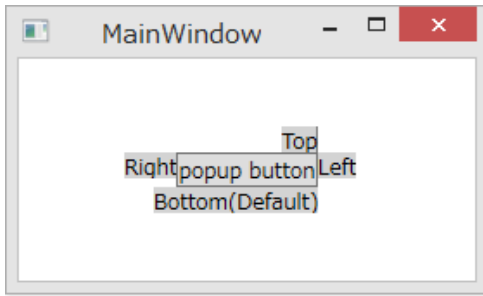
```

private void Button_Click(object sender, RoutedEventArgs e)
{
    var popups = new[]
    {
        this.popup1,
        this.popup2,
        this.popup3,
        this.popup4
    };

    foreach (var popup in popups)
    {
        popup.IsOpen = !popup.IsOpen;
    }
}

```

実行して Button をクリックすると以下のように上下左右に Popup コントロールが表示されます。



Popup コントロールは非常に低レベルなレイヤのコントロールで、柔軟に細かく設定ができる反面制御がとても難しいコントロールになります。普通は、ComboBox コントロールや ContextMenu コントロールなどで内部的に Popup コントロールが使われているので、そちらを使いますが、どうしても Popup させる Window がほしい場合は、このコントロールの利用を検討してください。（例としてはインテリセンスの自前実装など）

Popup コントロールの細かい制御方法については MSDN の以下のページを参照してください。

- ポップアップの概要
[http://msdn.microsoft.com/ja-jp/library/ms749018\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/ms749018(v=vs.110).aspx)
- ポップアップの配置動作
[http://msdn.microsoft.com/ja-jp/library/bb613596\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/bb613596(v=vs.110).aspx)

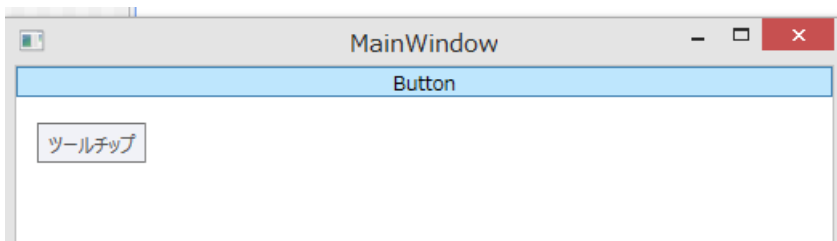
4.9.6. ToolTip コントロール

ToolTip コントロールは、マウスをコントロール上に置いたときに補助的な情報を表示するためのコントロールです。ToolTip をコントロールに表示するには、ほぼ全てのコントロールの基底クラスである FrameworkElement クラスと FrameworkContentElement クラスで定義されている ToolTip プロパティを使用します。

一番単純な例は、ToolTip プロパティに文字列を指定する方法です。

```
<Button Content="Button" ToolTip="ツールチップ" />
```

このボタンの上にマウスカーソルを持っていくと、以下のようにツールチップが表示されます。

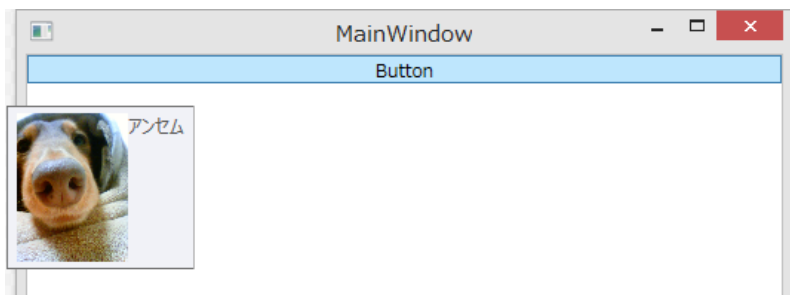


ToolTip コントロールは、Button コントロールなどと同じ ContentControl を親に持ちます。そのため、ToolTip コントロールの Content プロパティにコントロールを直接入れて表示したり、ContentTemplate プロパティを使ってデータを任意の形で表示することが出来ます。

以下は、ToolTip 内に画像を表示している例です。

```
<Button Content="Button">
  <Button.ToolTip>
    <ToolTip>
      <StackPanel Orientation="Horizontal">
        <Image Source="anthem.jpg" Height="100" />
        <TextBlock Text="アンセム" />
      </StackPanel>
    </ToolTip>
  </Button.ToolTip>
</Button>
```

実行して、ボタンの上にマウスカーソルを持っていくと以下のように表示されます。



4.10. 入力

4.10.1. TextBox コントロール

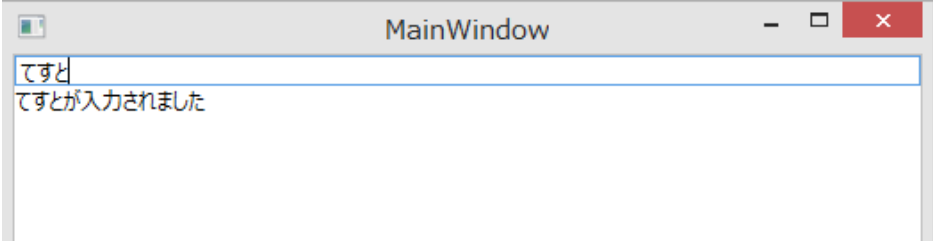
TextBox コントロールは文字列を入力するインターフェースを提供するコントロールです。入力された文字列は、Text プロパティで取得できます。Text が変更されたタイミングは TextChanged イベントを購読することで判別できます。

基本的な TextBox の使用方法を以下に示します。

```
<StackPanel>
    <TextBox_TextChanged="TextBox_TextChanged" />
    <TextBlock x:Name="textBlock" TextWrapping="Wrap"/>
</StackPanel>
```

```
private void TextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    var textBox = (TextBox)sender;
    this.textBlock.Text = textBox.Text + "が入力されました";
}
```

TextBox コントロールの入力内容を TextBlock コントロールへ設定しています。実行すると以下のようになります。



4.10.1.1. 改行やタブを受け入れる TextBox コントロール

TextBox コントロールは、デフォルトでは、改行やタブを受け付けません。メモ帳のような動作をさせるためには、以下の 4 つのプロパティを使用します。

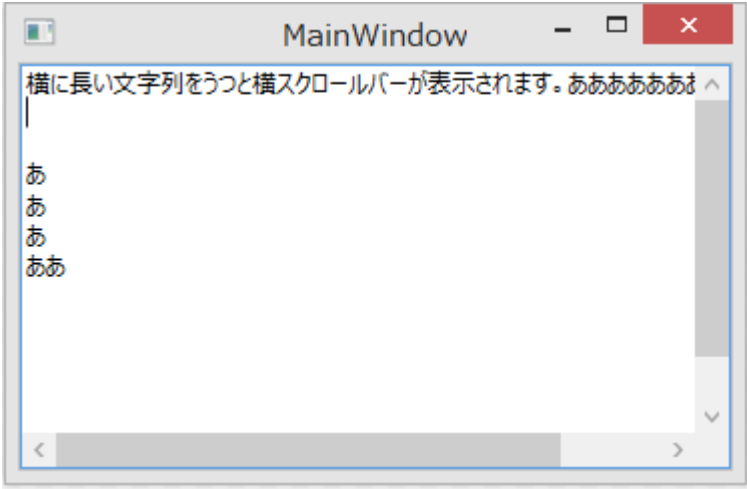
プロパティ	説明
public bool AcceptsReturn { get; set; }	Enter キーを押すことで改行をすることができるかどうかを取得または設定します。デフォルト値は false（改行できない）です。
public bool AcceptsTab { get; set; }	Tab キーを押すことでタブを挿入するかどうかを取得または設定します。デフォルト値は false（タブではなくフォーカス移動をする）です。
public Visibility HorizontalScrollBarVisibility { get; set; }	水平方向のスクロールバーの表示方法を指定します。以下の値が設定可能です。 <ul style="list-style-type: none">● Auto：必要な場合は表示をして不要な場合は非表示になります。● Disable：無効化します。

	<ul style="list-style-type: none">● Visible : 常に表示します。● Hidden : 非表示にします。
public Visibility VerticalScrollBarVisibility { get; set; }	垂直方向のスクロールバーの表示方法を指定します。設定値は HorizontalScrollBarVisibility と同じです。

改行や、タブの挿入が可能で、横スクロールバーと縦スクロールバーが常に表示される TextBox コントロールは以下のように定義します。

```
<TextBox
    AcceptsReturn="True"
    AcceptsTab="True"
    HorizontalScrollBarVisibility="Visible"
    VerticalScrollBarVisibility="Visible"/>
```

実行して文字を打ち込むと以下のように改行や、スクロールバーの表示を確認できます。



4.11. メディア

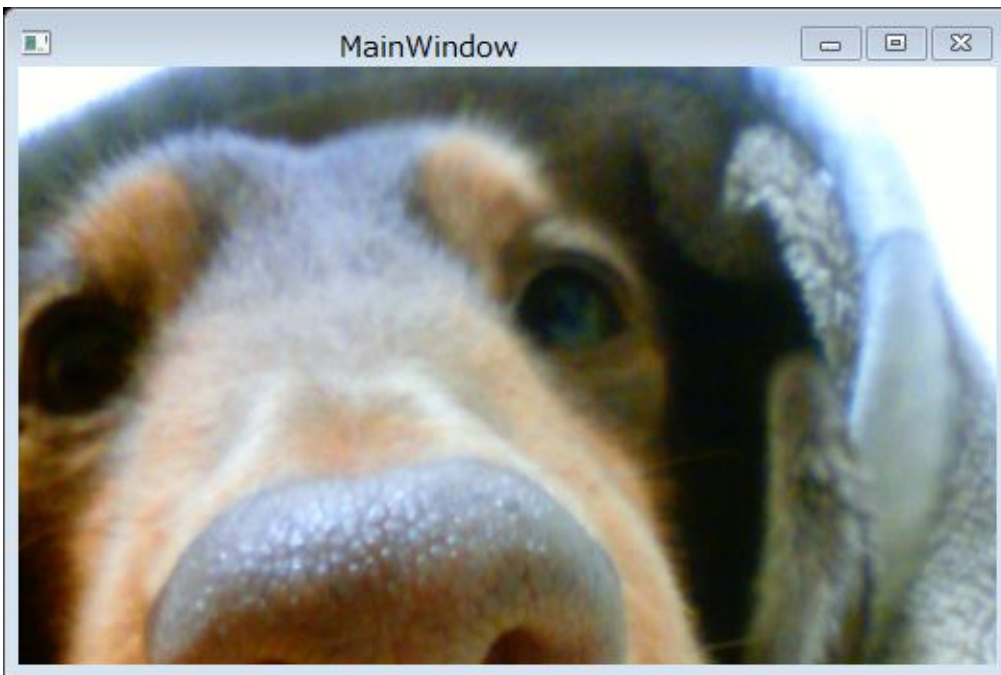
4.11.1. Image コントロール

Image コントロールは、画面に画像を表示するコントロールです。Source プロパティに指定した画像と、Stretch プロパティに指定した画像の拡大方法をもとに、画像を表示します。Stretch プロパティは ViewBox の拡大方法と同じ方法で画像を拡大・縮小します。

Source プロパティは ImageSource 型ですが、XAML からは文字列で画像の URI を指定することが出来ます。プロジェクトにビルドアクション Resource に指定された画像はプロジェクトのルートからの相対 URI で指定可能です。プロジェクトに anthem.jpg という画像がある場合に、それを表示する Image コントロールの XAML は以下のようになります。

```
<Image Source="anthem.jpg" Stretch="UniformToFill" />
```

実行すると以下のように表示されます。



コードから Source プロパティを指定するコードを以下に示します。画面の XAML は以下の通りです。

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Menu>
    <MenuItem Header="開く(_O)" Click="MenuItem_Click" />
  </Menu>
  <Image x:Name="image" Grid.Row="1" Stretch="Uniform" />
</Grid>
```

メニューで画像を開いて、Image に表示するという UI です。MenuItem のクリック時の処理は以下のようになります。処理内容はコメントの通りです。

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    // 画像を開く
    var dialog = new OpenFileDialog();
    dialog.Filter = "画像|*.jpg;*.jpeg;*.png;*.bmp";
    if (dialog.ShowDialog() != true)
    {
        return;
    }

    // ファイルをメモリにコピー
    var ms = new MemoryStream();
    using (var s = new FileStream(dialog.FileName, FileMode.Open))
    {
        s.CopyTo(ms);
    }
    // ストリームの位置をリセット
    ms.Seek(0, SeekOrigin.Begin);
    // ストリームをもとにBitmapImageを作成
    var bmp = new BitmapImage();
    bmp.BeginInit();
    bmp.StreamSource = ms;
    bmp.EndInit();
    // BitmapImageをSourceに指定して画面に表示する
    this.image.Source = bmp;
}
```

実行してファイルを開くと以下のように画面に画像が表示されます。



4.11.2. MediaElement コントロール

MediaElement コントロールは、音楽や動画を再生するためのコントロールになります。Source プロパティに再生したいファイルへの URI を指定して使用します。LoadedBehavior プロパティを Manual にすることで、Play メソッドで再生を行い、Pause メソッドで一時停止を行い、Stop メソッドで停止を行うなどの細かい制御が可能になります。また Volume プロパティや SpeedRatio プロパティによって一般的な動画プレイヤーが備えるべき基本機能を提供します。

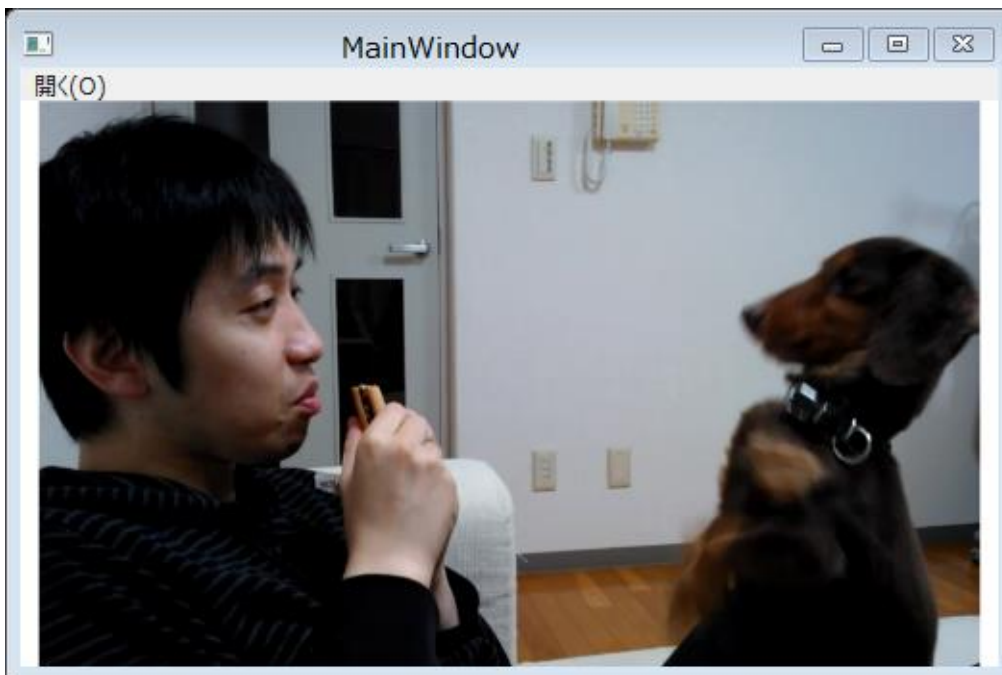
ファイルを開いて再生するコードを以下に示します。

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Menu>
    <MenuItem Header="開く (_O)" Click="MenuItem_Click" />
  </Menu>
  <MediaElement x:Name="mediaElement" Grid.Row="1" />
</Grid>
```

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    // 動画を開く
    var dialog = new OpenFileDialog();
    dialog.Filter = "動画|*.mp4";
    if (dialog.ShowDialog() != true)
    {
        return;
    }

    // SourceにURIを指定して再生する。
    // LoadedBehaviorがPlay(デフォルト値)なので自動再生される。
    var uri = new Uri(dialog.FileName);
    this.mediaElement.Source = uri;
}
```

実行して動画を開いた画面を以下に示します。



5. WPF deep dive

WPF を構成する基本クラス、プロパティシステム、イベントなどについて詳細に見ていきたいと思います。

5.1. DispatcherObject

WPF では、他の UI フレームワークと同様に UI を操作するには専用のスレッドから操作をする必要があります。

WPF では、この操作を簡単にするために Dispatcher という仕組みを提供しています。WPF の継承階層を上へ上へたどっていくと DispatcherObject というクラスに必ず当たります。このクラスには Dispatcher というプロパティが定義されていて、このオブジェクトが生成されたスレッドから操作されているかをチェックする仕組みや、オブジェクトが生成されたスレッドで処理のキューイングを行う仕組みを提供しています。

DispatcherObject で提供される主な機能を以下に示します。

プロパティ名	説明
public Dispatcher Dispatcher { get; }	DispatcherObject に紐づけられたスレッドに対応する Dispatcher オブジェクトを取得します。

メソッド名	説明
public void VerifyAccess()	現在のスレッドが DispatcherObject に紐づけられたスレッドかどうかチェックします。チェックの結果異なるスレッドの場合 InvalidOperationException の例外をスローします。
public bool CheckAccess()	現在のスレッドが DispatcherObject に紐づけられたスレッドかどうかチェックします。チェックの結果異なるスレッドの場合 false を返します。

これらの機能の使い方についてのサンプルプログラムを以下に示します。

DispatcherObject は抽象クラスなので、継承してメソッドを 1 つもつクラスを作成しました。メソッド内では、VerifyAccess メソッドを使って有効なスレッドからのアクセスかどうかを確認して、デバッグウィンドウへメッセージを出力しています。

```
public class DrivedObject : DispatcherObject
{
    public void DoSomething()
    {
        // UIスレッドからのアクセスかチェックする
        this.VerifyAccess();
        Debug.WriteLine("DoSomething");
    }
}
```

Window に 3 つのボタンを置いて、UI スレッドからの直接呼出し、UI スレッド以外からの呼び出し、UI スレッド以外から Dispatcher 経由での呼び出しの 3 パターンの呼び出しを確認します。

```
<Window x:Class="DispatcherObjectSample01.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">
    <StackPanel>
        <Button Content="UIスレッドからなのでOK" Click="OKButton_Click" />
        <Button Content="UIスレッド以外から呼ぶのでNG" Click="NGButton_Click" />
        <Button Content="UIスレッド以外からDispatcher経由で呼ぶのでOK"
                Click="DispatcherButton_Click" />
    </StackPanel>
</Window>
```

```
private void OKButton_Click(object sender, RoutedEventArgs e)
{
    // UIスレッドからの普通の呼び出しなのでOK
    var d = new DrivedObject();
    d.DoSomething();
}

private async void NGButton_Click(object sender, RoutedEventArgs e)
{
    // UIスレッド以外からの呼び出しなので例外が出る
    var d = new DrivedObject();
    try
    {
        await Task.Run(() => d.DoSomething());
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
    }
}

private async void DispatcherButton_Click(object sender, RoutedEventArgs e)
{
    // UIスレッド以外だがDispatcher経由での呼び出しなのでOK
    var d = new DrivedObject();
    await Task.Run(async () =>
    {
        if (!d.CheckAccess())
        {
            await d.Dispatcher.InvokeAsync(() => d.DoSomething()); // OK
        }
    });
}
```

実行して上から順番にボタンを押した結果のデバッグウィンドウを以下に示します。

```
DoSomething
```

```
... 省略...
```

```
System.InvalidOperationException: このオブジェクトは別のスレッドに所有されているため、呼び出しスレッドはこのオブジェクトにアクセスできません。
```

```
... 省略...
```

```
DoSomething
```

最初のボタンでは、UI スレッドからの呼び出しなので、エラーもなくデバッグウィンドウに結果が出ています。二番目のボタンでは、UI スレッド以外からの呼び出しで `VerifyAccess` の箇所で `InvalidOperationException` が発生しています。三番目のボタンでは、UI スレッド以外から `Dispatcher` 経由で、呼び出しているためデバッグウィンドウに結果が出ていることが確認できます。

通常の WPF を使った開発でも、UI スレッド以外から UI を直接または間接的に操作することがあります。そんなときは、この `Dispatcher` を使い今回のサンプルプログラムのように呼び出す必要があります。

5.2. WPF のプロパティシステム

`DispatcherObject` の 1 段下に継承階層をおけると、`DependencyObject` というクラスになります。

`DependencyObject` は、WPF で使われた独自のプロパティシステムを実装しています。この独自のプロパティシステムのことを、依存関係プロパティと添付プロパティといいます。

5.2.1. 依存関係プロパティ

依存関係プロパティは、通常の CLR のプロパティと比べて、以下の機能を追加で提供します。

- リソースからの値の取得
- データバインディングへの対応
- スタイルによる値の設定
- アニメーション
- オーバーライド可能なメタデータ
- 親子関係にあるインスタンスでのプロパティ値の継承

5.2.1.1. 依存関係プロパティの定義方法

依存関係プロパティは、`DependencyObject` を直接、または間接的に継承したクラスで定義可能です。定義方法は、`DependencyProperty` クラスの `Register` メソッドを使用します。`DependencyObject` を継承した `Person` クラスに `Name` という依存関係プロパティを定義する方法を以下に示します。

```

public class Person : DependencyObject
{
    public static readonly DependencyProperty NameProperty =
        DependencyProperty.Register(
            "Name", // プロパティ名を指定
            typeof(string), // プロパティの型を指定
            typeof(Person), // プロパティを所有する型を指定
            new PropertyMetadata("default name")); // メタデータを指定。ここではデフォルト値を設定して
る
}

```

Register メソッドを使い DependencyProperty クラスのインスタンスを作成します。作成したインスタンスは public static readonly のフィールドに「プロパティ名 Property」の命名規約で格納します。DependencyProperty の値は、DependencyObject クラスに定義されている GetValue、SetValue メソッドで取得と設定が可能です。上記クラスを使って Name 依存関係プロパティの値の取得と設定をするコード例を以下に示します。

```

// GetValue, SetValueの使用例
var p = new Person();
// 値を取得
Console.WriteLine(p.GetValue(Person.NameProperty));
// 値を設定
p.SetValue(Person.NameProperty, "おおた");
// 値を取得
Console.WriteLine(p.GetValue(Person.NameProperty));

```

実行すると、以下のような出力になります。

default name

おおた

GetValue メソッドと SetValue メソッドを使って依存関係プロパティの値の取得と設定が出来ますが、通常のプロパティの使用方法とかけ離れているため、通常は、以下のような CLR のプロパティのラッパーを作成します。

```

public class Person : DependencyObject
{
    public static readonly DependencyProperty NameProperty =
        DependencyProperty.Register(
            "Name", // プロパティ名を指定
            typeof(string), // プロパティの型を指定
            typeof(Person), // プロパティを所有する型を指定
            new PropertyMetadata("default name")); // メタデータを指定。ここではデフォルト値を設定して
る

    // 依存関係プロパティのCLRのプロパティのラッパー
    public string Name
    {
        get { return (string)GetValue(NameProperty); }
        set { SetValue(NameProperty, value); }
    }
}

```

上記のプロパティを使うと使用する側のコードは自然な C# によるクラスを利用したコードになります。

```

var p = new Person();
Console.WriteLine(p.Name);
p.Name = "おおた";
Console.WriteLine(p.Name);

```

5.2.1.2. デフォルト値の設定

Person クラスの例で示したように、依存関係プロパティは、メタデータを使ってデフォルト値の設定が出来ます。デフォルト値は、全てのクラスで同じインスタンスが使われます。このようにして、大量のインスタンスが生成されたときにメモリをデフォルト値によって無駄に使うことがないようにになっています。その反面、List 型などのような参照型の値の場合同じインスタンスを使うと不都合があるケースがあります。

例えば先ほどの Person クラスに Children という List<Person> 型の依存関係プロパティを追加してデフォルト値に List<Person> 型のインスタンスを指定したとします。


```

public class Person : DependencyObject
{
    // Nameプロパティは省略

    public static readonly DependencyProperty ChildrenProperty =
        DependencyProperty.Register(
            "Children",
            typeof(List<Person>),
            typeof(Person),
            new PropertyMetadata(new List<Person>())); // デフォルト値は共有される

    public List<Person> Children
    {
        get { return (List<Person>)GetValue(ChildrenProperty); }
        set { SetValue(ChildrenProperty, value); }
    }
}

```

このようにすると、2つの Person クラスのインスタンスを作った時に、Children プロパティの値が共有されて不都合がおきてしまいます。

```

// Childrenプロパティの使用
var p1 = new Person();
var p2 = new Person();

p1.Children.Add(new Person());
p2.Children.Add(new Person());

Console.WriteLine("p1.Children.Count = {0}", p1.Children.Count);
Console.WriteLine("p2.Children.Count = {0}", p2.Children.Count);

```

このプログラムの実行結果はどちらも 2 が表示されてしまいます。このような問題を避けるためには、通常のプロパティと同じように、デフォルト値をコンストラクタで行う必要があります。

```
public Person()
{
    // デフォルト値をコンストラクタで指定するようにする
    this.Children = new List<Person>();
}
```

これで問題は起きなくなります。

5.2.1.3. 値の変更の検出

依存関係プロパティのメタデータには、第二引数にプロパティの値に変更があったときに呼ばれるコールバックメソッドを指定することが出来ます。以下のように設定をします。

```
public static readonly DependencyProperty NameProperty =
    DependencyProperty.Register(
        "Name", // プロパティ名を指定
        typeof(string), // プロパティの型を指定
        typeof(Person), // プロパティを所有する型を指定
        new PropertyMetadata(
            "default name", // デフォルト値の設定
            NamePropertyChanged)); // プロパティの変更時に呼ばれるコールバックの設定

private static void NamePropertyChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    Console.WriteLine("Nameプロパティが{0}から{1}に変わりました", e.OldValue, e.NewValue);
}
```

DependencyPropertyChangedEventArgs の OldValue プロパティと NewValue プロパティで変更前、変更後の値の取得が可能です。プロパティの値が変わった時に何か処理をしたいときに使用します。注意点としては、static メソッドで、値が変更されたインスタンスはメソッドの引数に DependencyObject の形で渡されるという点です。値が変更されたインスタンスに何か操作をしたい場合は、引数で渡されたものをキャストして使用します。

5.2.1.4. 値の矯正

依存関係プロパティには、値が有効範囲にあるかどうかを指定する方法があります。メタデータの第三引数に coerceValueCallback という引数を指定することで、値がプロパティにとって正しい範囲にあるかを検証する処理を追加することができます。以下に Person クラスに Age というプロパティを追加して、値の範囲が 0~120 であるように矯正する処理を設定している例を示します。

```

public static readonly DependencyProperty AgeProperty =
    DependencyProperty.Register(
        "Age",
        typeof(int),
        typeof(Person),
        new PropertyMetadata(
            0,
            AgeChanged,
            CoerceAgeValue));

private static object CoerceAgeValue(DependencyObject d, object baseValue)
{
    // 年齢は0-120の間
    var value = (int)baseValue;
    if (value < 0)
    {
        return 0;
    }
    if (value > 120)
    {
        return 120;
    }
    return value;
}

private static void AgeChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    Console.WriteLine("Ageプロパティが{0}から{1}に変わりました。", e.OldValue, e.NewValue);
}

public int Age
{
    get { return (int)GetValue(AgeProperty); }
    set { SetValue(AgeProperty, value); }
}

```

CoerceAgeValue メソッドが値を矯正している処理になります。範囲外の値が設定された場合は、範囲内の値を返しています。この処理がどのように動くか示すためのコードを以下に示します。

```
var p = new Person();  
p.Age = 10;  
p.Age = -10;  
p.Age = 150;
```

実行結果は以下のようになります。

Age プロパティが 0 から 10 に変わりました。

Age プロパティが 10 から 0 に変わりました。

Age プロパティが 0 から 120 に変わりました。

-10 を設定したのに 0 が設定されていることと、150 を設定したのに 120 が設定されていることが確認できます。この値の矯正処理は、プロパティの変更時だけではなく `DependencyObject` の `CoerceValue` メソッドに依存関係プロパティを渡すことでも呼び出すことができます。よく使われる例として、最大値（Maximum）と最小値（Minimum）を指定できるクラスで、このプロパティの値が変わった時に `this.CoerceValue(ValueProperty);` のように値のプロパティを最大値と最小値の範囲内に矯正する処理を呼び出すといったケースがあります。

5.2.1.5. プロパティの妥当性検証

プロパティの値の矯正の他に、不正な値が設定されたときに例外をスローする検証処理を記述する方法も提供されています。これはメタデータではなく、`Register` メソッドの第 5 引数として指定します。値を受け取り、その値が妥当な値の場合は `true` を返し、不正な値の場合は `false` を返すようにします。

Age プロパティが `MinValue`、`MaxValue` の場合に不正な値とするコード例を以下に示します。

```

public static readonly DependencyProperty AgeProperty =
    DependencyProperty.Register(
        "Age",
        typeof(int),
        typeof(Person),
        new PropertyMetadata(
            0,
            AgeChanged,
            CoerceAgeValue),
        ValidateAgeValue);

private static bool ValidateAgeValue(object value)
{
    // MinValueとMaxValueはやりすぎだろ
    int age = (int)value;
    return age != int.MaxValue && age != int.MinValue;
}

```

このようにすると、以下のように MaxValue や MinValue を設定すると ArgumentException の例外がスローされます。

```

var p = new Person();
try
{
    // 不正な値なので例外が出る
    p.Age = int.MinValue;
}
catch (ArgumentException ex)
{
    Console.WriteLine(ex);
}

```

5.2.1.6. 読み取り専用の依存関係プロパティ

これまで見てきた依存関係プロパティは全て読み書きできるものでしたが、読み取り専用の依存関係プロパティも定義できます。読み取り専用の依存関係プロパティは、DependencyPropertyKey というクラスを使用します。

読み取り専用の依存関係プロパティの例を以下に示します。

```
// RegisterReadOnlyメソッドでDependencyPropertyKeyを取得
private static readonly DependencyPropertyKey BirthdayPropertyKey =
    DependencyProperty.RegisterReadOnly(
        "Birthday",
        typeof(DateTime),
        typeof(Person),
        new PropertyMetadata(DateTime.Now));
// DependencyPropertyは、DependencyPropertyKeyから取得する
public static readonly DependencyProperty BirthdayProperty =
    BirthdayPropertyKey.DependencyProperty;

public DateTime Birthday
{
    // getは従来通り
    get { return (DateTime)GetValue(BirthdayProperty); }
    // setはDependencyPropertyKeyを使って行う
    private set { SetValue(BirthdayPropertyKey, value); }
}
```

コメントにある通り、DependencyPropertyKey クラスのインスタンスは DependencyProperty クラスの RegisterReadOnly メソッドを使って取得します。この DependencyPropertyKey クラスのインスタンスは、外部に公開しないように管理します。

DependencyProperty のインスタンスは、DependencyPropertyKey クラスの DependencyProperty プロパティを使って取得します。これは、普通の依存関係プロパティと同じように public static readonly のフィールドで管理します。あとは、GetValue メソッドと SetValue メソッドを使った CLR のプロパティのラッパーを作るのですが、このとき SetValue では DependencyPropertyKey のインスタンスを使って設定を行います。DependencyProperty クラスのインスタンスを使うと例外が発生するので注意してください。また、プロパティの setter は、読み取り専用の依存関係プロパティでは外部に公開しないように管理します。

このように、値の取得には内部で管理している DependencyPropertyKey クラスのインスタンスを使うようにすることで読み取り専用の依存関係プロパティを実現します。DependencyPropertyKey クラスのインスタンスを外部に公開すると、読み取り専用ではなくなってしまうため実装するさいは注意をして行ってください。

5.2.1.7. 拡張されたプロパティメタデータ

依存関係プロパティのメタデータは、PropertyMetadata クラスの他に、PropertyMetadata クラスを継承した UIPropertyMetadata クラスや、FrameworkPropertyMetadata クラスがあります。UIPropertyMetadata クラスは、WPF のアニメーションを無効化にする機能を提供します。UIPropertyMetadata クラスを継承した

FrameworkPropertyMetadata クラスは、FrameworkPropertyMetadataOptions 列挙体によるレイアウトシステムへの影響の有無の設定や値の継承の有無の設定、データバインディングのデフォルト値の設定など WPF のフレームワークレベルの設定をサポートしています。カスタムコントロールを作成する場合などは通常

FrameworkPropertyMetadata クラスを依存関係プロパティのメタデータに使用するといいいでしょう。

FrameworkPropertyMetadata クラスの使用方法の一例として、子要素へ継承するプロパティの定義方法を以下に示します。

```

public class Person : FrameworkElement
{
    public static readonly DependencyProperty FirstNameProperty =
        DependencyProperty.Register(
            "FirstName",
            typeof(string),
            typeof(Person),
            new FrameworkPropertyMetadata(null));

    public string FirstName
    {
        get { return (string)GetValue(FirstNameProperty); }
        set { SetValue(FirstNameProperty, value); }
    }

    public static readonly DependencyProperty LastNameProperty =
        DependencyProperty.Register(
            "LastName",
            typeof(string),
            typeof(Person),
            // 子要素へ継承するプロパティ
            new FrameworkPropertyMetadata(
                null,
                FrameworkPropertyMetadataOptions.Inherits));

    public string LastName
    {
        get { return (string)GetValue(LastNameProperty); }
        set { SetValue(LastNameProperty, value); }
    }

    public void AddChild(Person child)
    {
        this.AddLogicalChild(child);
    }
}

```

FrameworkElement という、DependencyObject を間接的に継承したオブジェクトを継承して Person クラスを作成しています。Person クラスには、名前を表す FirstName 依存関係プロパティと、苗字を表す LastName 依存関係プロパティがあります。苗字は、親子関係にある場合は同じものを使用するのが自然なので、

FrameworkPropertyMetadata を使って、Inherits を指定しています。親子関係の構築は、FrameworkElement クラスの AddLogicalChild メソッドを使って指定しています。

Person クラスの使用例を以下に示します。親と子のインスタンスを作って親のほうにだけ LastName プロパティを指定します。その状態で、親と子の両方の LastName プロパティと FirstName プロパティを表示しています。

```
[STAThread] // FrameworkElement使うのに必要
static void Main(string[] args)
{
    var parent = new Person { FirstName = "taro", LastName = "tanaka" };
    var child = new Person { FirstName = "jiro" };

    parent.AddChild(child);

    Console.WriteLine("{0} {1}", parent.LastName, parent.FirstName);
    Console.WriteLine("{0} {1}", child.LastName, child.FirstName);
}
```

実行すると以下のような結果になります。LastName プロパティの値が継承されていることが確認できます。

tanaka taro

tanaka jiro

ここで紹介した FrameworkPropertyMetadata クラスは、カスタムコントロールを作成する場合でもない限り使用することはありませんが、カスタムコントロールを作成するときには、以下のリンクを読んでおくに役立ちます。

- FrameworkPropertyMetadata クラス
[http://msdn.microsoft.com/ja-jp/library/system.windows.frameworkpropertymetadata\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/system.windows.frameworkpropertymetadata(v=vs.110).aspx)
- フレームワーク プロパティ メタデータ
[http://msdn.microsoft.com/ja-jp/library/ms751554\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/ms751554(v=vs.110).aspx)

5.2.2. 添付プロパティ

添付プロパティは、別の DependencyObject を継承したクラスに対して、任意のプロパティを設定することが出来る機能です。例えば、Grid クラスの Row 添付プロパティや Column 添付プロパティがあります。これは、Grid 内の別コントロールに対して、何行目、何列目に表示するのかを設定するのに使用します。注目すべきなのは、Row

添付プロパティと Column 添付プロパティは Button などには定義されていませんが、Button などの様々なコントロールに設定可能な点です。これが添付プロパティの特徴になります。

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <!-- Grid.Row添付プロパティの設定例 -->
  <Button Grid.Row="0" Content="Button1" />
  <Button Grid.Row="1" Content="Button2" />
</Grid>
```

添付プロパティを定義する方法は、基本的に依存関係プロパティと同じような流れになります。コード例を以下に示します。

```
public static class Sample
{
    // RegisterAttachedメソッドを使って添付プロパティを作成する
    public static readonly DependencyProperty BirthdayProperty =
        DependencyProperty.RegisterAttached(
            "Birthday",
            typeof(DateTime),
            typeof(Sample),
            new PropertyMetadata(DateTime.MinValue));
}
```

依存関係プロパティが Register メソッドを使っていたのに対して添付プロパティは、RegisterAttach メソッドを使用します。メソッドの引数は基本的に Register メソッドと同じです。添付プロパティのもう 1 つの特徴として、定義するクラス自体には DependencyObject クラスの継承は必要ないという点があります。添付プロパティを設定するクラス側で DependencyObject クラスを継承していれば問題ありません。

添付プロパティの値の取得や設定も、依存関係プロパティと同様に GetValue メソッドと SetValue メソッドを使っています。

```
// 依存関係プロパティと同様にSetValue、GetValueで値の設定を取得が可能
var p = new Person();
p.SetValue(Sample.BirthdayProperty, DateTime.Now);
Console.WriteLine(p.GetValue(Sample.BirthdayProperty));
```

GetValue メソッドと SetValue メソッドを使って値の取得や設定を行うのは、非現実的なため、通常は添付プロパティを定義したクラスに Get プロパティ名、Set プロパティ名という名前の静的メソッドを定義します。上記の Birthday 添付プロパティの完全な定義を以下に示します。

```
public static class Sample
{
    // RegisterAttachedメソッドを使って添付プロパティを作成する
    public static readonly DependencyProperty BirthdayProperty =
        DependencyProperty.RegisterAttached(
            "Birthday",
            typeof(DateTime),
            typeof(Sample),
            new PropertyMetadata(DateTime.MinValue));

    // プログラムからアクセスするための添付プロパティのラッパー
    public static DateTime GetBirthday(DependencyObject obj)
    {
        return (DateTime)obj.GetValue(BirthdayProperty);
    }

    public static void SetBirthday(DependencyObject obj, DateTime value)
    {
        obj.SetValue(BirthdayProperty, value);
    }
}
```

このメソッドを使うと、添付プロパティを使うコードは以下のようになります。

```
// 通常はラッパーを使ってアクセスする
var p = new Person();
Sample.SetBirthday(p, DateTime.Now);
Console.WriteLine(Sample.GetBirthday(p));
```

5.3. WPF のイベントシステム

WPF は、イベントも独自の機構を構築しています。WPF のイベントシステムの特徴を説明する前に、なぜその仕組みが必要になるかというシンプルな例を示したいと思います。以下のように Button の中に Button があるシンプルなケースでのイベントについて考えてみます。

```
<StackPanel Margin="10">
  <Button Click="Button_Click">
    <Button Content="Button" />
  </Button>
</StackPanel>
```

外側の Button の Click イベントには、以下のような MessageBox を表示するコードを記述しています。

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

外側のボタンをクリックしたときには、このイベントハンドラが呼び出されて MessageBox が表示されることは予想できますが、ボタンの中に置かれたボタンをクリックしたときにはどうなるでしょうか？ 答えは、MessageBox が表示されます。

WPF は、複雑にコントロールを組み合わせた UI を作ることができます。そのため、上記のような露骨なものではなくても上記の例のようにコントロール内のコントロールによって通常の CLR のイベントではボタンが本来のボタンの役割を果たさない可能性が出てきます。WPF のイベントシステムは、通常のイベントを拡張して、親要素へイベントを伝搬するバブルイベントという仕組みを提供しています。HTML になじみのある人にとってはおなじみの動きです。

バブルイベントとは、イベント発生元でイベントが処理されなかった場合親要素へイベントを伝搬させる機能をもったイベントです。上記のボタンの例では、ボタンの中に置いたボタンがクリックされたときに、中に置いたボタンでイベントが処理されなかったため、親要素のボタンにクリックイベントが伝搬して、親要素のクリックイベントハンドラが呼び出されて MessageBox が表示されるという動きになります。

WPF では、バブルイベントの他にトンネルイベントという形のイベントも提供しています。一般的に Preview という命名規則で始まるイベントがそれになります。バブルイベントが、イベントの発生元から親要素・親要素…へ伝搬していくのに対して、トンネルイベントはルート要素からイベント発生元のオブジェクトの順番でイベントが伝搬

していきます。（ちょうどトンネルイベントの逆の動きになります）トンネルイベントは、ユーザーの入力を処理するイベントに対して、プログラムが処理前に割り込むポイントを提供するために使用されます。そのため、ユーザーの入力を処理するカスタムコントロールを作成する場合以外は、自分で定義することはないでしょう。

これらの、バブルイベントやトンネルイベントなどのように、イベントの発生元だけでなく WPF のコントロールのツリー上の他のオブジェクトに対しても影響を与えるイベントをルーティングイベントと言います。

5.3.1. ルーティングイベントの定義方法

ルーティングイベントの定義は、EventManager の RegisterEvent メソッドを使って行います。定義の例を以下に示します。

```
class Person : FrameworkElement
{
    // イベント名Eventの命名規約のstaticフィールドに格納する
    public static RoutedEvent ToAgeEvent = EventManager.RegisterRoutedEvent(
        "ToAge", // イベント名
        RoutingStrategy.Tunnel, // イベントタイプ
        typeof(RoutedEventHandler), // イベントハンドラの型
        typeof(Person)); // イベントのオーナー
    // CLRのイベントのラッパー
    public event RoutedEventHandler ToAge
    {
        add { this.AddHandler(ToAgeEvent, value); }
        remove { this.RemoveHandler(ToAgeEvent, value); }
    }

    // 子を追加するメソッド
    public void AddChild(Person child)
    {
        this.AddLogicalChild(child);
    }
}
```

基本的には、依存関係プロパティなどと同じで専用の登録メソッドを使ってイベントを登録して、その CLR 用のラッパーを作成するという流れになります。第二引数に、トンネルかバブルかを指定します。一般的にトンネルイベントの名前は Preview イベント名になります。

今回定義したイベントはトンネルイベントなのでイベント発生元から親へ登っていく形になります。以下にイベントを発行するプログラムの例を示します。

```

var parent = new Person { Name = "parent" };
var child = new Person { Name = "child" };

parent.AddChild(child);

parent.ToAge += (object s, RoutedEventArgs e) =>
{
    Console.WriteLine(((Person)e.Source).Name);
};

parent.RaiseEvent(new RoutedEventArgs(Person.ToAgeEvent));
child.RaiseEvent(new RoutedEventArgs(Person.ToAgeEvent));

```

まず、トンネルイベントの挙動を確認するための親子関係を構築しています。そして親のオブジェクトのほうで ToAge イベントハンドラの登録を行っています。ルーティングイベントでは、イベントの発生元が sender とは限りません。（今回の例では sender には parent が入ってきます）イベントの発生元を取得するには、イベント引数の Source プロパティを利用します。今回の例では、イベントの発生元の Name プロパティの値を表示しています。

最後の2行は、parent と child で、イベントの発行を行っています。ルーティングイベントの発行は、RaiseEvent メソッドに RoutedEventArgs を渡す形で行います。RoutedEventArgs は、イベントの種類を表す RoutedEventArgs を受け取ります。

このプログラムを実行すると以下のように表示されます。

```

parent

child

```

child には、イベントハンドラを登録していませんが、親のイベントハンドラが呼び出されてることが確認できます。

5.3.2. イベントのキャンセル

ルーティングイベントは、RoutedEventArgs の Handled プロパティを true にすることで、後続のイベントをキャンセルすることが出来ます。この機能を使うと、トンネルイベントやバブルイベントを途中でインターセプトして後続のイベントの処理をキャンセルすることが出来ます。

5.3.3. 添付イベント

WPF のイベントシステムは、トンネルイベント、バブルイベントがあることを説明しました。このようなイベントがあるとクリックイベントがボタンで発生したとき、Window や Panel 系コントロールでも Click イベントが発生することになります。このような状況に対応するために Window や Panel 系コントロールに全てのオブジェクトの全てのイベントを実装するのは現実的ではありません。WPF では添付イベントという仕組みで、本来そのオブジェクトで定義されていないルーティングイベントを処理する方法を提供しています。

StackPanel に Button の Click イベントを添付イベントとして設定する XAML を以下に示します。

```
<StackPanel Button.Click="StackPanel_Click">
    <Button Content="Button1" />
</StackPanel>
```

基本的に添付プロパティと同じような記述になります。これと同じことをコードで記述する場合は以下のようになります。stackPanel という変数に Button が置いてある StackPanel が入っている場合のコード例です。

```
this.stackPanel.AddHandler(Button.ClickEvent, new RoutedEventHandler(this.StackPanel_Click));
```

5.4. コンテンツモデル

WPF の重要なコントロールの 1 つに ContentControl クラスがあります。このクラスは、Content プロパティに設定された単一の要素を表示するという機能を提供するコントロールです。「2.5 WPF のコンセプト」でも紹介しましたが、このコントロールが、要素を表示する際の詳細なロジックを以下に示します。

- ContentTemplate に DataTemplate が設定されている場合、Content プロパティに ContentTemplate を適用した結果を表示します。
- ContentTemplateSelector に DataTemplateSelector が設定されている場合、Content プロパティに ContentTemplateSelector が返した DataTemplate を適用した結果を表示します。
- Content プロパティに設定された値の型に紐づけられた DataTemplate がある場合、その DataTemplate を適用した結果を表示します。
- Content プロパティが UIElement 型の場合、そのまま表示されます。（UIElement にすでに親がいる場合は例外が出ます）
- Content プロパティに設定された値の型に紐づけられた TypeConverter で UIElement に変換するものがある場合は、変換した結果を表示します。

- Content プロパティに設定された値の型に紐づけられた `TypeConverter` で `String` 型に変換するものがある場合は `String` 型に変換して `TextBlock` にラップして表示します。
- Content プロパティに設定された値の型が `XmlElement` の場合は、`InnerText` プロパティの値を `TextBlock` にラップして表示します。
- Content プロパティに設定された値を `ToString` した結果を `TextBlock` にラップして表示します。

複雑なロジックですが、端的にいうと、可能な限り `UIElement` に変換できるか試した後に、ダメだったら文字列型にして `TextBlock` に格納して表示するというロジックになります。このような処理を `ContentControl` が行ってくれるおかげで、`ContentControl` を継承する `Button` クラスや `Label` クラスや `ListBoxItem` クラスで、以下のような直観的なプログラミングが可能になっています。

`Button` クラスに文字列を表示する場合は以下のように文字列を設定できます。

```
this.button.Content = "こんにちは世界";
```

`Button` クラス内に `Button` を表示する場合も以下のように直接 `Button` を設定できます。

```
this.button.Content = new Button { Content = "ボタンの中のボタン" };
```

5.4.1. DataTemplate

`ContentControl` クラスの `ContentTemplate` プロパティに設定可能な `DataTemplate` について説明します。

`DataTemplate` は、主に `Content` プロパティにオブジェクトが設定されている場合に、どのようにそのオブジェクトを表示するかを定義します。以下に `ListBox` クラスの `ItemTemplate` (`ListBoxItem` の `Content` プロパティに適用される `DataTemplate`) を使ったプログラム例を示します。


```

<Window x:Class="DataTemplateSample01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:DataTemplateSample01"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <ListBox x:Name="listBox">
            <ListBox.ItemTemplate>
                <DataTemplate DataType="{x:Type local:Person}">
                    <Border BorderBrush="Red" BorderThickness="1" Padding="5">
                        <StackPanel Orientation="Horizontal">
                            <Label Content="Name" />
                            <TextBlock Text="{Binding Name}" VerticalAlignment="Center"/>
                            <Label Content="Age" />
                            <TextBlock Text="{Binding Age}" VerticalAlignment="Center"/>
                        </StackPanel>
                    </Border>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>

```

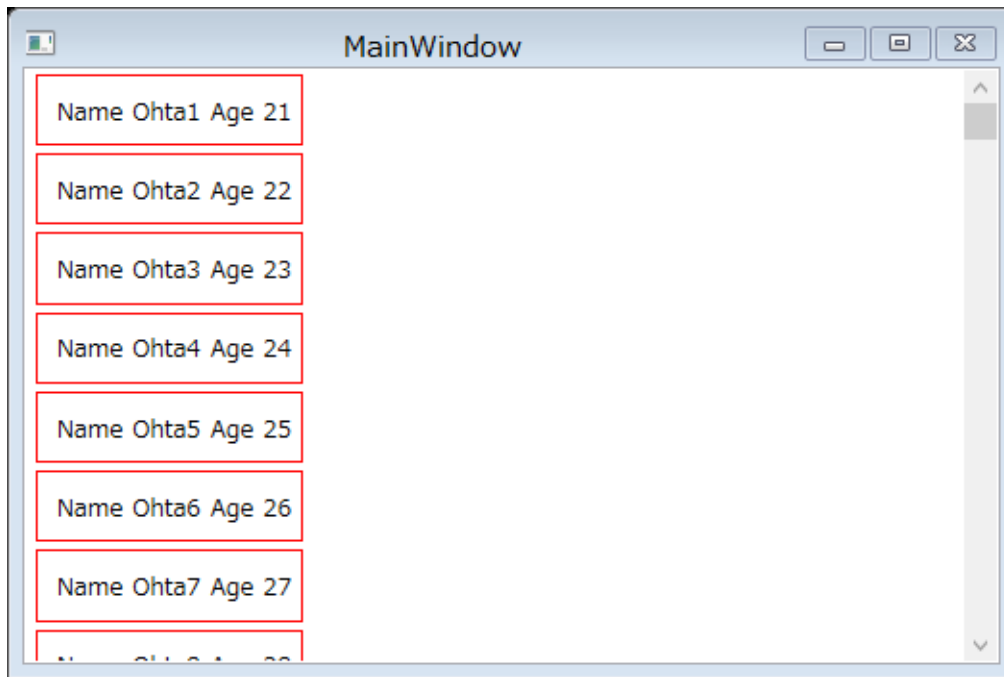
ListBox の ItemsSource プロパティに設定するオブジェクトは以下のように定義しています。

```

namespace DataTemplateSample01
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
}

```

ListBox の ItemsSource プロパティに適切な値を設定して実行すると以下ようになります。



この例では、ListBox の中に DataTemplate を定義していますが、通常は Window や App.xaml 中の Resources に DataTemplate を定義します。こうすることで複数個所で同一のオブジェクトの見た目を再利用することができるようになります。DataTemplate の定義を Window の Resources に移動させた場合のコード例を以下に示します。

```
<Window x:Class="DataTemplateSample01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:DataTemplateSample01"
        Title="MainWindow" Height="350" Width="525">
  <Window.Resources>
    <DataTemplate x:Key="PersonTemplate" DataType="{x:Type local:Person}">
      ...省略...
    </DataTemplate>
  </Window.Resources>
  <Grid>
    <ListBox x:Name="listBox" ItemTemplate="{StaticResource PersonTemplate}" />
  </Grid>
</Window>
```

DataTemplate を Resources に移動して、Resources のオブジェクトを参照するための StaticResource マークアップ拡張で ItemTemplate に DataTemplate を設定しています。Resources に定義された DataTemplate は、

x:Key を指定せずに DataType だけ設定したときに、デフォルトでその型の DataTemplate として使われるという動きをします。そのため、上記の記述は x:Key を使わずに以下のように書くことも出来ます。

```
<Window x:Class="DataTemplateSample01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:DataTemplateSample01"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <DataTemplate DataType="{x:Type local:Person}">
            ...省略...
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <ListBox x:Name="listBox" />
    </Grid>
</Window>
```

5.4.1.1. DataTrigger

DataTemplate には、データの値に応じて表示の見た目を切り替えるロジックを書くことが出来ます。例えば Person クラスを拡張して 40 歳以上の場合 true を返すプロパティを追加します。

```
namespace DataTemplateSample01
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public bool IsOver40 { get { return this.Age >= 40; } }
    }
}
```

この IsOver40 プロパティが true の時は、枠線の色を青色にするということが DataTrigger を使って実現できます。DataTrigger は DataTemplate の Triggers プロパティに設定できて以下のように記述します。

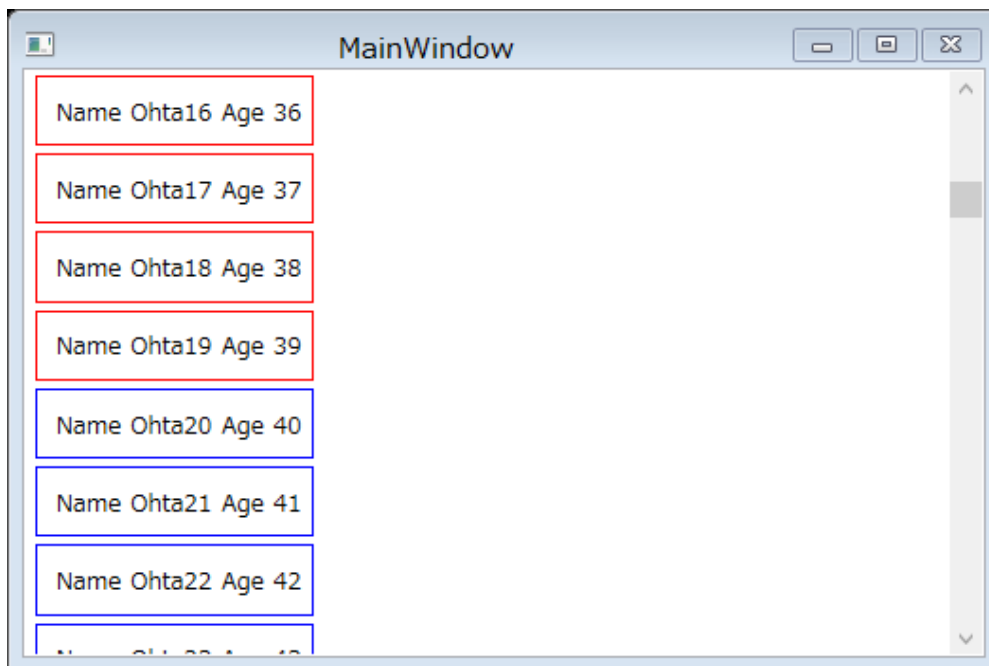
```

<DataTemplate DataType="{x:Type local:Person}">
    <Border x:Name="border" BorderBrush="Red" BorderThickness="1" Padding="5">
        <StackPanel Orientation="Horizontal">
            <Label Content="Name" />
            <TextBlock Text="{Binding Name}" VerticalAlignment="Center"/>
            <Label Content="Age" />
            <TextBlock Text="{Binding Age}" VerticalAlignment="Center"/>
        </StackPanel>
    </Border>
    <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding IsOver40}" Value="True">
            <Setter TargetName="border" Property="BorderBrush" Value="Blue" />
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>

```

DataTrigger の Binding の値と Value の値が等しいとき、DataTrigger の中に定義した Setter が実行されます。Setter はこの例では 1 つだけしか設定していませんが、複数指定できます。

実行すると Age プロパティが 40 を超えると枠線が青色になっていることが確認できます。



5.4.2. DataTemplateSelector

DataTemplateSelector は、条件に応じて DataTemplate を切り替える仕組みです。DataTemplateSelector は、C#で DataTemplateSelector クラスを継承して作成します。DataTemplateSelector クラスを継承して、SelectTemplate メソッドで、状況に応じて適切な DataTemplate を返します。

以下の例では、Person クラスの Age プロパティが 40 より小さい場合は Resources から PersonTemplate1 を検索して返して、Age プロパティが 40 以上の場合は Resources から PersonTemplate2 を検索して返すという処理を行っています。

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

namespace DataTemplateSample02
{
    public class PersonDataTemplateSelector : DataTemplateSelector
    {
        public override DataTemplate SelectTemplate(object item, DependencyObject container)
        {
            var p = (Person)item;
            if (p.Age < 40)
            {
                // Ageが40より小さければPersonTemplate1
                return
                    (DataTemplate)((FrameworkElement)container).FindResource("PersonTemplate1");
            }
            else
            {
                // Ageが40以上ならPersonTemplate2
                return
                    (DataTemplate)((FrameworkElement)container).FindResource("PersonTemplate2");
            }
        }
    }
}
```

PersonTemplate1 と 2 は、Window の Resources に以下のように定義しています。

```

<!-- NameとAgeを表示 -->
<DataTemplate x:Key="PersonTemplate1" DataType="{x:Type local:Person}">
    <StackPanel Orientation="Horizontal">
        <Label Content="Name" />
        <TextBlock Text="{Binding Name}" VerticalAlignment="Center"/>
        <Label Content="Age" />
        <TextBlock Text="{Binding Age}" VerticalAlignment="Center"/>
    </StackPanel>
</DataTemplate>
<!-- Nameだけ表示 -->
<DataTemplate x:Key="PersonTemplate2" DataType="{x:Type local:Person}">
    <StackPanel Orientation="Horizontal">
        <Label Content="Name" />
        <TextBlock Text="{Binding Name}" VerticalAlignment="Center"/>
    </StackPanel>
</DataTemplate>

```

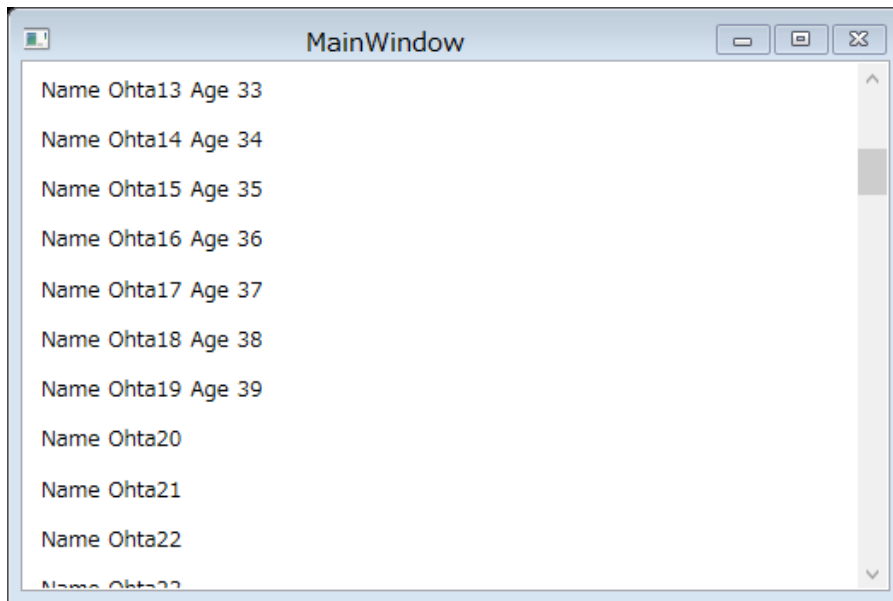
DataTemplateSelector は、ListBox では ItemTemplateSelector プロパティに設定します。Button クラスのような ContentControl を継承しているクラスに指定する場合は ContentTemplateSelector プロパティを使用します。

```

<ListBox x:Name="listBox">
    <ListBox.ItemTemplateSelector>
        <local:PersonDataTemplateSelector />
    </ListBox.ItemTemplateSelector>
</ListBox>

```

ListBox に適当な Person クラスのリストを設定して実行した結果は以下のようになります。適用されているテンプレートが変わっていることが確認できます。



5.5. WPF のアニメーション

WPF は、アニメーションを組み込みでサポートしています。WPF のアニメーションは、指定した依存関係プロパティを指定した時間内で、指定した変化量で、指定した範囲の値を変化させ続ける仕組みになります。単純な WPF のアニメーションの定義例を以下に示します。

```
<Storyboard x:Key="rectAnimation">
  <DoubleAnimation
    Storyboard.TargetName="rect"
    Storyboard.TargetProperty="(Canvas.Left)"
    To="300"
    Duration="0:0:5" />
</Storyboard>
```

アニメーションは通常 Storyboard というものに纏められます。Storyboard は通常 Resources で定義されます。Storyboard の中に ****Animation(****は型名)という値のアニメーションをさせるタグを定義します。上記では Double 型のアニメーションを行う DoubleAnimation を使用しています。Animation には、Storyboard.TargetName 添付プロパティと、Storyboard.TargetProperty 添付プロパティでターゲットとなるオブジェクトとプロパティを指定します。上記の例では、rect という名前で定義されたオブジェクトの Canvas.Left 依存関係プロパティを指定しています。プロパティの指定がカッコで括られているのは、プロパティが添付プロパティであるためです。通常の依存関係プロパティの場合はカッコは必要ありません。

続けて To プロパティで、アニメーションが最終的にいくつ値を設定します。今回の例では省略していますが From を指定することで、どここの値を開始とするか指定することもできます。省略した場合は、現在の値が使われます。最後に Duration で、アニメーションにかかる時間を指定します。書式は時:分:秒です。上記の例では、5 秒を指定しています。今回のアニメーションの定義は、rect という名前のオブジェクトの Canvas.Left プロパティを 5 秒間かけて 300 という値に変更するという意味になります。

アニメーションは定義しただけでは起動しません。アニメーションを開始する方法はいくつかありますが、ここでは、単純な EventTrigger を使う方法を紹介します。EventTrigger は名前の通り、コントロールの Triggers プロパティに指定できるルーティングイベントをきっかけにアニメーションを開始するためのクラスです。以下のように定義して使います。

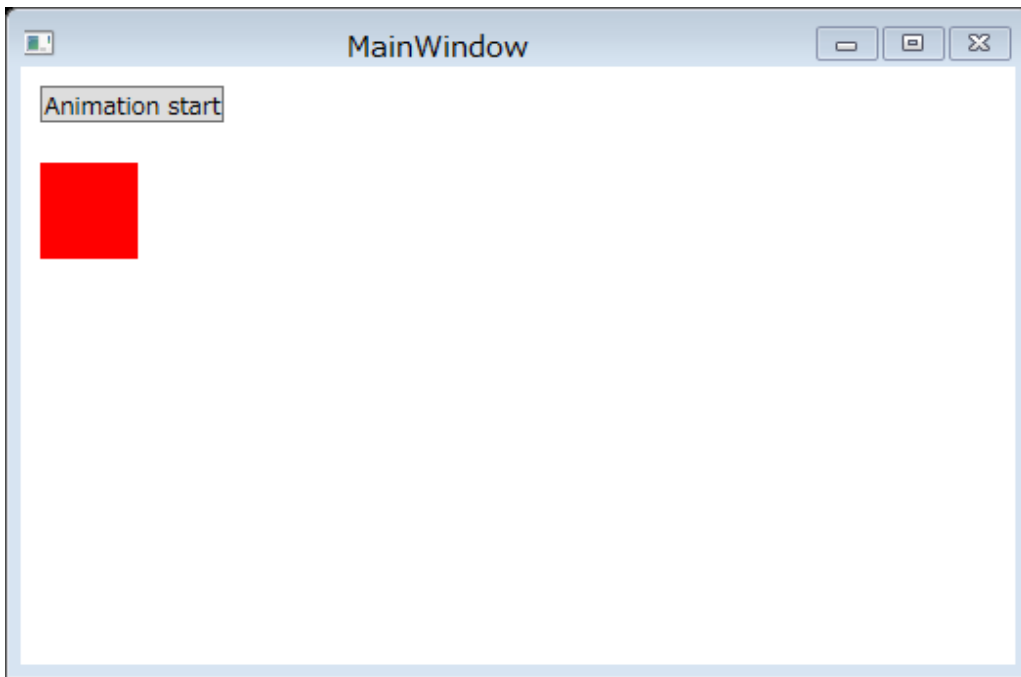
```
<Button Canvas.Top="10" Canvas.Left="10" Content="Animation start">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
      <BeginStoryboard Storyboard="{StaticResource rectAnimation}" />
    </EventTrigger>
  </Button.Triggers>
</Button>
```

上記の例では Button の Click イベントをきっかけに rectAnimation を開始するように定義しています。

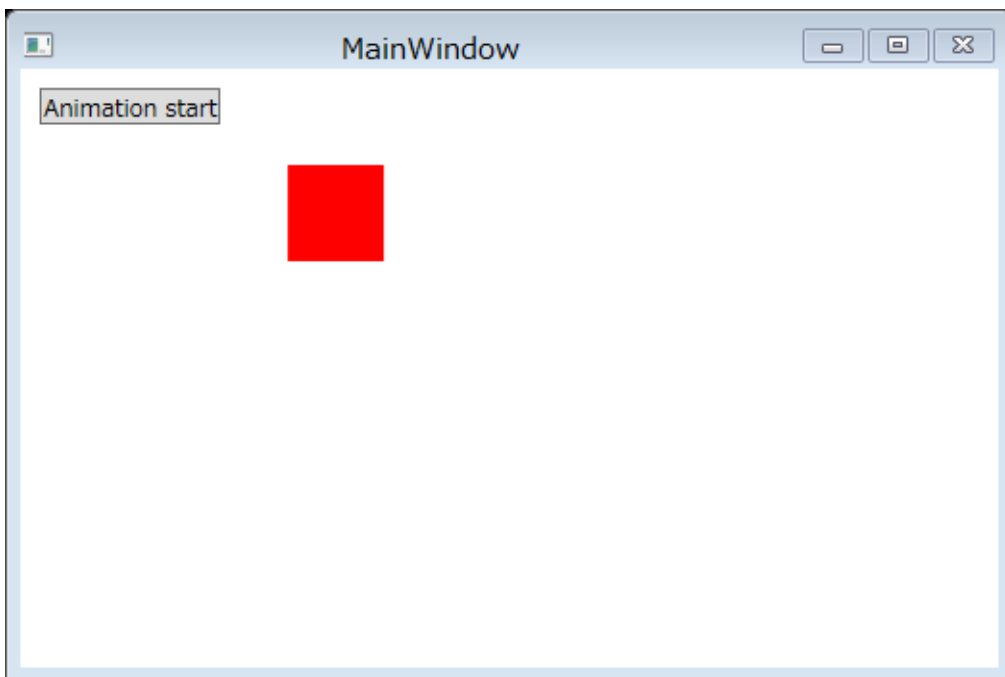
このサンプルの全体の XAML を以下に示します。


```
<Window x:Class="AnimationSample01.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">
    <Window.Resources>
        <Storyboard x:Key="rectAnimation">
            <DoubleAnimation
                Storyboard.TargetName="rect"
                Storyboard.TargetProperty="(Canvas.Left)"
                To="300"
                Duration="0:0:5" />
        </Storyboard>
    </Window.Resources>
    <Canvas>
        <Button Canvas.Top="10" Canvas.Left="10" Content="Animation start">
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.Click">
                    <BeginStoryboard Storyboard="{StaticResource rectAnimation}" />
                </EventTrigger>
            </Button.Triggers>
        </Button>
        <!-- アニメーションのターゲット -->
        <Rectangle
            x:Name="rect"
            Canvas.Top="50" Canvas.Left="10"
            Width="50" Height="50" Fill="Red"/>
    </Canvas>
</Window>
```

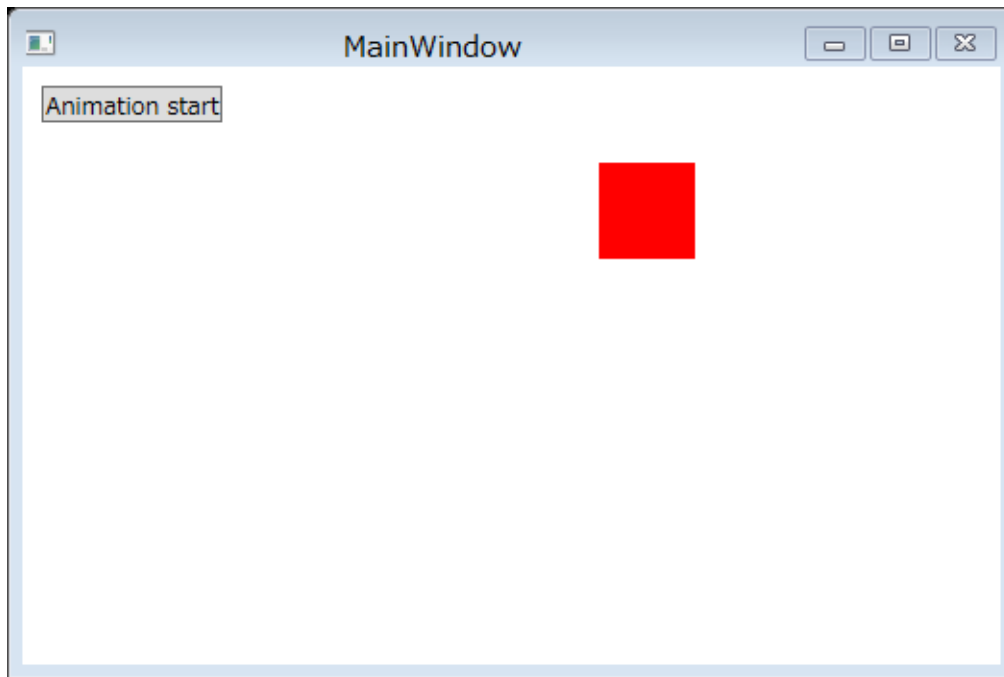
実行結果を以下に示します。起動直後は以下のような配置になっています。



ボタンをクリックすると矩形が右に移動を始めます。



5 秒たつと矩形の移動が終わります。



5.5.1. By による値の指定方法

アニメーションの値の指定方法で一番単純なのは、From と To を指定する方法です。アニメーションの指定方法には From と To 以外に、From と By を使う方法があります。この方法は From（省略可能）を基準として、By だけ変化させるということになります。初期値が 10 で By に 20 を指定した場合、アニメーションが終わった時の値は 30 になります。以下に By を使ったアニメーションの定義例を示します。

```
<Storyboard x:Key="rectAnimationBy">
  <DoubleAnimation
    Storyboard.TargetName="rect"
    Storyboard.TargetProperty="(Canvas.Left)"
    By="100"
    Duration="0:0:5" />
</Storyboard>
```

上記の例は、5 秒かけて 100 だけ Canvas.Left がアニメーションします。

5.5.2. アニメーションの繰り返しの指定

デフォルトでは、アニメーションは 1 度終了すると再度手動で実行するまで停止しています。アニメーションに繰り返しを指定すると、指定した回数、指定した方法でアニメーションを繰り返させることが出来ます。

AutoReverse プロパティを指定すると、アニメーションが終了したあと、逆方向のアニメーションを再生するか指定できます。サンプルで示した矩形が右に移動するアニメーションに指定すると、右に移動が完了したあと元の位置に向かって左方向にアニメーションするようになります。

また、RepeatBehavior にアニメーションを繰り返す時間を指定できます。ここで指定した時間だけ、アニメーションを再生し続けることが出来ます。AutoReverse と組み合わせることで、行ったり来たりというアニメーションを実行させることも出来ます。色に対して指定すると点滅させるという効果も持たせることが出来ます。TimeSpan 型なので Duration プロパティと同様に時:分:秒の書式で指定しますが、Forever を指定することで無限にアニメーションを再生し続けることが出来ます。

以下にアニメーションの繰り返しの指定例を示します。

```
<Storyboard x:Key="rectAnimationRepeat">
  <DoubleAnimation
    Storyboard.TargetName="rect"
    Storyboard.TargetProperty="(Canvas.Left)"
    To="300"
    Duration="0:0:5"
    RepeatBehavior="0:0:13"
    AutoReverse="True"/>
</Storyboard>
```

この例ではアニメーションが 13 秒間繰り返しを続けます。

5.5.3. その他の型のアニメーション

ここまでは DoubleAnimation を例にアニメーションを説明してきました。WPF のアニメーションには Double 型以外の型もサポートしています。以下に主なものを示します。

- ColorAnimation
SolidColorBrush などの色をアニメーションします。
- PointAnimation
Point 型をアニメーションします。

完全なリストは、以下のページを参照してください。

- From/To/By アニメーションの概要
[http://msdn.microsoft.com/ja-jp/library/aa970265\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/aa970265(v=vs.110).aspx)

5.5.4. コードからのアニメーション

WPF のアニメーションはコードで組み立てたり、実行することが出来ます。最初に示した矩形のアニメーションを行う例を C# で書いたものを以下に示します。

```
var storyboard = new Storyboard();

var a = new DoubleAnimation();
// TargetName 添付プロパティではなく、Target 添付プロパティで
// 直接アニメーションのターゲットを指定しています。
Storyboard.SetTarget(a, rect);
Storyboard.SetTargetProperty(a, new PropertyPath("(Canvas.Left)"));
a.To = 300;
a.Duration = TimeSpan.FromSeconds(5);
storyboard.Children.Add(a);

// アニメーションを開始します
storyboard.Begin();
```

コードでアニメーションを組むことで、計算に基づいた複雑なアニメーションを組み立てることが出来ます。

5.5.5. キーフレームアニメーション

これまで説明した型名 Animation で指定するアニメーションの他に、WPF では、キーフレームアニメーションと呼ばれるアニメーションを定義するための型があります。<型名>AnimationUsingKeyFrame という名前で定義されています。サポートされている主な型を以下に示します。

- StringAnimationUsingKeyFrame
- DoubleAnimationUsingKeyFrame
- BooleanAnimationUsingKeyFrame
- ObjectAnimationUsingKeyFrame

完全なリストについては以下のページを参照してください。

- キー フレーム アニメーションの概要

[http://msdn.microsoft.com/ja-jp/library/ms742524\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/ms742524(v=vs.110).aspx)

キーフレームアニメーションの特徴は、1 つのアニメーションの中に KeyFrame という複数のアニメーションのフレームを定義できる点です。1 つのフレームごとにアニメーションの値と時間を指定できるため、1 つのキーフレームアニメーションで、複雑なアニメーションを定義することが出来ます。

以下にキーフレームアニメーションを使ったコード例を示します。

```
<Storyboard x:Key="rectAnimation">
  <DoubleAnimationUsingKeyFrames
    Storyboard.TargetName="rect"
    Storyboard.TargetProperty="(Canvas.Left)"
    Duration="0:0:10">
    <LinearDoubleKeyFrame KeyTime="0:0:2" Value="300" />
    <LinearDoubleKeyFrame KeyTime="0:0:4" Value="0" />
    <LinearDoubleKeyFrame KeyTime="0:0:6" Value="200" />
    <LinearDoubleKeyFrame KeyTime="0:0:8" Value="100" />
    <LinearDoubleKeyFrame KeyTime="0:0:10" Value="300" />
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

キーフレームアニメーションもこれまでやってきたアニメーションと同様に Storyboard の中に定義します。そして、Storyboard.TargetName 添付プロパティ、Storyboard.TargetProperty 添付プロパティでアニメーションのターゲットを指定します。そして、Duration でアニメーションの時間を指定します。普通の Animation クラスでは、From/To/By を指定していましたが、キーフレームアニメーションでは、KeyFrame をアニメーション内に指定します。今回の例では LinearDoubleKeyFrame を指定しています。KeyFrame には KeyTime で、この KeyFrame の Duration 内での時間と、その時の値を指定します。LinearDoubleKeyFrame は、値の間を線形補間します。

上記のアニメーションでは、2 秒かけて初期位置から 300 へ移動して、その後 2 秒かけて 0 へ移動して、その後 2 秒かけて 200 へ移動して、その後 2 秒かけて 100 へ移動して、最後に 2 秒かけて 300 へ移動します。

LinearDoubleKeyFrame 以外に、DiscreteDoubleKeyFrame という KeyFrame を指定することで、間が補間されずに、指定した時間に指定した値に切り替わるという効果を設定できます。

```

<Storyboard x:Key="rectAnimation2">
  <DoubleAnimationUsingKeyFrames
    Storyboard.TargetName="rect"
    Storyboard.TargetProperty="(Canvas.Left)"
    Duration="0:0:10">
    <DiscreteDoubleKeyFrame KeyTime="0:0:2" Value="300" />
    <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="0" />
    <DiscreteDoubleKeyFrame KeyTime="0:0:6" Value="200" />
    <DiscreteDoubleKeyFrame KeyTime="0:0:8" Value="100" />
    <DiscreteDoubleKeyFrame KeyTime="0:0:10" Value="300" />
  </DoubleAnimationUsingKeyFrames>
</Storyboard>

```

この、Discrete 型名 KeyFrame を使うことで、間を補間することができない String や Boolean などといった値に対してもアニメーションを設定することが出来ます。

5.5.6. イー징関数

WPF のアニメーションには、アニメーションの変化量に数式を適用して特殊な効果を与える機能があります。以下に適用可能な効果の一覧を MSDN から抜粋して紹介します。

- BoundEase : 弾むようなバウンド効果を作成します。
- CircleEase : 円関数を使って加速と減速のアニメーションを作成します。
- ElasticEase : バネが伸び縮みしながら最終的に停止するアニメーションを作成します。
- SineEase : サイン式を使って加速と減速のアニメーションを作成します。

完全なリストは MSDN の以下のページを参照してください。

- イー징関数

[http://msdn.microsoft.com/ja-jp/library/ee308751\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/ee308751(v=vs.110).aspx)

イー징関数を指定するには Animation クラスの EasingFunction に指定します。KeyFrame の場合は、KeyFrame の EasingFunction に指定します。DoubleAnimation クラスに BoundEase を指定した例を以下に示します。

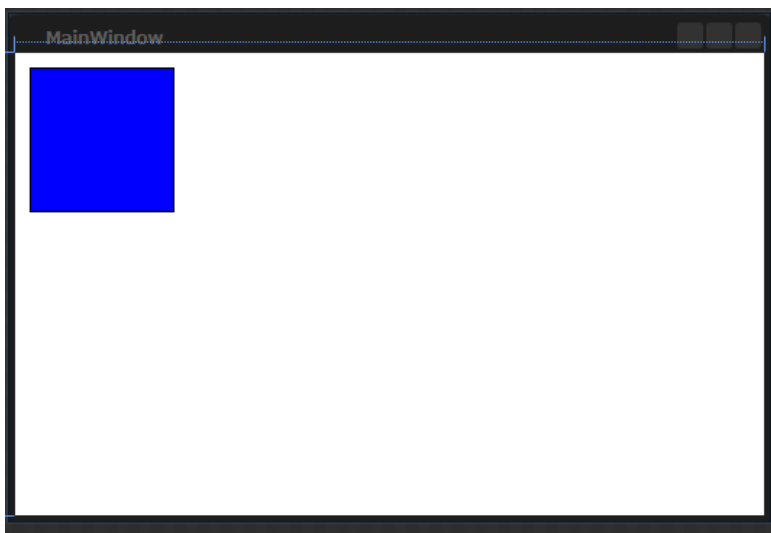
```
<Storyboard x:Key="rectAnimation">
  <DoubleAnimation
    Storyboard.TargetName="rect"
    Storyboard.TargetProperty="(Canvas.Left)"
    Duration="0:0:5"
    To="300">
    <DoubleAnimation.EasingFunction>
      <BounceEase EasingMode="EaseOut" />
    </DoubleAnimation.EasingFunction>
  </DoubleAnimation>
</Storyboard>
```

上記のアニメーションは、5 秒間で 300 に向けてバウンドするようにアニメーションします。

5.5.7. Blend によるアニメーションの作成

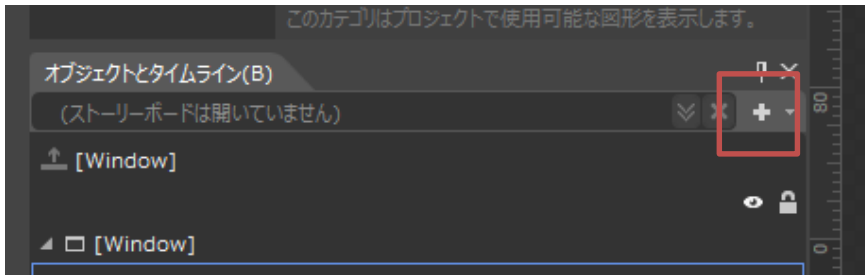
XAML を使用してアニメーションを作成することは出来ませんが、複雑なアニメーションになると手書きによるアニメーションは現実的ではなくなってきます。WPF には、Blend for Visual Studio というデザイナー向けのツールがあります。このツールには、Visual Studio にない機能として、アニメーションの作成機能があります。ここでは、簡単にアニメーションの作成機能について紹介します。

以下のように画面に Rectangle を 1 つ置いた状態を前提として操作説明を行います。

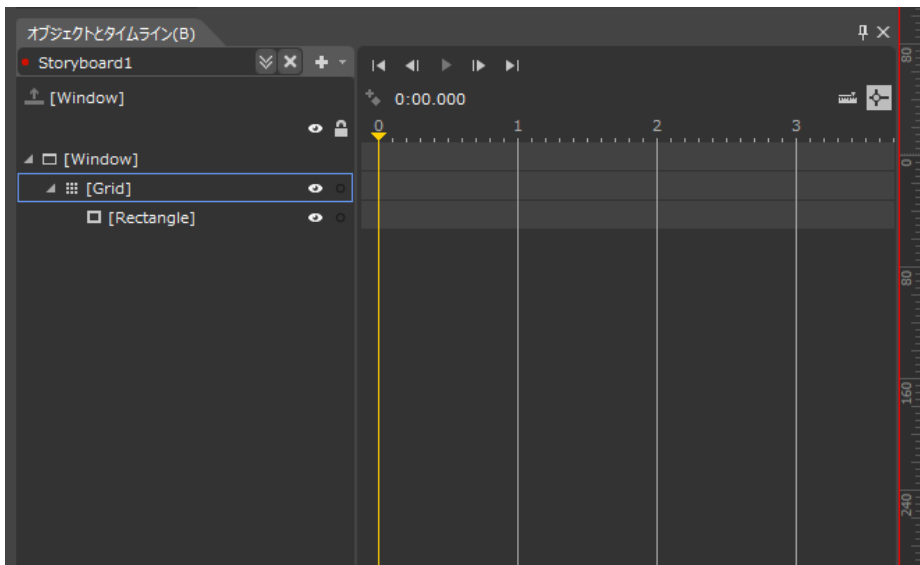


5.5.7.1. 単純なアニメーション

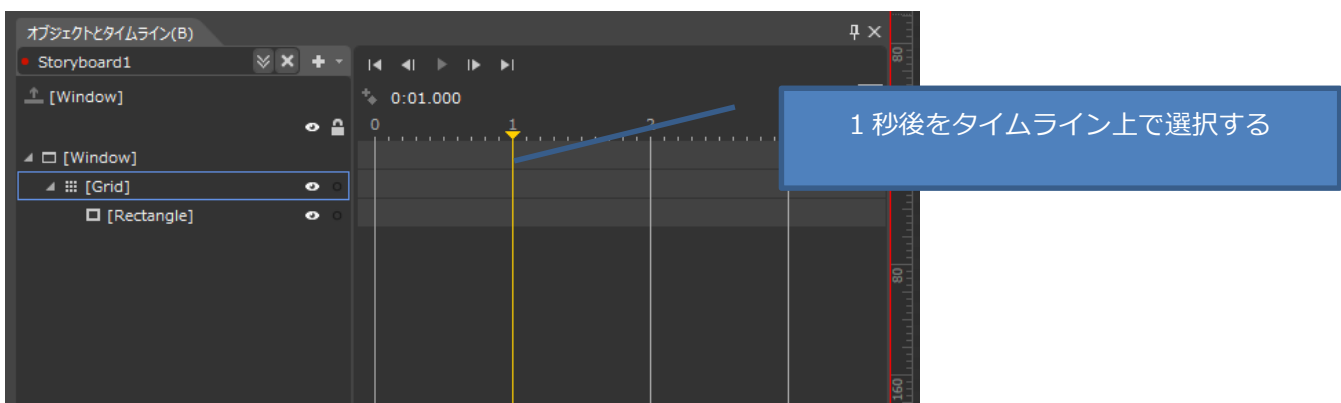
アニメーションを作成するには、「オブジェクトとタイムライン」ウィンドウの+ボタンを押して Storyboard を新規作成します。



+ボタンをクリックすると Storyboard の名前を付けるダイアログが出てくるので任意の名前を入力してください。そうすると、アニメーションを作成する画面になります。デザイナーが赤枠で囲まれ、オブジェクトとタイムラインウィンドウが以下のような見た目になります。



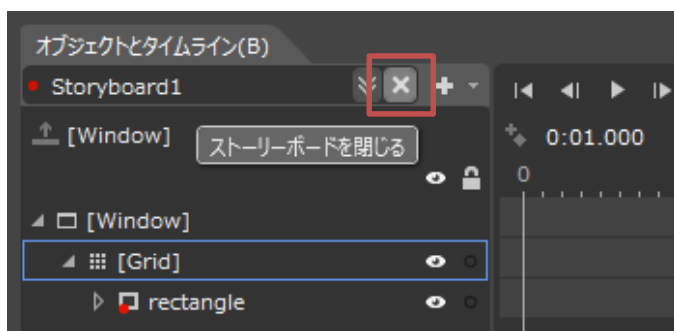
例として 1 秒後に画面に置いた Rectangle を右に移動させる場合のアニメーションの作成方法を示します。まず、オブジェクトとタイムラインウィンドウのタイムラインで、1 秒後を選択します。



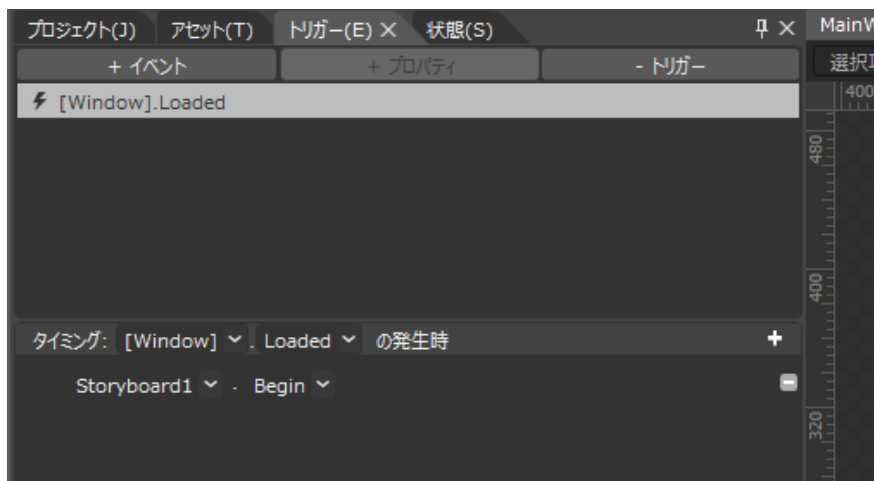
その状態で、デザイナー上で Rectangle を右に移動させます。



オブジェクトとタイムラインウィンドウでストーリーボード名の右にある×ボタンをクリックしてストーリーボードを閉じます。

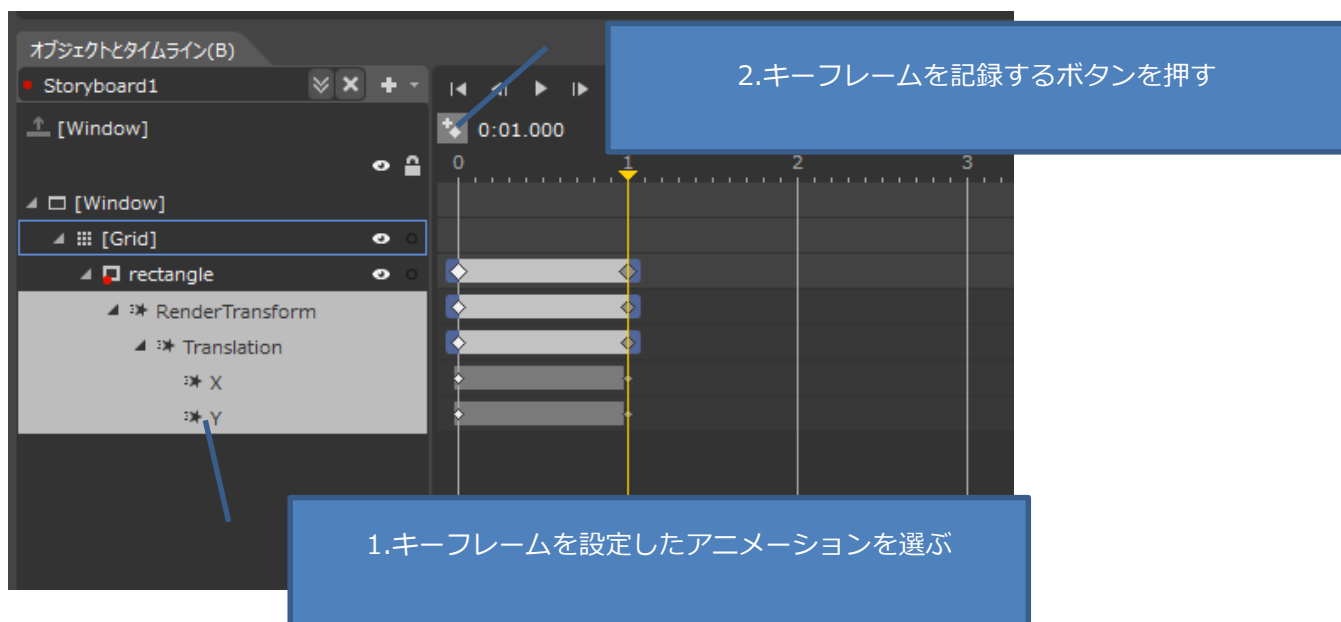


Storyboard の起動はトリガーウィンドウで設定されています。デフォルトでは Window の Loaded イベントに割り当てられています。画面にボタンを置いて Button の Click などに割り当てることでアニメーションの起動タイミングを変えることができます。

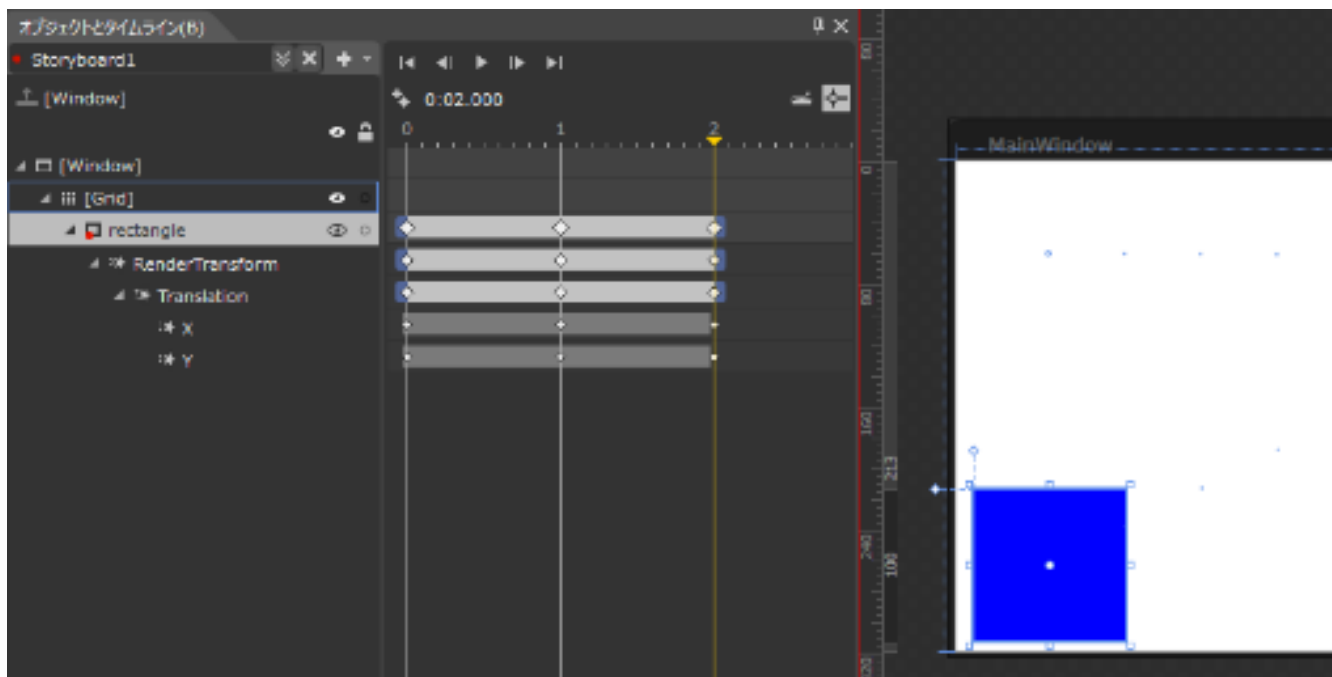


5.5.7.2. キーフレームアニメーション

オブジェクトとタイムラインウィンドウで、キーフレームを設定したいアニメーション（この場合 Rectangle の X と Y 方向の移動）を選んで、キーフレームを記録するボタンを押すと、その時点でキーフレームが作成されます。



キーフレームが作成されたら、そこから先の時間をタイムラインで指定してデザイナーでオブジェクトを移動させたりプロパティを設定することで、次のキーフレームに対してアニメーションを設定できます。下記の例は、タイムラインで 2 秒を選択して、Rectangle を移動させた場合の表示結果です。



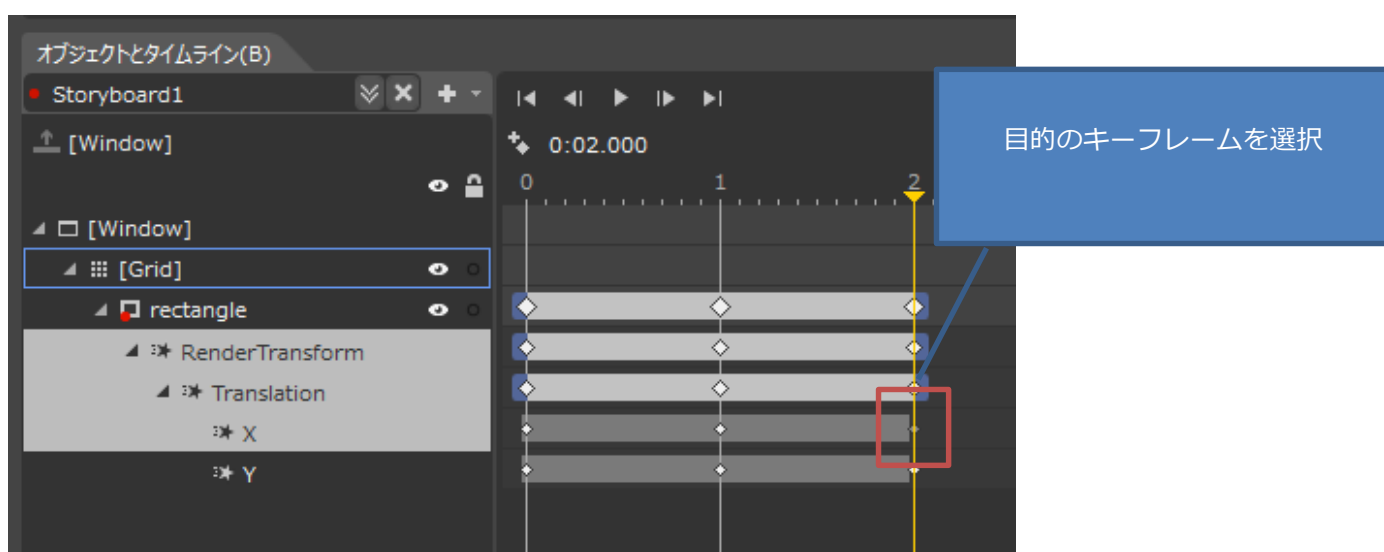
この状態で起動すると、Rectangle がいったん右へ移動したあと、左下へ移動するアニメーションが実行されます。

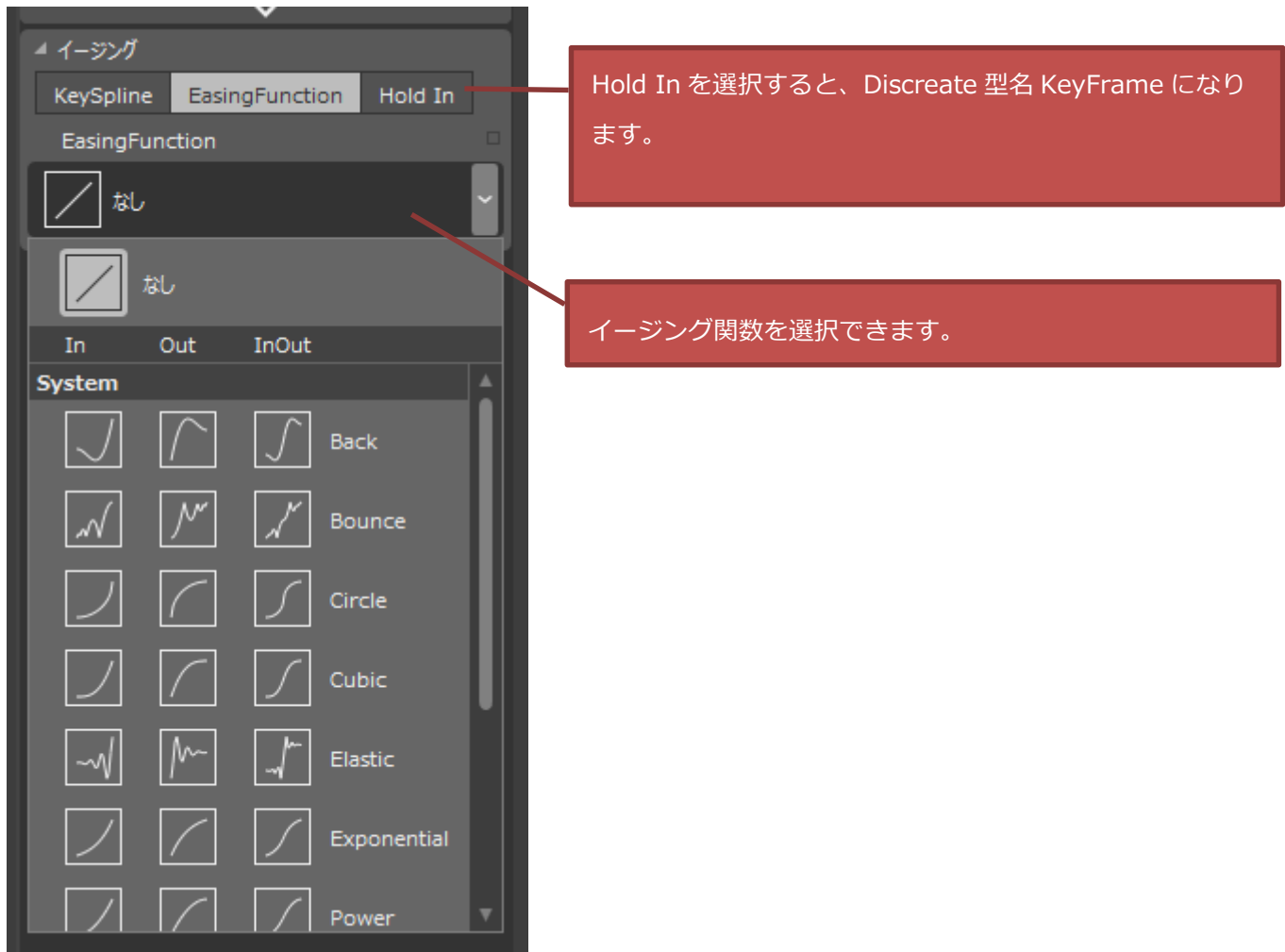
5.5.7.3. 繰り返し設定

オブジェクトとタイムラインウィンドウで、左上の Storyboard を選択すると、プロパティウィンドウで AutoReverse プロパティと RepeatBehavior を設定できるようになります。ここで、自動で繰り返すアニメーションや、永久的に実行されるアニメーションの設定などが出来ます。

5.5.7.4. イージング関数の設定

イージング関数の設定は、設定したいキーフレームや設定したいアニメーションを選択した状態で、プロパティウィンドウから行います。





5.6. Style

Style については、「2.6 スタイル」で簡単に紹介したとおり、コントロールに設定するプロパティの値のセットを集めるためのものです。共通の設定を行うコントロールが多数ある場合は、Style を使うことで手間を軽減できます。

ここでは、まだ説明していない Style の機能を中心に説明していきます。

5.6.1. スタイルの継承

Style は、別のスタイルを元にして新しい Style を作ることが出来ます。Style の継承は、BaseOn というプロパティに元になる Style を指定することで実現出来ます。例えば、基本的なテキストのフォントサイズを 12 にしてフォントを Meiryo UI にするような TextBlock 用の Style を定義して、フォントはそのままに、サイズを 24 にして色を赤にしたいというケースの場合の Style の定義例を以下に示します。

```

<Window.Resources>
  <!-- 継承元のスタイル -->
  <Style x:Key="DefaultTextStyle" TargetType="{x:Type TextBlock}">
    <Setter Property="FontFamily" Value="Meiryo UI" />
    <Setter Property="FontSize" Value="12" />
  </Style>

  <!-- 継承先のスタイル -->
  <Style x:Key="TitleTextStyle" TargetType="{x:Type TextBlock}"
    BasedOn="{StaticResource DefaultTextStyle}">
    <Setter Property="FontSize" Value="24" />
    <Setter Property="Foreground" Value="Red" />
  </Style>
</Window.Resources>

```

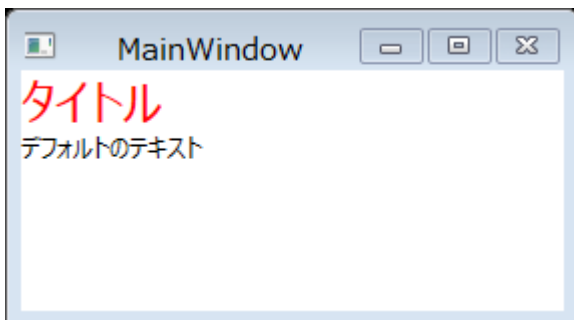
Style を適用する側の TextBlock のコード例を以下に示します。

```

<StackPanel>
  <TextBlock Text="タイトル" Style="{StaticResource TitleTextStyle}" />
  <TextBlock Text="デフォルトのテキスト" Style="{StaticResource DefaultTextStyle}" />
</StackPanel>

```

実行すると、以下のような見た目になります。

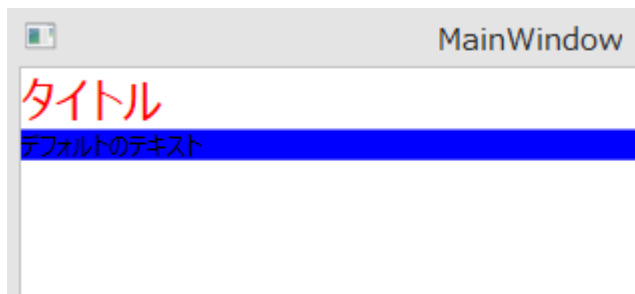


5.6.2. トリガー

Style では、Trigger を使うことでプロパティの値に応じてプロパティの値を変更することが出来ます。例えばマウスが上にあるときに True になる IsMouseOver プロパティが True の時に、背景色を青にするには以下のような Style を記述します。

```
<Style x:Key="DefaultTextStyle" TargetType="{x:Type TextBlock}">
  <Setter Property="FontFamily" Value="Meiryo UI" />
  <Setter Property="FontSize" Value="12" />
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Background" Value="Blue" />
    </Trigger>
  </Style.Triggers>
</Style>
```

この例では、Trigger の中に Setter が 1 つだけですが、複数の Setter を指定することも出来ます。実行して、TextBlock の上にマウスを移動させると以下のように背景色が青色になります。



Trigger の Property に設定かのようなプロパティは、依存関係プロパティなので、その点に注意が必要です。

5.7. リソース

ここでは、リソースについて説明します。WPF のコントロールには、ResourceDictionary 型の Resources というプロパティが定義されています。ResourceDictionary クラスの中には、画像・文字列・オブジェクトなど様々なものを名前をつけて保持することが出来ます。そして、保持しているリソースには XAML やプログラムから参照して使うことが出来ます。

5.7.1. リソースの定義

リソースの定義は、通常 Window や App クラスの Resources プロパティに定義します。以下に App.xaml にブラシのリソースを 2 つ定義する例を示します。

```
<Application x:Class="ResourceSample01.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ImageBrush x:Key="AnthemBrush" ImageSource="anthem.jpg" />
        <SolidColorBrush x:Key="RedBrush" Color="Red" />
    </Application.Resources>
</Application>
```

リソースは、x:Key 属性で名前をつけて定義します。この x:Key で指定した値を元に XAML やプログラム内からリソースにアクセスします。

App.xaml にリソースを定義すると、全ての Window から共通で使用できるというメリットがあります。共通で使用するリソースは App.xaml で定義をして、Window 固有のリソースは Window で定義して利用するのが一般的です。例外的な使い方として、各コントロールの Resources プロパティにリソースを定義する方法もありますが、この場合、そのコントロール内でしか利用できないリソースとなります。

5.7.2. リソースの参照方法

リソースを参照する方法は、大きくわけて 3 つあります。StaticResource マークアップ拡張を使う方法、DynamicResource マークアップ拡張を使う方法、プログラムからアクセスする方法です。

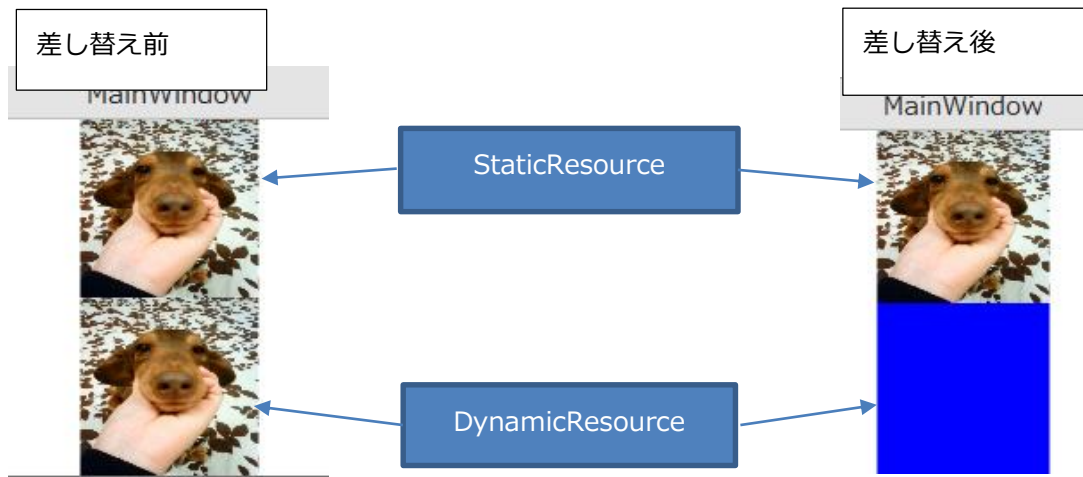
StaticResource マークアップ拡張は、実質的にはリソースの値を代入するのと等価です。DynamicResource マークアップ拡張は、設定したキーのリソースが変更されるかどうかを実行時に監視していて、リソースが変わったタイミングで再代入が行われます。以下のように、Border の Background プロパティに AnthemBrush を StaticResource マークアップ拡張と DynamicResource マークアップ拡張で設定します。

```
<Border x:Name="border" Width="100" Height="100" Background="{StaticResource AnthemBrush}" />
<Border Width="100" Height="100" Background="{DynamicResource AnthemBrush}" />
```

この状態で、以下のように AnthemBrush を置き換えるコードを記述します。

```
App.Current.Resources["AnthemBrush"] = new SolidColorBrush(Colors.Blue);
```

この状態で、プログラムを実行すると、以下のような結果になります。



DynamicResource マークアップ拡張を使用したほうが実行時に動的にリソースの変更に対応できていることが確認できます。DyanmicResource マークアップ拡張を使うと、例えばアプリケーションのテーマの切り替えといったことが可能になります。しかし、必要でない限り StaticResource マークアップ拡張を使うべきです。その理由は、単純に StaticResource マークアップ拡張のほうが DynamicResource マークアップ拡張よりもパフォーマンスが良いからです。

StaticResource マークアップ拡張を使う時の注意点として、単純な代入という特徴から、使用するよりも前でリソースが定義されてないといけないという特徴があります。DynamicResource マークアップ拡張は、このような制約が無いため、どうしても前方でリソースの宣言が出来ないときも DynamicResource マークアップ拡張を使う理由になります。

マークアップ拡張を使う以外にコードからリソースを参照する方法を示します。

```
var brush = (SolidColorBrush)this.FindResource("RedBrush");
this.border.Background = brush;
```

FindResource メソッドを使うことで、起点となるインスタンス（Window など）から App クラスまで親へ親へリソースを探して最初にみつかったものを返します。見つからない場合は例外を返します。存在するかどうか分からないリソースを参照する場合は TryFindResource メソッドを使ってください。

5.8. ControlTemplate

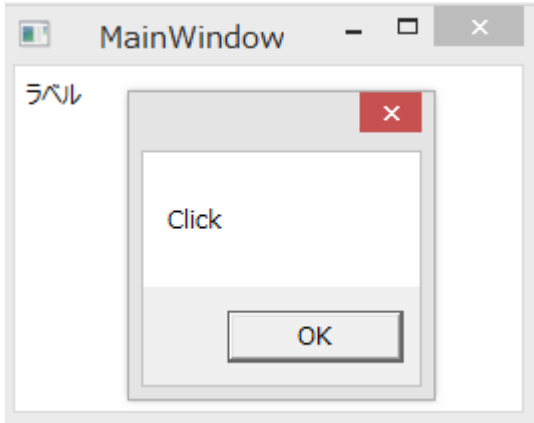
WPF のコントロールは、見た目を完全にカスタマイズする方法が提供されています。コントロールは、Template というプロパティに ControlTemplate を設定することで、見た目を 100% カスタマイズすることが出来るようになっています。

コントロールの Template の差し替え例を示します。WPF の Label コントロールには、Windows Form と異なり Click イベントが提供されていません。ここでは Click 可能な Label の実現のために、Button コントロールの見た目を Label にします。

```
<Button Content="ラベル" Click="Button_Click">
  <Button.Template>
    <ControlTemplate TargetType="{x:Type Button}">
      <Label Content="{TemplateBinding Content}" />
    </ControlTemplate>
  </Button.Template>
</Button>
```

ControlTemplate は、TargetType にテンプレートを適用するコントロールの型を指定します。そして、ControlTemplate の中に、コントロールの見た目を定義します。このとき、TemplateBinding という特殊な Binding を使うことで、コントロールのプロパティをバインドすることが出来ます。上記の例では Button の Content に設定された値を Label の Content に Binding しています。

上記のコードを実行すると以下ようになります。Button の見た目が完全に Label になっていることが確認できます。クリックすると、コードビハインドに記述している `MessageBox.Show("クリック");` が実行されます。



コントロールには、そのコントロールが動作するために必要な構成要素がある場合があります。スクロールバーのバーやバーを左右に移動するためのボタンなど、見た目だけでなく、操作することが出来る要素がそれにあたります。このようなコントロールは、ControlTemplate 内に、コントロールごとに決められた名前で定義する必要があります。どのように定義しているかは、MSDN にある、デフォルトのコントロールテンプレートの例を参照してください。

コントロールのスタイルとテンプレート

[http://msdn.microsoft.com/ja-jp/library/aa970773\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/aa970773(v=vs.110).aspx)

WPF では、Window もコントロールなので、以下のような Window 向けの ControlTemplate を定義して適用することで、ヘッダー部やフッター部に共通の見た目を定義することも簡単に出来ます。

```
<ControlTemplate x:Key="WindowTemplate" TargetType="{x:Type Window}">
  <Border Background="{TemplateBinding Background}" Padding="10">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>

      <Grid>
        <TextBlock Text="System title" FontSize="24" />
        <Button Content="Common command" HorizontalAlignment="Right" />
      </Grid>

      <ContentPresenter Grid.Row="1" Margin="0, 10"/>

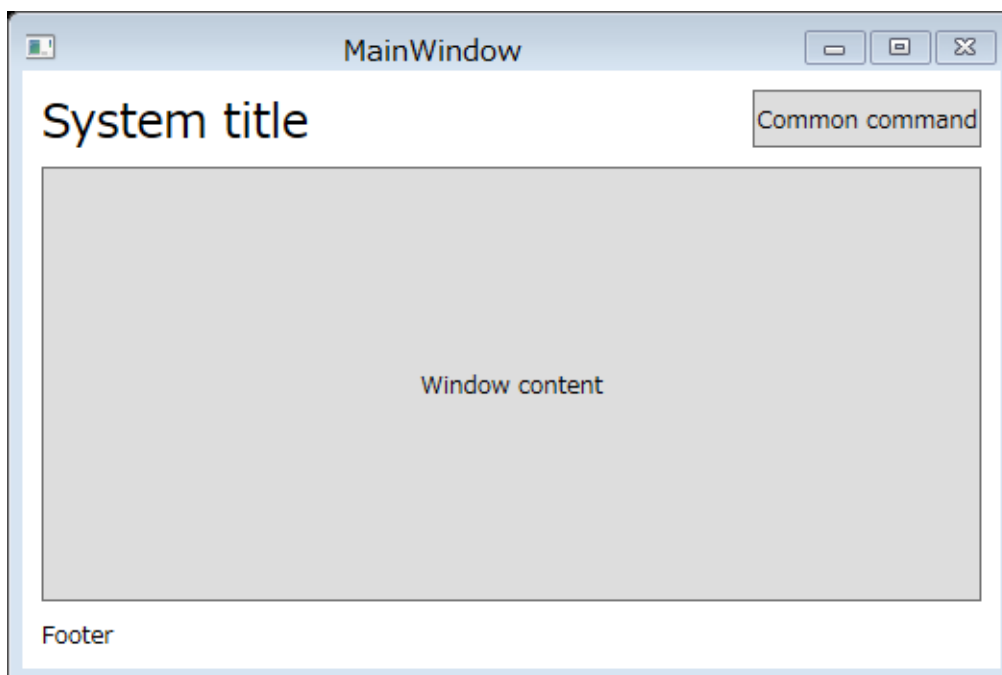
      <Grid Grid.Row="2">
        <TextBlock Text="Footer" />
      </Grid>
    </Grid>
  </Border>
</ControlTemplate>
```

上記の ControlTemplate 内で使用している ContentPresenter は、ContentControl 系のコントロールの ControlTemplate で Content プロパティを表示するのに使用するコントロールになります。

テンプレートを適用した Window の例を以下に示します。

```
<Window x:Class="ControlTemplateSample02.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"
        Template="{StaticResource WindowTemplate}">
    <Grid>
        <Button Content="Window content"/>
    </Grid>
</Window>
```

表示すると以下ようになります。



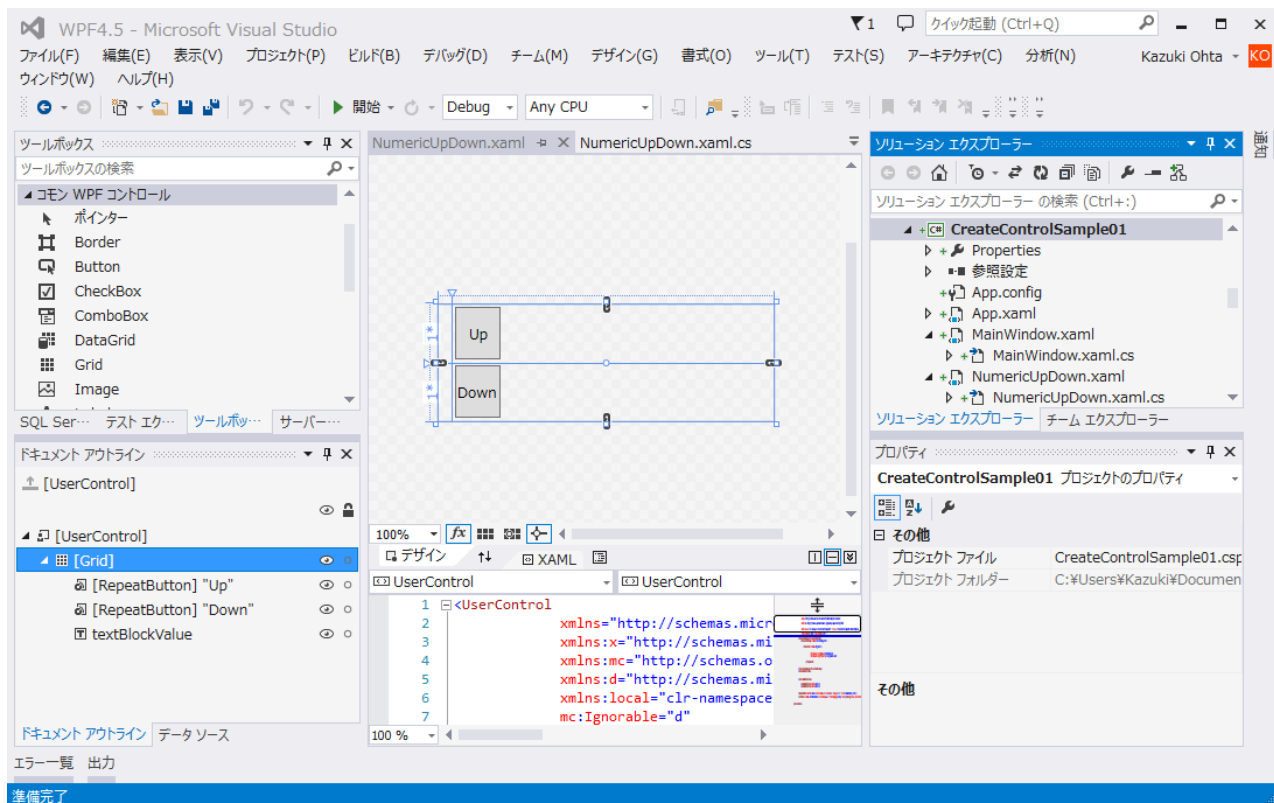
5.9. コントロールの作成

WPF では、コントロールの見た目を少し変えたいといった要求は、Style や DataTemplate、ControlTemplate を使って簡単に実現できます。そのため、Windows Form のころに行っていた、見た目を変えるためのカスタムコントロールの作成は、ほとんど不要になっています。複数のコントロールを組み合わせたコントロールの作成や、独自の動作をするコントロールなど既存のコントロールのカスタマイズで対応できないような要件のみに限られています。

5.9.1. UserControl

UserControl は、複数のコントロールを組み合わせたコントロールを作成するのに向いています。UserControl は、Visual Studio のアイテムテンプレートからユーザーコントロール（WPF）を選択することで作成できます。これまでの Window をベースに開発していたのと同じ要領で、デザイナを使って開発が出来る点が大きな特徴です。

以下に UserControl をデザイナで開いている画面を示します。Window の開発と変わりがないことが確認できます。



UserControl の例として、NumericUpDown コントロールを作成する手順について示します。新規作成からユーザーコントロール（WPF）を選択し、NumericUpDown という名前で作成します。作成したら、以下のように 2 行 2 列の Grid を作り数字を表示するための TextBlock と、数字を増やしたり減らしたりするための RepeatButton を置きます。

```

<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:CreateControlSample01"
  x:Class="CreateControlSample01.NumericUpDown"
  mc:Ignorable="d"
  d:DesignHeight="100" d:DesignWidth="287">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <RepeatButton Content="Up" Grid.Column="1" Margin="2.5" Click="UpButton_Click"/>
    <RepeatButton Content="Down" Grid.Column="1" Grid.Row="1"
      Margin="2.5" Click="DownButton_Click"/>
    <TextBlock x:Name="textBlockValue" Grid.RowSpan="2" TextWrapping="Wrap"
      HorizontalAlignment="Right" VerticalAlignment="Center" Margin="5"
      Foreground="Black"/>
  </Grid>
</UserControl>

```

次に、NumericUpDown の値を保持するための Value 依存関係プロパティを NumericUpDown コントロールに作成します。

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
        "Value",
        typeof(int),
        typeof(NumericUpDown),
        new PropertyMetadata(0));

public int Value
{
    get { return (int)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}
```

画面の TextBlock の Text プロパティと Value プロパティをバインドします。今回は、Binding の RelativeSource というものを使って Binding の元になるオブジェクトを、コントロールのツリーを親へ親へ辿っていった NumeridUpDown コントロールに行きあたるまで探索するように指定しています（FindAncestor）。

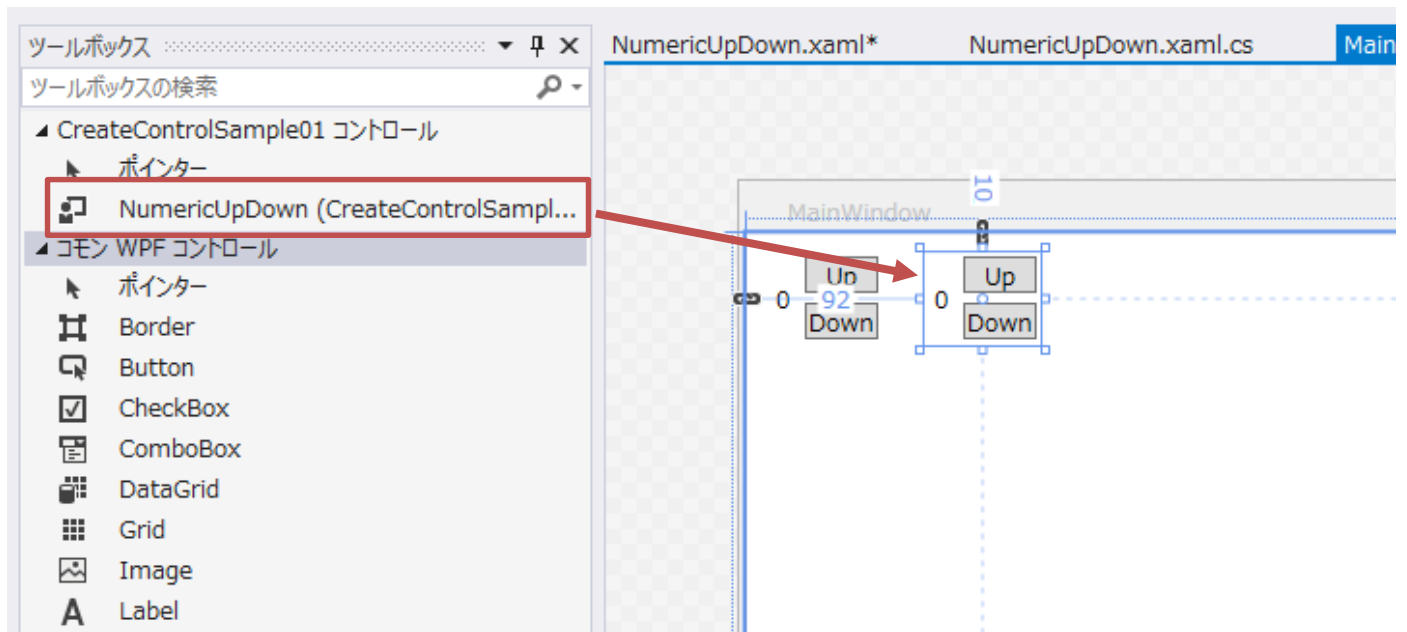
```
Text="{Binding Value, RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type
local:NumericUpDown}}}"
```

そして、RepeatButton のクリックイベントで、Value プロパティの値を操作します。

```
private void UpButton_Click(object sender, RoutedEventArgs e)
{
    this.Value++;
}

private void DownButton_Click(object sender, RoutedEventArgs e)
{
    this.Value--;
}
```

これで、NumeridUpDown コントロールは完成です。UserControl を作成すると、デザイナのツールボックスで自動的に表示されるので、通常のコントロールと同じ要領で画面に配置することが出来ます。



5.9.1.1. VisualStateManager

UserControl で NumericUpDown コントロールを作成したので、ここで、コントロールの状態に応じてアニメーションを行う VisualStateManager という機能について紹介します。VisualStateManager は、見た目の状態を管理する機能です。Style の Trigger などでは IsMouseOver が True の時にアニメーションを実行するといったことが可能でしたが、VisualStateManager は、状態に名前を付けて管理することが出来る点が異なります。状態の遷移はプログラムから行うので、Trigger に比べてより複雑な条件を指定することが出来ます。

VisualStateManager は、VisualStateManager クラスの VisualStateGroups 添付プロパティでコントロールに対して設定します。VisualStateGroups 添付プロパティには、x:Name で名前を付けた VisualStateGroup を設定します。VisualStateGroup の中には、x:Name で名前をつけた VisualState が定義できます。この VisualState の中に Storyboard を設定してアニメーションを定義します。VisualStateGroup の役割ですが、同一の VisualStateGroup 内の VisualState は同時に 1 つしかアクティブになれないという制約があります。逆にいうと、異なる意味を持つ VisualState を別の VisualStateGroup に置くことで、同時に複数の VisualState を有効にするといったことが可能になっています。

ここでは、Value の値がマイナスのときだけ Value の値を赤色にする VisualState を定義したいと思います。VisualStateGroup の名前を NegativePositive にして、その中に Negative と Positive という VisualState を定義します。


```

<UserControl
  ...省略...
  d:DesignHeight="100" d:DesignWidth="287">
  <Grid>
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup x:Name="PositiveNegative">
        <VisualState x:Name="Positive" />
        <VisualState x:Name="Negative">
          <Storyboard>
            <ColorAnimation
              Storyboard.TargetName="textBlockValue"
              Storyboard.TargetProperty="Foreground.Color"
              To="Red" />
          </Storyboard>
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    ...省略...
  </Grid>
</UserControl>

```

Negative の VisualState 内には、TextBlock の値を赤色にするアニメーションを定義しています。VisualState の定義が終わったのでコードビハインドで、VisualState の切り替え処理を書きます。VisualState の切り替えは、VisualStateManager.GoToState メソッドを使います。GoToState メソッドは、VisualState を切り替えるコントロールと、VisualState 名と、VisualState が切り替わるときのアニメーション効果を使用するかどうかを設定します。

Value プロパティの値が書き換わったタイミングで VisualState を切り替えればいいので以下のようなコードになります。

```

public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
        "Value",
        typeof(int),
        typeof(NumericUpDown),
        new PropertyMetadata(0, ValueChanged));

private static void ValueChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    ((NumericUpDown)d).UpdateState(true);
}

...省略...

public NumericUpDown()
{
    InitializeComponent();
    this.UpdateState(false);
}

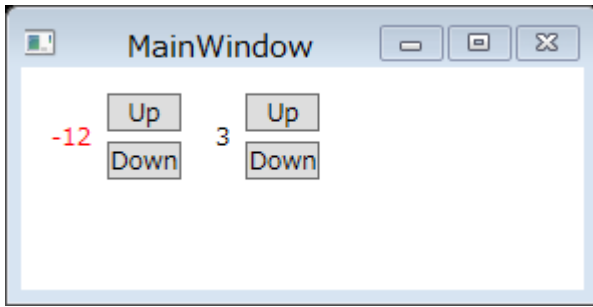
...省略...

private void UpdateState(bool useTransition)
{
    if (this.Value >= 0)
    {
        VisualStateManager.GoToState(this, "Positive", useTransition);
    }
    else
    {
        VisualStateManager.GoToState(this, "Negative", useTransition);
    }
}

```

UpdateState という VisualState を切り替える処理を作り、Value プロパティの変更時と、NumericUpDown コントロールの初期化時に呼び出しています。初期化のときはアニメーション効果は不要なため切り替え効果は false を指定しています。Value プロパティの値が変わった時は切り替え効果を有効にするため true を設定しています。

NumericUpDown コントロールを実行すると以下のようにマイナスのときは赤色になることが確認できます。

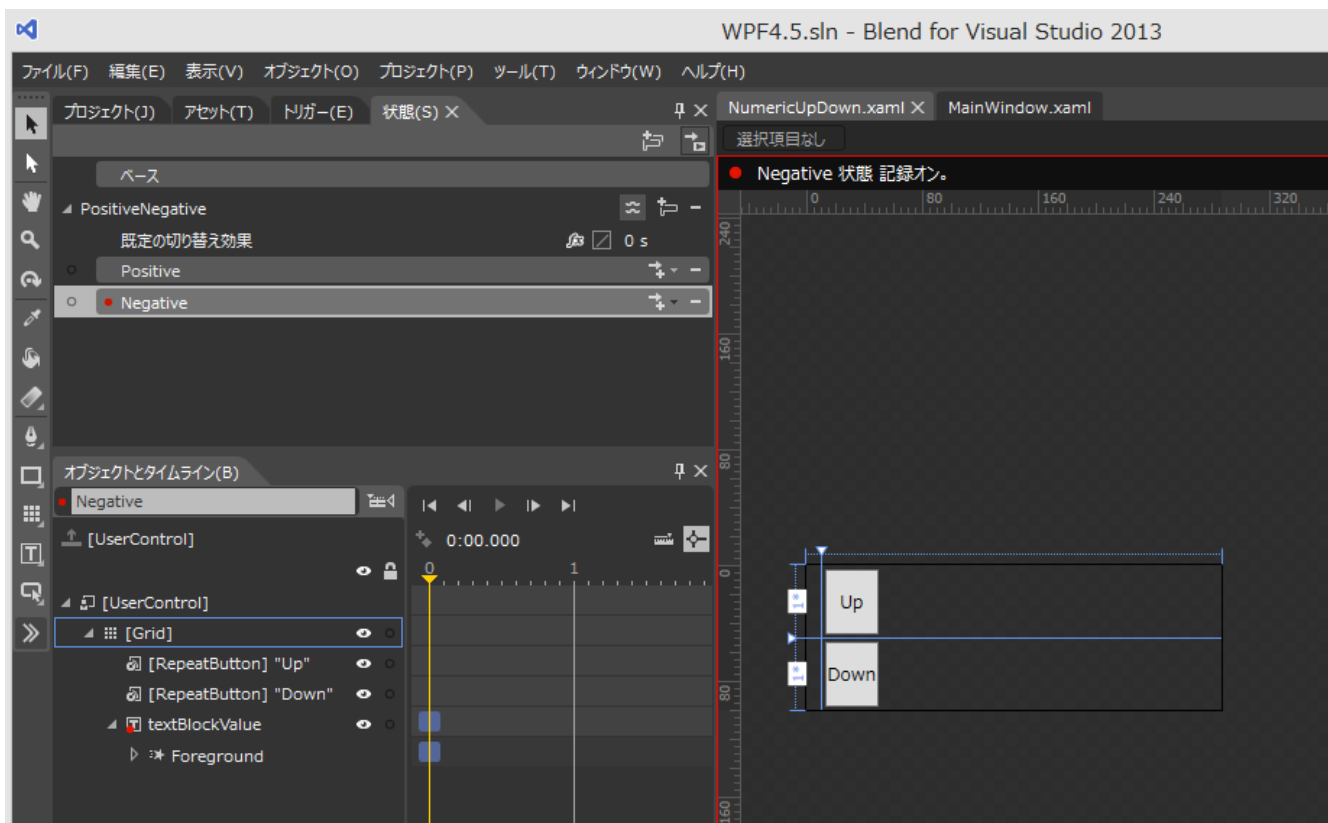


VisualStateManager は、フォーカスの状態に応じた見た目の管理など組み込みのコントロールの様々な個所で使用されています。

5.9.1.2. Blend での VisualStateManager の設定方法

VisualStateManager は、複雑になると手書きするのが大変になってくるため Blend を使って作成するのが一般的です。Blend の状態タブを開くとベースという状態がデフォルトで選択されていて、その下に VisualStateGroup と VisualState を定義できるようになっています。VisualState を選択すると、通常のアニメーション作成と同じ要領で切り替え時の動作を設定できます。

以下の画面例は、NumericUpDown コントロールで Negative の VisualState を選択したときの表示です。



5.9.2. カスタムコントロール

UserControl で、独自コントロールを作る方法を紹介しましたが、UserControl ではできないことがあります。ControlTemplate へ対応です。ControlTemplate へ対応した完全な WPF の独自コントロールを作るには、これから紹介するカスタムコントロールを作成する必要があります。

カスタムコントロールは、新規作成のカスタムコントロール（WPF）から作成します。作成すると、クラスが 1 つと Themes フォルダの中に Generic.xaml が作成されます。この Generic.xaml 内にコントロールのデフォルトの Style を定義してコントロールを作成します。コントロールのデフォルトの Style のキーはクラスの静的コンストラクタで以下のように DefaultStyleKey 依存関係プロパティのデフォルト値を上書きすることで指定されています。

```
static NumericUpDown()
{
    DefaultStyleKeyProperty.OverrideMetadata(typeof(NumericUpDown),
        new FrameworkPropertyMetadata(typeof(NumericUpDown)));
}
```

Generic.xaml は、以下のようにデフォルトの Style（型名がキーの Style）のみが定義されています。

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:CreateControlSample02">
    <Style TargetType="{x:Type local:NumericUpDown}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type local:NumericUpDown}">
                    <Border Background="{TemplateBinding Background}"
                        BorderBrush="{TemplateBinding BorderBrush}"
                        BorderThickness="{TemplateBinding BorderThickness}">
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </ResourceDictionary>
```

ここでは、UserControl と同じように NumericUpDown コントロールを作成していきます。UserControl の時と同じように XAML を定義します。異なる点は、イベントハンドラの紐づけは XAML で行わない点です。カスタムコン

コントロールでは XAML ではなく、C# で指定します。C# から参照するために、RepeatButton には名前をつけています。名前は、カスタムコントロールでは PART_名前という命名規約でつけることが多いです。UserControl と同様に VisualStateManager の定義と、TextBlock の Text プロパティをコントロールの Value プロパティと Binding しています。

```

<ControlTemplate TargetType="{x:Type local:NumericUpDown}">
  <Border Background="{TemplateBinding Background}"
    BorderBrush="{TemplateBinding BorderBrush}"
    BorderThickness="{TemplateBinding BorderThickness}" d:DesignWidth="231"
    d:DesignHeight="86">
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup x:Name="PositiveNegative">
        <VisualState x:Name="Positive" />
        <VisualState x:Name="Negative">
          <Storyboard>
            <ColorAnimation
              Storyboard.TargetName="textBlockValue"
              Storyboard.TargetProperty="Foreground.Color"
              To="Red" />
          </Storyboard>
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="21*" />
        <RowDefinition Height="22*" />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>
      <TextBlock x:Name="textBlockValue" TextWrapping="Wrap"
        Text="{Binding Value, RelativeSource={RelativeSource FindAncestor,
        AncestorType={x:Type local:NumericUpDown}}}"
        Height="Auto" Grid.RowSpan="2" Width="Auto"
        HorizontalAlignment="Right" VerticalAlignment="Center" Foreground="Black"
      />
      <RepeatButton x:Name="PART_UpButton" Content="Up" Grid.Column="1" Height="Auto"
        Width="Auto" Margin="2.5" />
      <RepeatButton x:Name="PART_DownButton" Content="Down" Grid.Column="1" Grid.Row="1"
        Margin="2.5" />
    </Grid>
  </Border>
</ControlTemplate>

```

NumericUpDown コントロールでは、UserControl と同様に Value 依存関係プロパティの定義と、VisualState の切り替えのコードを記述します。

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
        "Value",
        typeof(int),
        typeof(NumericUpDown),
        new PropertyMetadata(0, ValueChanged));

private static void ValueChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    ((NumericUpDown)d).UpdateState(true);
}

public int Value
{
    get { return (int)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}

private void UpdateState(bool useTransition)
{
    if (this.Value >= 0)
    {
        VisualStateManager.GoToState(this, "Positive", useTransition);
    }
    else
    {
        VisualStateManager.GoToState(this, "Negative", useTransition);
    }
}
```

カスタムコントロールの RepeatButton にイベントハンドラの紐づけを行います。これは、カスタムコントロールにテンプレートが適用されたときに呼び出される OnApplyTemplate メソッドで行います。ここでは、古いテンプレートから取得したコントロールの後始末と、新しいテンプレートから取得したコントロールの初期化を行います。ここでは、イベントハンドラの解除と登録がそれにあたります。テンプレートで定義されたコントロールの取得には GetTemplateChild メソッドで名前を指定して取得します。

```
// XAMLで定義されたボタン格納用変数
private RepeatButton upButton;
private RepeatButton downButton;

// ボタンのクリックイベント
private void UpClick(object sender, RoutedEventArgs e) { this.Value++; }
private void DownClick(object sender, RoutedEventArgs e) { this.Value--; }

public override void OnApplyTemplate()
{
    base.OnApplyTemplate();

    // 前のテンプレートのコントロールの後処理
    if (this.upButton != null)
    {
        this.upButton.Click -= this.UpClick;
    }
    if (this.downButton != null)
    {
        this.downButton.Click -= this.DownClick;
    }

    // テンプレートからコントロールの取得
    this.upButton = this.GetTemplateChild("PART_UpButton") as RepeatButton;
    this.downButton = this.GetTemplateChild("PART_DownButton") as RepeatButton;

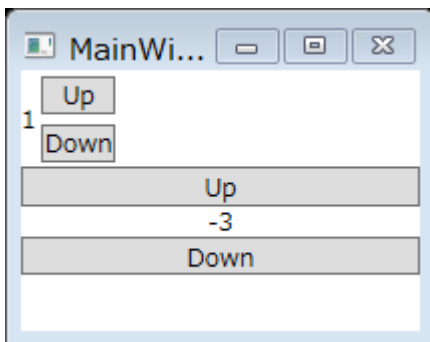
    // イベントハンドラの登録
    if (this.upButton != null)
    {
        this.upButton.Click += this.UpClick;
    }
    if (this.downButton != null)
    {
        this.downButton.Click += this.DownClick;
    }

    // VSMの更新
    this.UpdateState(false);
}
```


このコントロールは、ControlTemplate をサポートした完全なコントロールです。以下のように定義することで見た目のカスタマイズが使用者側で出来るようになっていきます。

```
<StackPanel>
  <!-- 通常の見たい目 -->
  <local:NumericUpDown />
  <!-- コントロールテンプレートの差し替え -->
  <local:NumericUpDown>
    <local:NumericUpDown.Template>
      <ControlTemplate TargetType="{x:Type local:NumericUpDown}">
        <StackPanel>
          <RepeatButton x:Name="PART_UpButton" Content="Up" />
          <TextBlock Text="{Binding Value, RelativeSource={RelativeSource
AncestorType=local:NumericUpDown}}}"
                    HorizontalAlignment="Center"/>
          <RepeatButton x:Name="PART_DownButton" Content="Down" />
        </StackPanel>
      </ControlTemplate>
    </local:NumericUpDown.Template>
  </local:NumericUpDown>
</StackPanel>
```

実行結果を以下に示します。テンプレートが置き換わって Up ボタンと Down ボタンの位置が変わっていることが確認できます。



最後に、コントロールがどんな PART と、VisualState をクラスの属性として定義します。これは、必須ではありませんが、Blend や Visual Studio が使用したり、ドキュメントとして役立つのでカスタムコントロールを作った時にはつけておくことをおすすめします。

TemplatePart 属性で、どの型がどのような名前ですべて ControlTemplate 内に存在することを期待しているかを表します。TemplateVisualState 属性で、どんな VisualStateGroup 内に VisualState が必要なのかを表します。今回作成した NumericUpDown コントロールでは、以下のようになります。

```
[TemplatePart(Type = typeof(RepeatButton), Name = "PART_UpButton")]
[TemplatePart(Type = typeof(RepeatButton), Name = "PART_DownButton")]
[TemplateVisualState(GroupName = "PositiveNegative", Name = "Negative")]
[TemplateVisualState(GroupName = "PositiveNegative", Name = "Positive")]
public class NumericUpDown : Control
```

5.10. Binding

WPF には、見た目とデータを分離して管理するための強力なデータバインディングの機能があります。WPF のデータバインディングは、依存関係プロパティとプロパティの間の同期をとる機能になります。

5.10.1. 単純な Binding

データバインディングは、ソースに設定されたオブジェクトのプロパティとターゲットに設定された依存関係プロパティ（添付プロパティも可）の間の同期をとります。例えば、以下のような Person クラスがあるとします。

```

using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace DataBindingSample01
{
    public class Person : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        private void SetProperty<T>(ref T field, T value, [CallerMemberName]string propertyName =
null)
        {
            field = value;
            var h = this.PropertyChanged;
            if (h != null) { h(this, new PropertyChangedEventArgs(propertyName)); }
        }

        private int age;

        public int Age
        {
            get { return this.age; }
            set { this.SetProperty(ref this.age, value); }
        }

        private string name;

        public string Name
        {
            get { return this.name; }
            set { this.SetProperty(ref this.name, value); }
        }
    }
}

```

このオブジェクトを Window のリソースに登録します。

```

<Window.Resources>
    <local:Person x:Key="Person" Name="tanaka" Age="34" />
</Window.Resources>

```

この Person オブジェクトをソースに指定して TextBlock の Text プロパティに Binding するには以下のように記述します。

```
<TextBlock Text="{Binding Name, Source={StaticResource Person}}" />
```

Binding の最初に指定するのが Path プロパティです。Path にはプロパティ名を指定します。Binding のソースは、指定されていない場合 DataContext プロパティが自動的に使われるため以下のように書くことも出来ます。

```
<Window.DataContext>
    <local:Person Name="tanaka" Age="34" />
</Window.DataContext>
<Grid>
    <TextBlock Text="{Binding Name}" />
</Grid>
```

5.10.2. Binding の Mode

Binding には、値の同期方法を指定するための Mode プロパティがあります。Mode プロパティの値を以下に示します。

モード	説明
OneWay	ソースからターゲットへの一方通行の同期になります。
TwoWay	ソースとターゲットの双方向の同期になります。
OneWayToSource	ターゲットからソースへの一方通行の同期になります。
OneTime	ソースからターゲットへ初回の一度だけ同期されます。

ソースからターゲットへの同期をするには、ソースとなるオブジェクトが INotifyPropertyChanged を実装してプロパティの変更通知を実装している必要があります。ターゲットからソースへの同期は、特に実装すべきインターフェースなどはありません。

Mode は、依存関係プロパティごとにデフォルト値が指定されています。一般的には OneWay が指定されていて、TextBox の Text プロパティなどのように、双方向同期が必要なものについては TwoWay が指定されています。以下のように TextBlock と TextBox を Binding すると、Person オブジェクトを通して TextBox と TextBlock の値が同期されます。

```

<Window.DataContext>
    <local:Person Name="tanaka" Age="34" />
</Window.DataContext>
<StackPanel>
    <TextBlock Text="{Binding Name}" />
    <TextBox Text="{Binding Name}" />
    <Button Content="TextBoxからフォーカス外す用" />
</StackPanel>

```

上記のコードを動かすと、TextBox からフォーカスを外したタイミングで TextBlock と TextBox の値が同期されます。これは TextBox が Binding された値を同期するタイミングがフォーカスを外したタイミングだからです。この動きをカスタマイズするには Binding の UpdateSourceTrigger プロパティを指定します。UpdateSourceTrigger プロパティには以下の値を設定できます。

値	説明
LostFocus	フォーカスが外れたタイミングでソースの値を更新します。
PropertyChanged	プロパティの値が変化したタイミングでソースの値を更新します。
Explicit	UpdateSource メソッドを呼び出して明示的にソースの更新を指示したときのみソースの値を更新します。

先ほどの TextBlock と TextBox に Person オブジェクトを同期した例で、TextBox の Binding を以下のように書き換えると、TextBox に入力した値が即座に Person オブジェクトを経由して TextBlock に反映されます。

```

<Window.DataContext>
    <local:Person Name="tanaka" Age="34" />
</Window.DataContext>
<StackPanel>
    <TextBlock Text="{Binding Name}" />
    <TextBox Text="{Binding Name, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
    <Button Content="TextBoxからフォーカス外す用" />
</StackPanel>

```

5.10.3. ElementName によるソースの指定

Binding は、Source プロパティ指定や DataContext による自動的なソースの指定以外に、いくつかのソースの指定方法があります。その中の 1 つが ElementName による指定です。ElementName は、コントロールをソースとし

て使う時に使用します。ソースに指定したいコントロールの Name プロパティか x:Name で指定された名前と同じものを ElementName に指定します。以下に TextBox をソースに指定して、Text プロパティと TextBlock の Text プロパティをバインドするコード例を示します。

```
<TextBox x:Name="textBox" />
<TextBlock Text="{Binding Text, ElementName=textBox}" />
```

5.10.4. RelativeSource によるソースの指定

RelativeSource は、Binding ターゲットから見た相対的な位置でソースを指定します。例えば自分自身をソースに指定するコード例を以下に示します。自分自身を指定するには RelativeSource に RelativeSource マークアップ拡張の Mode プロパティに Self を指定したものを設定します。(Mode は省略可能です)

```
<TextBlock
    Text="{Binding HorizontalAlignment, RelativeSource={RelativeSource Self}}"
    HorizontalAlignment="Left"/>
```

上記の例は Left と表示されます。

このほかに、自分の親へ親へ辿っていき、指定した型にたどり着くまで遡る AncestorType というものもあります。自分自身が置かれている Window の Title と Binding する例を以下に示します。

```
<TextBlock
    Text="{Binding Title, RelativeSource={RelativeSource AncestorType={x:Type Window}}}" />
```

Window の Title に MainWindow という文字列が設定されている場合、上記の設定で MainWindow と TextBlock に表示されます。

Self と AncestorType 以外に TemplatedParent という TemplateBinding と同様の機能を提供する方法もあります。TemplateBinding が一方通行な Binding なのに対して TemplatedParent を指定した場合は TwoWay などの Binding を指定することが出来る点が異なります。

5.10.5. 入力値の検証

データバインディングには、ターゲットに入力された値を検証する方法も備わっています。歴史的な経緯から、ValidationRule を指定する方法、ソースのプロパティで例外をスローする方法、IDataErrorInfo をソースに実装する方法、INotifyDataErrorInfo をソースに実装する方法のように様々な方法が提供されています。ここでは、デフォ

ルトで有効になっていて、もっとも柔軟な指定が可能な `INotifyDataErrorInfo` インターフェースを実装する方法について解説します。

`INotifyDataErrorInfo` インターフェースは同期、非同期の値の検証を実装するためのインターフェースで以下のように定義されています。

```
public interface INotifyDataErrorInfo
{
    bool HasErrors { get; }
    event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
    IEnumerable GetErrors(string propertyName);
}
```

`HasErrors` プロパティでオブジェクトにエラーが有るか無いかを返します。`ErrorsChanged` イベントでプロパティのエラーの状態に変化があったことを外部に通知します。`GetErrors` メソッドでプロパティのエラーを返します。

`Name` プロパティが必須入力で、`Age` プロパティに 0 以上を設定しないといけない `Person` クラスの実装例を以下に示します。まず、必須のインターフェースの `INotifyPropertyChanged` と、`INotifyDataErrorInfo` のイベントやメソッドなどを実装します。

```

public class Person : INotifyPropertyChanged, INotifyDataErrorInfo
{
    // INotifyPropertyChanged
    public event PropertyChangedEventHandler PropertyChanged;
    private void SetProperty<T>(ref T field, T value, [CallerMemberName] string propertyName =
null)
    {
        field = value;
        var h = this.PropertyChanged;
        if (h != null) { h(this, new PropertyChangedEventArgs(propertyName)); }
    }

    // INotifyErrorsInfo
    private Dictionary<string, IEnumerable> errors = new Dictionary<string, IEnumerable>();
    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
    private void OnErrorsChanged([CallerMemberName] string propertyName = null)
    {
        var h = this.ErrorsChanged;
        if (h != null) { h(this, new DataErrorsChangedEventArgs(propertyName)); }
    }

    public IEnumerable GetErrors(string propertyName)
    {
        IEnumerable error = null;
        this.errors.TryGetValue(propertyName, out error);
        return error;
    }

    public bool HasErrors
    {
        get { return this.errors.Values.Any(e => e != null); }
    }
}

```

これらのメソッドを使って Name プロパティと Age プロパティを実装します。エラーがあれば errors にエラーの内容を追加します。エラーが無い場合はエラーの情報を null にします。そして最後にエラーに変化があったことを通知する ErrorsChanged イベントを発行します。

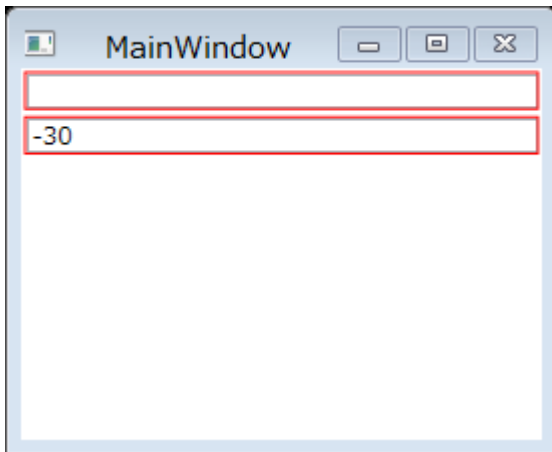

```
public class Person : INotifyPropertyChanged, INotifyDataErrorInfo
{
    private string name;
    public string Name
    {
        get { return this.name; }
        set
        {
            this.SetProperty(ref this.name, value);
            if (string.IsNullOrEmpty(value))
            {
                this.errors["Name"] = new[] { "名前を入力してください" };
            }
            else
            {
                this.errors["Name"] = null;
            }
            this.OnErrorsChanged();
        }
    }

    private int age;
    public int Age
    {
        get { return this.age; }
        set
        {
            this.SetProperty(ref this.age, value);
            if (value < 0)
            {
                this.errors["Age"] = new[] { "年齢は0以上を入力してください" };
            }
            else
            {
                this.errors["Age"] = null;
            }
            this.OnErrorsChanged();
        }
    }
}
```

入力値の検証を追加したオブジェクトをソースにして、Binding を行います。

```
<Window x:Class="DataBindingSample04.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:DataBindingSample04"
        Title="MainWindow" Height="350" Width="525">
    <Window.DataContext>
        <local:Person />
    </Window.DataContext>
    <StackPanel>
        <TextBox Text="{Binding Name}" Margin="2.5" />
        <TextBox Text="{Binding Age}" Margin="2.5" />
    </StackPanel>
</Window>
```

TextBox は、エラー中は自動で赤色の矩形が表示されます。



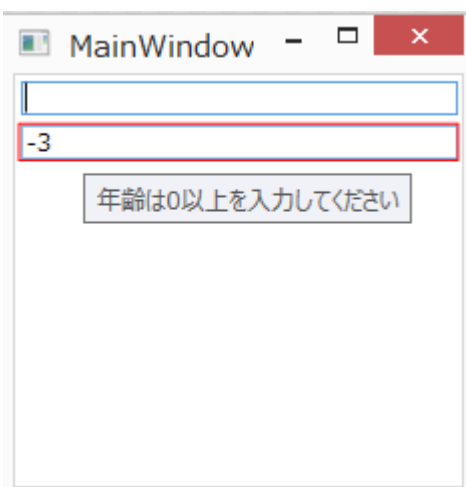
検証エラーの結果は、コントロールに Validation.Errors 添付プロパティに格納されます。Validation.Errors 添付プロパティは、コレクション型で、その中の ErrorContent プロパティに実際のエラーの内容が入っています。Binding の Path を工夫して書くことで、Validation.Errors 添付プロパティに値があるときだけ ToolTip に表示させることが出来ます。以下に記述例を示します。

```

<TextBox Text="{Binding Name}"
    Margin="2.5"
    ToolTip="{Binding (Validation.Errors)/ErrorContent, RelativeSource={RelativeSource Self}}" />
<TextBox Text="{Binding Age}"
    Margin="2.5"
    ToolTip="{Binding (Validation.Errors)/ErrorContent, RelativeSource={RelativeSource Self}}" />

```

ソースを自分自身にして Validation.Errors 添付プロパティを取り出しています。Validation.Errors 添付プロパティの現在選択中の項目を表す/を指定して、さらに、その中の ErrorContent プロパティを指定しています。バリデーションエラーを起こした状態でマウスカーソルを上に移動させると以下ようになります。



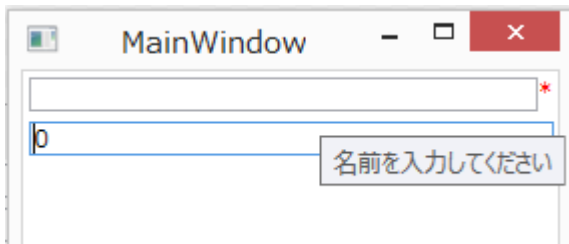
デフォルトの赤い矩形が表示されるエラーを変えたい場合は、Validation.ErrorTemplate 添付プロパティに ControlTemplate を指定します。ControlTemplate 内では、AdornedElementPlaceholder を使って TextBox の表示箇所を指定できます。また、DataContext には Validation.Errors 添付プロパティの値が入ってくるためエラーの内容を簡単に表示することが出来ます。例えば、エラーが起きた時に赤色の * を TextBox の右側に表示して、その ToolTip にエラーの内容を表示する例を以下に示します。

```

<TextBox Text="{Binding Name}"
    Margin="2.5, 2.5, 10, 2.5">
    <Validation.ErrorTemplate>
        <ControlTemplate>
            <DockPanel>
                <AdornedElementPlaceholder />
                <TextBlock
                    DockPanel.Dock="Right"
                    Text="*"
                    Foreground="Red"
                    ToolTip="{Binding /ErrorContent}" />
            </DockPanel>
        </ControlTemplate>
    </Validation.ErrorTemplate>
</TextBox>

```

実行すると以下のように表示されます。



5.10.6. コレクションのデータバインド

データバインディングでは、ここまで説明してきた単一項目のデータバインディングの他に、コレクションをバインディングすることができます。コレクションのデータバインディングは、IEnumerable を実装したコレクションなら、どれでも対象になります。その中でも、INotifyCollectionChanged インターフェースを実装したコレクションは、追加・削除などの変更操作をデータバインディングのターゲットと同期をとることが出来ます。

INotifyCollectionChanged インターフェースの実装は大変な作業なので、デフォルトで ObservableCollection という実装クラスが提供されています。特に理由がなければ、WPF でコレクションのデータバインディングを行う場合は ObservableCollection を使います。

Name と Age プロパティを持つ Person クラスのコレクションを ListBox にデータバインディングする例を以下に示します。DataContext に Person クラスのコレクションを MainWindow クラスのコンストラクタで設定します。

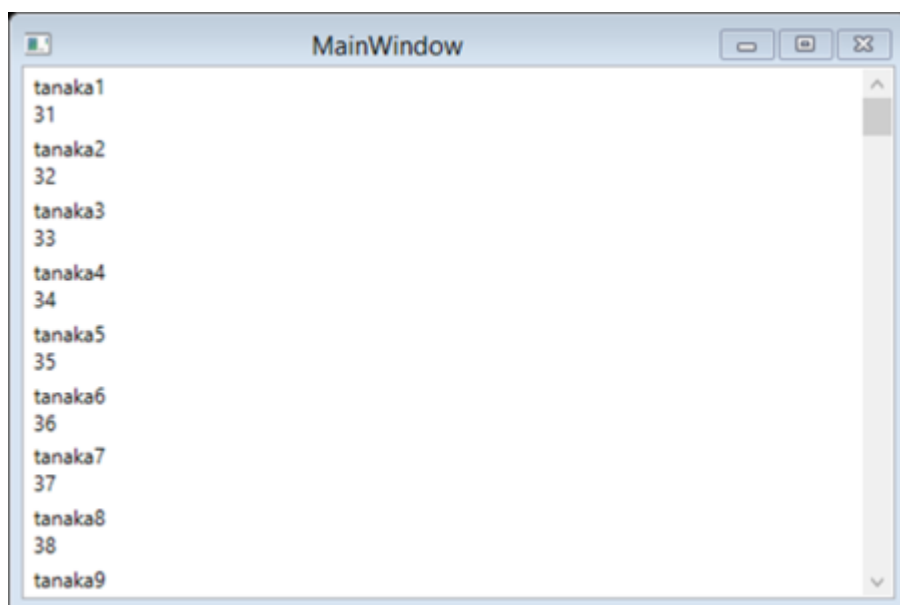
```
using System.Collections.ObjectModel;
using System.Linq;
using System.Windows;

namespace CollectionBindingSample01
{
    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        private ObservableCollection<Person> people;
        public MainWindow()
        {
            InitializeComponent();
            // DataContextにPersonのコレクションを設定する
            this.people = new ObservableCollection<Person>(Enumerable.Range(1, 100)
                .Select(x => new Person
                {
                    Name = "tanaka" + x,
                    Age = (30 + x) % 50
                }));
            this.DataContext = people;
        }
    }
}
```

XAML では、DataContext のコレクションを ItemsSource に Binding して、ItemTemplate で整形して出力します。

```
<Window x:Class="CollectionBindingSample01.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>
        <ListBox
            ItemsSource="{Binding}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel>
                        <TextBlock Text="{Binding Name}" />
                        <TextBlock Text="{Binding Age}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>
```

表示は以下のようになります。



コレクションのデータバインディングは、基本的に、このように ItemsSource プロパティにバインドして、ItemTemplate で見た目を整える形になります。詳しくは、ListBox や DataGrid の解説の箇所を参照してください。

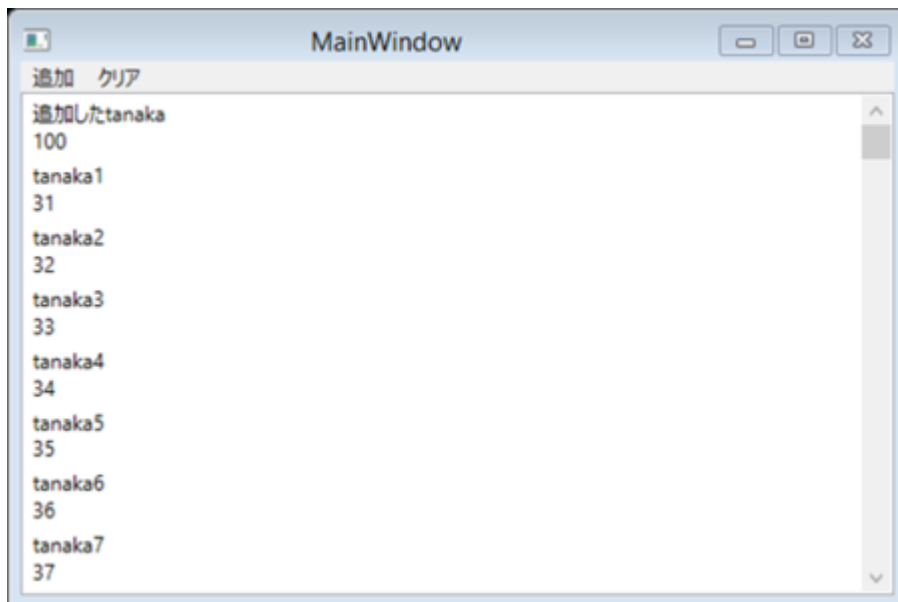
5.10.6.1. コレクションの変更通知

このサンプルプログラムは、ObservableCollection を使用しているので、コレクションに追加や削除などを行うと、画面の表示も更新されます。先程のサンプルプログラムに、Menu を追加して、そこに追加とクリアの MenuItem を追加します。そして、イベントハンドラに以下のようなコードを記述します。

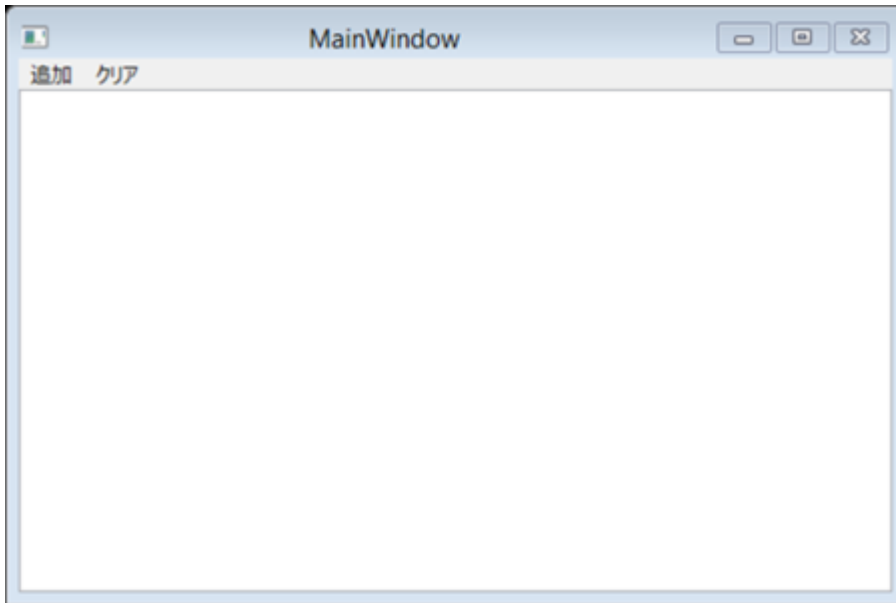
```
private void MenuItemAdd_Click(object sender, RoutedEventArgs e)
{
    // コレクションに要素を追加する。
    this.people.Insert(0, new Person { Name = "追加したtanaka", Age = 100 });
}

private void MenuItemClear_Click(object sender, RoutedEventArgs e)
{
    // 全削除
    this.people.Clear();
}
```

コレクションへの要素の追加と、コレクションから全要素の削除処理を行っています。サンプルプログラムを起動して追加メニューをクリックすると以下のようにコレクションに要素が追加され、それにあわせて表示も更新されます。



クリアメニューをクリックすると、people コレクションの中身がクリアされ、それにあわせて表示もクリアされます。



5.10.6.2. CollectionView

コレクションのデータバインディングを行うと、内部的には、直接コレクションがバインドされるのではなく、間に `CollectionView` というものが暗黙的に生成されます。これをデフォルトの `CollectionView` といいます。このデフォルトの `CollectionView` を取得するには `CollectionViewSource` クラスの `GetDefaultView` メソッドを呼んで取得します。

`CollectionView` は、コレクション本体を操作することなく、要素の並び替えやフィルター、グルーピングなどを行うことが出来るコレクションに対するビューの役割を担います。Filter 処理を行うには、`CollectionView` の `Filter` プロパティに `Predicate` 型のデリゲートを渡します。年齢が偶数の人のみを表示するコード例を以下に示します。

```
var collectionView = CollectionViewSource.GetDefaultView(this.people);
collectionView.Filter = x =>
{
    // 年齢が偶数の人だけにフィルタリング
    var person = (Person)x;
    return person.Age % 2 == 0;
};
```

ソート処理を行うには、`SortDescriptions` プロパティに `SortDescription` を追加することでソートができます。`SortDescription` は、ソートのキーになるプロパティ名と、昇順・降順を表す `ListSortDirection` 列挙体を渡します。`Age` プロパティをキーにして昇順にソートするコード例を以下に示します。


```
var collectionView = CollectionViewSource.GetDefaultView(this.people);
// Ageプロパティで昇順にソートする
collectionView.SortDescriptions.Add(new SortDescription("Age", ListSortDirection.Ascending));
```

グルーピングを行うには、GroupDescriptions プロパティに PropertyGroupDescription を追加することでグルーピングが出来ます。PropertyGroupDescription には、グルーピングするプロパティ名と、必要に応じてプロパティ値を変換するための IValueConverter を指定出来ます。Age プロパティの 10 の位の値をもとにグルーピングを行うコード例を以下に示します。

```
class AgeConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        return (int)value / 10;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}

// -----
var collectionView = CollectionViewSource.GetDefaultView(this.people);
// Ageプロパティの10の位の値でグルーピングする
collectionView.GroupDescriptions.Add(new PropertyGroupDescription("Age", new AgeConverter()));
```

グルーピングした結果を表示するには、ListBox コントロールなどの GroupStyle プロパティに GroupStyle を設定する必要があります。GroupStyle の HeaderTemplate に DataTemplate を指定することで、グループ単位のヘッダー要素を指定出来ます。コード例を以下に示します。

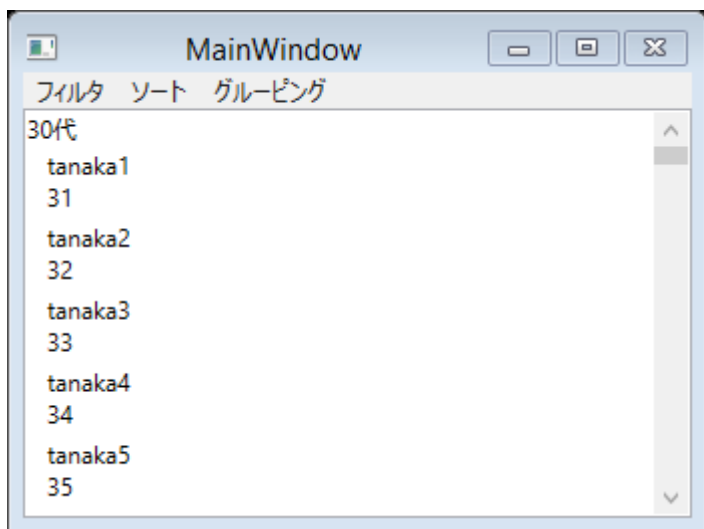
```

<ListBox.GroupStyle>
  <GroupStyle>
    <GroupStyle.HeaderTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding Name, StringFormat={}{0}0代}" />
      </DataTemplate>
    </GroupStyle.HeaderTemplate>
  </GroupStyle>
</ListBox.GroupStyle>

```

GroupStyle の HeaderTemplate では、Name プロパティでグルーピングするときにつかわれた値（今回の例では、10 代なら 1、20 代なら 2）が取得できます。そして、Binding の StringFormat プロパティを使ってデータバインディングの値の書式が指定できるため、“X0 代”と表示するようにしています。

実行してグルーピング処理を走らせた結果の画面を以下に示します。



5.10.6.3. CollectionViewSource

ここまで、デフォルトの CollectionView を使ってきましたが、CollectionView を明示的に作成することもできます。CollectionView を作成するには、CollectionViewSource クラスを使用します。CollectionViewSource クラスの Source プロパティにコレクションを設定すると、その型に応じて自動的に CollectionView が内部で生成されます。CollectionView の取得は CollectionViewSource クラスの View プロパティで行えます。

CollectionViewSource は、通常以下のように XAML の Resources に定義します。Resources に定義した CollectionViewSource は、Binding の Source に指定して使います。

```
<Window x:Class="CollectionBindingSample03.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">
    <Window.Resources>
        <CollectionViewSource
            x:Key="Source"
            Source="{Binding}" />
    </Window.Resources>
    <Grid>
        <DataGrid ItemsSource="{Binding Source={StaticResource Source}}" />
    </Grid>
</Window>
```

CollectionViewSource には、CollectionView と同様にソート、フィルタ、グルーピング機能があります。CollectionViewSource では、宣言的にこれらの機能を定義することが出来ます。

年齢の降順でソートして、年齢ごとにグルーピングする場合の定義は以下のようになります。

```
<CollectionViewSource
    x:Key="Source"
    Source="{Binding}" >
    <CollectionViewSource.GroupDescriptions>
        <PropertyGroupDescription PropertyName="Age"/>
    </CollectionViewSource.GroupDescriptions>
    <CollectionViewSource.SortDescriptions>
        <ComponentModel:SortDescription Direction="Descending" PropertyName="Age"/>
    </CollectionViewSource.SortDescriptions>
</CollectionViewSource>
```

フィルタは、この例では示していませんが、Filter イベントを定義することで実現できます。

5.10.6.4. リアルタイムソート・フィルター・グルーピング

CollectionView と CollectionViewSource を使ったフィルタ・ソート・グルーピングは、適用した直後のコレクションの状態を元にして処理が行われます。フィルタ・ソート・グルーピングを行った後に、コレクションが変更されたらグルーピングなどの状態は更新されません。コレクションの変更に追従して、フィルタ・ソート・グルーピングも更新されるようにするには、CollectionViewSource クラスで以下のプロパティに True を設定します。

プロパティ名

説明

IsLiveFilteringRequested	リアルタイムにフィルタリングをサポートしている CollectionView の場合はリアルタイムにフィルタリングを行う。
IsLiveSortingRequested	リアルタイムにソートをサポートしている CollectionView の場合はリアルタイムにソートを行う。
IsLiveGroupingRequested	リアルタイムにグルーピングをサポートしている CollectionView の場合はリアルタイムにグルーピングを行う。

上記のプロパティでリアルタイムの情報更新を有効にしたうえで、LiveFilteringProperties、LiveSortingProperties、LiveGroupingProperties で、フィルタ・ソート・グルーピングを有効にするプロパティ名を指定します。Age プロパティでリアルタイム機能を有効にする場合の XAML を以下に示します。

```
<CollectionViewSource
  x:Key="Source"
  Source="{Binding}"
  IsLiveFilteringRequested="True"
  IsLiveGroupingRequested="True"
  IsLiveSortingRequested="True" >
  <CollectionViewSource.LiveSortingProperties>
    <System:String>Age</System:String>
  </CollectionViewSource.LiveSortingProperties>
  <CollectionViewSource.LiveGroupingProperties>
    <System:String>Age</System:String>
  </CollectionViewSource.LiveGroupingProperties>
  <CollectionViewSource.LiveFilteringProperties>
    <System:String>Age</System:String>
  </CollectionViewSource.LiveFilteringProperties>
</CollectionViewSource>
```

これに、通常のソート機能・グルーピング機能などを追加します。今回の例では Age プロパティのみがリアルタイムに反映されます。

```
<CollectionViewSource.GroupDescriptions>
  <PropertyGroupDescription PropertyName="Age"/>
</CollectionViewSource.GroupDescriptions>
<CollectionViewSource.SortDescriptions>
  <ComponentModel.SortDescription Direction="Descending" PropertyName="Age"/>
</CollectionViewSource.SortDescriptions>
```

5.10.6.5. 現在の選択項目をバインディングする方法

CollectionView は、現在の選択項目を管理しているので、同じ CollectionView をバインドしているもの同士では、選択項目を同期することが可能です。選択項目を同期するには、Selector コントロール (ListBox などの親クラス) の `IsSynchronizedWithCurrentItem` を `True` にする必要があります。選択中の項目をバインドするには、`"/`”を使ってコレクションとバインドします。たとえば、Categories というコレクションにバインドしていて、その選択中の項目の `Name` プロパティをバインドするためには `“Categories/Name”` のようにバインディングの Path を指定します。この `"/`”を使ったバインディングは、親子だけでなく孫やその先までコレクションがある限り指定できます。例えば、People コレクションの選択中の項目の `Children` プロパティ (これもコレクション) の選択中の項目の `Name` プロパティは `“People/Children/Name”` のように指定できます。

例として以下のような Person クラスをバインドしたケースを示します。

```

using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace CollectionBindingSample04
{
    public class Person : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private void SetProperty<T>(ref T field, T value, [CallerMemberName] string propertyName
= null)
        {
            field = value;
            var h = this.PropertyChanged;
            if (h != null) { h(this, new PropertyChangedEventArgs(propertyName)); }
        }

        private string name;

        public string Name
        {
            get { return this.name; }
            set { this.SetProperty(ref this.name, value); }
        }

        private int age;

        public int Age
        {
            get { return this.age; }
            set { this.SetProperty(ref this.age, value); }
        }
    }
}

```

このクラスのコレクションを持った DataContext に格納するクラスを以下に示します。

```

using System.Collections.ObjectModel;

namespace CollectionBindingSample04
{
    public class MainWindowViewModel
    {
        private ObservableCollection<Person> people;

        public ObservableCollection<Person> People
        {
            get { return people; }
            set { people = value; }
        }
    }
}

```

MainWindow のコンストラクタで DataContext に設定します。

```

public MainWindow()
{
    InitializeComponent();
    this.DataContext = new MainWindowViewModel
    {
        People = new ObservableCollection<Person>(
            Enumerable.Range(1, 100)
                .Select(x => new Person
                {
                    Name = "tanaka" + x,
                    Age = (30 + x) % 50
                })
        )
    };
}

```

この MainWindowViewModel クラスの People を DataGrid にバインドします。バインド時に、IsSynchronizedWithCurrentItem を True に設定して、選択項目の同期を有効にします。

```

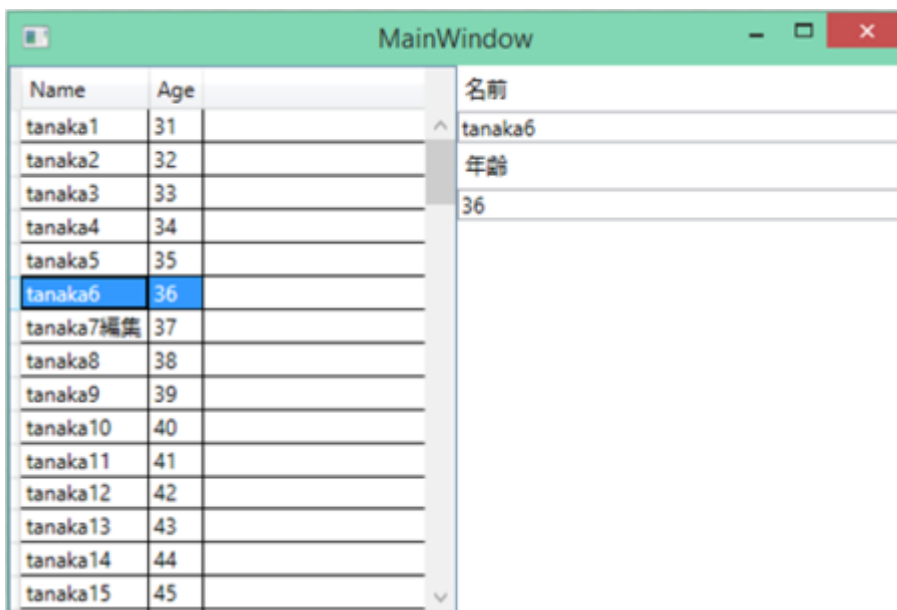
<!-- 選択項目を同期するように設定してバインドする -->
<DataGrid IsSynchronizedWithCurrentItem="True" ItemsSource="{Binding People}"/>

```

選択項目をバインドするために People/Name のように Path を指定してデータバインディングを行います。

```
<!-- 現在選択中の項目をバインドする -->
<Label Content="名前"/>
<TextBox TextWrapping="Wrap" Text="{Binding People/Name}"/>
<Label Content="年齢"/>
<TextBox TextWrapping="Wrap" Text="{Binding People/Age}"/>
```

実行すると選択項目が TextBox に表示されることが確認できます。



5.10.6.6. 別スレッドからのコレクションの操作

データバインディングしたコレクションは、通常 UI スレッドから操作する必要がありますが、WPF では、`BindingOperations.EnableCollectionSynchronization` メソッドを呼び出すことで、コレクションを UI スレッド以外から操作できるようになります。`EnableCollectionSynchronization` メソッドは、第一引数にコレクションを渡し、第二引数にコレクションを操作するときに使用するロックオブジェクトを指定します。使用例を以下に示します。


```

public partial class MainWindow : Window
{
    private Timer timer;
    private ObservableCollection<Person> people;

    public MainWindow()
    {
        InitializeComponent();
        // コレクションをDataContextに設定する
        this.people = new ObservableCollection<Person>();
        this.DataContext = this.people;

        // コレクションの操作をロックするように設定
        BindingOperations.EnableCollectionSynchronization(this.people, new object());

        // 別スレッドからコレクションを操作する
        this.timer = new Timer(1000);
        this.timer.Elapsed += (_, __) =>
            this.people.Add(new Person { Name = "tanaka " + this.people.Count });
        this.timer.Start();
    }
}

```

XAML を以下に示します。

```

<Window x:Class="CollectionBindingSample05.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>
        <DataGrid ItemsSource="{Binding}" />
    </Grid>
</Window>

```

BindingOperations.EnableCollectionSynchronization の行をコメントアウトすると、例外が出てアプリケーションが終了することが確認できます。バックグラウンドスレッドでコレクション操作をするときは、自分で UI スレッドで操作をするようにするか、ここで紹介した、BindingOperations.EnableCollectionSynchronization メソッドを使用しましょう。

5.11. コマンド

WPF には、ICommand インターフェースというユーザーの操作を抽象化する仕組みがあります。ICommand インターフェースは、以下のように定義されています。

```
public interface ICommand
{
    // コマンドを実行するかどうかに影響するような変更があった場合に発生します。
    event EventHandler CanExecuteChanged;

    // 現在の状態でこのコマンドを実行できるかどうかを判断するメソッドを定義します。
    //
    // パラメーター:
    //     parameter:
    //     コマンドで使用されたデータ。 コマンドにデータを渡す必要がない場合は、このオブジェクトを null に
    //     設定できます。
    //
    // 戻り値:
    //     このコマンドを実行できる場合は true。それ以外の場合は false。
    bool CanExecute(object parameter);

    // コマンドの起動時に呼び出されるメソッドを定義します。
    //
    // パラメーター:
    //     parameter:
    //     コマンドで使用されたデータ。 コマンドにデータを渡す必要がない場合は、このオブジェクトを null に
    //     設定できます。
    void Execute(object parameter);
}
```

コマンドが実行可能かどうかという状態に変化があったことを通知する CanExecuteChanged イベントと、実際にコマンドが実行可能かどうかを返す CanExecute メソッドがあります。そして、コマンドの処理を実行するための Execute メソッドがあります。

ICommand は、ICommandSource という以下のようなインターフェースを実装したクラスに対して設定することが出来ます。ICommandSource は以下のように定義されています。

```
// コマンドを呼び出す方法を認識しているオブジェクトを定義します。
public interface ICommandSource
{
    // コマンド ソースが呼び出されると実行されるコマンドを取得します。
    //
    // 戻り値:
    // コマンド ソースが呼び出されると実行されるコマンド。
    ICommand Command { get; }

    // コマンドの実行時にコマンドに渡すことのできるユーザー定義データの値を表します。
    //
    // 戻り値:
    // コマンド固有のデータ。
    object CommandParameter { get; }

    // コマンドが実行されているオブジェクト。
    //
    // 戻り値:
    // コマンドが実行されているオブジェクト。
   InputElement CommandTarget { get; }
}
```

実行するコマンドを取得するための Command プロパティと、コマンドに渡すための CommandParameter が定義されています。CommandTarget は後述する RoutedCommand のみに適用される特殊なプロパティなのでここでは省略します。ICommandSource インターフェースは、ButtonBase（ボタン系コントロールの基本クラス）や MenuItem など、ユーザーがアクションを実行するコントロールに主に実装されています。

WPF での ICommand インターフェースの実装クラスの RoutedCommand クラスは、CommandBinding という仕組みを通じてユーザーのアクションと処理を結びつける機能を持っています。RoutedCommand クラスは、以下のよう、クラスの静的メンバーとして定義して使用します。

```
public partial class MainWindow : Window
{
    public static RoutedCommand AlertCommand = new RoutedCommand();

    public MainWindow()
    {
        InitializeComponent();
    }
}
```

この RoutedCommand と、実際の処理を結びつけるには、UIElement クラスに定義されている CommandBindings プロパティに CommandBinding を設定して行います。一般的に Window クラスの CommandBindings プロパティを使って以下のように定義します。

```
<Window.CommandBindings>
  <CommandBinding
    Command="{x:Static local:MainWindow.AlertCommand}"
    Executed="CommandBinding_Executed"
    CanExecute="CommandBinding_CanExecute"/>
</Window.CommandBindings>
```

CommandBinding クラスの Command プロパティに、先程定義した Command のインスタンスを設定します。そして、Execute イベントに Command の実行時の処理のイベントハンドラと、CanExecute イベントに Command の実行可否の処理のイベントハンドラを設定します。ここでは、仮に CheckBox コントロールが画面にあり、CheckBox コントロールにチェックがされているときだけ実行可能なコマンドを定義します。画面に CheckBox コントロールと、コマンドを実行するための Button を置きます。Button の Command プロパティには、CommandBinding に設定したものと同じ、MainWindow クラスの AlertCommand プロパティを設定します。

```

<Window x:Class="CommandSample01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:CommandSample01"
        Title="MainWindow" Height="350" Width="525">
    <Window.CommandBindings>
        <CommandBinding
            Command="{x:Static local:MainWindow.AlertCommand}"
            Executed="CommandBinding_Executed"
            CanExecute="CommandBinding_CanExecute"/>
    </Window.CommandBindings>
    <StackPanel>
        <CheckBox x:Name="checkBox" Content="CanExecute"/>
        <Button Content="AlertCommand" Command="{x:Static local:MainWindow.AlertCommand}" />
    </StackPanel>
</Window>

```

そして、コードビハインドでイベントハンドラの処理を記述します。

```

public partial class MainWindow : Window
{
    public static RoutedCommand AlertCommand = new RoutedCommand();

    public MainWindow()
    {
        InitializeComponent();
    }

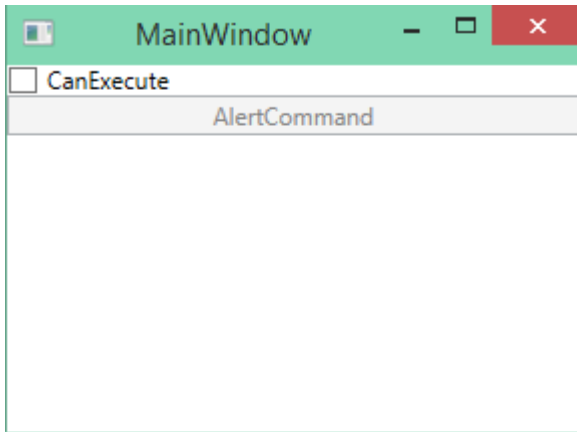
    private void CommandBinding_Executed(object sender, ExecutedRoutedEventArgs e)
    {
        MessageBox.Show("Hello world");
    }

    private void CommandBinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
    {
        e.CanExecute = this.checkBox.IsChecked.Value;
    }
}

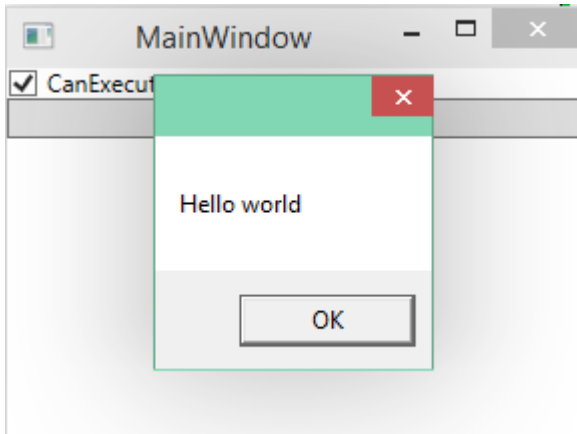
```

全体のつながりを説明すると、CommandBinding と Button の Command に同じコマンドのインスタンスを設定することで、この2つを繋げます。そして実際の処理は、CommandBinding のイベントハンドラで行います。実行結果を以下に示します。

実行すると、以下のようにボタンが押せない状態で起動します。



CheckBox コントロールにチェックを入れると、ボタンが押せるようになります。これは CommandBinding に設定した CanExecute イベントのイベントハンドラで行っている処理で CheckBox コントロールのチェック状態を見てイベント引数の CanExecute プロパティに実行可否の値を設定しているためです。



このように、RoutedCommand クラスと、CommandBinding クラスを使うことで操作を表すコマンドと実際の処理を分離して記述することが出来ます。

コマンドは、InputBinding を使うことで簡単にキーボードショートカットやマウスジェスチャーに対応させることが出来ます。Window などの InputBindings プロパティに、KeyBinding を設定することでキーボードショートカットとコマンドの関連付けを行うことが出来ます。KeyBinding の Modifiers プロパティに修飾キーを設定して、Key プロパティにキーを設定して、Command プロパティに該当するキーボードが押されたときに実行する処理を表すコ

マンドを設定します。例として、先程の `AlertCommand` を `Ctrl+Alt+A` を押したときに表示するようにするコードを示します。

```
<Window.InputBindings>
  <KeyBinding
    Modifiers="Alt+Control"
    Key="A"
    Command="{x:Static local:MainWindow.AlertCommand}" />
</Window.InputBindings>
```

実行して、`CheckBox` コントロールにチェックを入れた状態で `Ctrl+Alt+A` を押すと `MessageBox` が表示されます。

これまでの例では、自分で `RoutedCommand` のインスタンスを用意したものを使用しましたが、WPF では組み込みで、アプリケーションによくあるコマンドがあらかじめ定義されています。コピーやペーストなどの一般的な操作を `RoutedCommand` で定義する場合は、下記の `ApplicationCommands` クラスに定義されているものを使用するとよいでしょう。

- `ApplicationCommands` クラス

[http://msdn.microsoft.com/ja-jp/library/system.windows.input.applicationcommands\(v=vs.110\).aspx](http://msdn.microsoft.com/ja-jp/library/system.windows.input.applicationcommands(v=vs.110).aspx)

このように、WPF では、組み込みの `ICommand` インターフェースの実装が提供されています。しかし、`ICommand` インターフェースを実装していれば、`InputBinding` などの機能は使うことが出来ます。最近の WPF をはじめとする XAML を使った開発では `ICommand` インターフェースを実装して `Execute` や `CanExecute` の処理をデリゲートで受け取る `DelegateCommand` (`RelayCommand` という名前の場合もある) という実装が使われるのが一般的です。これらのコマンドの独自実装については後述します。

6. 応用

ここでは、これまで説明していなかった応用的な話について説明します。

6.1. Behavior

WPF の標準部品ではないですが、Blend SDK for WPF に同梱されている `Behavior` という部品があります。Blend は Visual Studio に同梱されているため、基本的に標準でついていていいライブラリです。`Behavior` は、`Trigger` と `Action` (WPF の同名の機能とは別物になります) と、`Behavior` を提供しています。`Trigger` は、名前の

とおり、何か処理のきっかけがあったことを示すための部品です。Action は、Trigger に設定して Trigger の条件が満たされたタイミングで実行される処理になります。

6.1.1. Trigger と Action

Trigger は、WPF 標準の Trigger と比較して独自の Trigger が作れるようになっているなど柔軟に作られています。代表的な Trigger を以下に示します。

- **EventTrigger**
指定したイベントが発生したときに Action を呼び出します。
- **TimerTrigger**
指定したタイミングで Action を呼び出します。
- **PropertyChangedEventTrigger**
指定したプロパティが変化したときに Action を呼び出します。

Trigger が特定の条件を満たしたときに呼び出される Action の中で代表的なものを以下に示します。

- **CallMethodAction**
指定したメソッドを呼び出します。
- **ChangePropertyAction**
プロパティの値を指定した値にしたり、インクリメントしたりすることができます。
- **ControlStoryboardAction**
ストーリーボードを再生、停止などの操作をすることができます。
- **GoToStateAction**
VisualState を変更することができます。
- **InvokeCommandAction**
指定した Command を呼び出します。
- **LaunchUriOrFileAction**
指定した Uri かファイルを開きます。

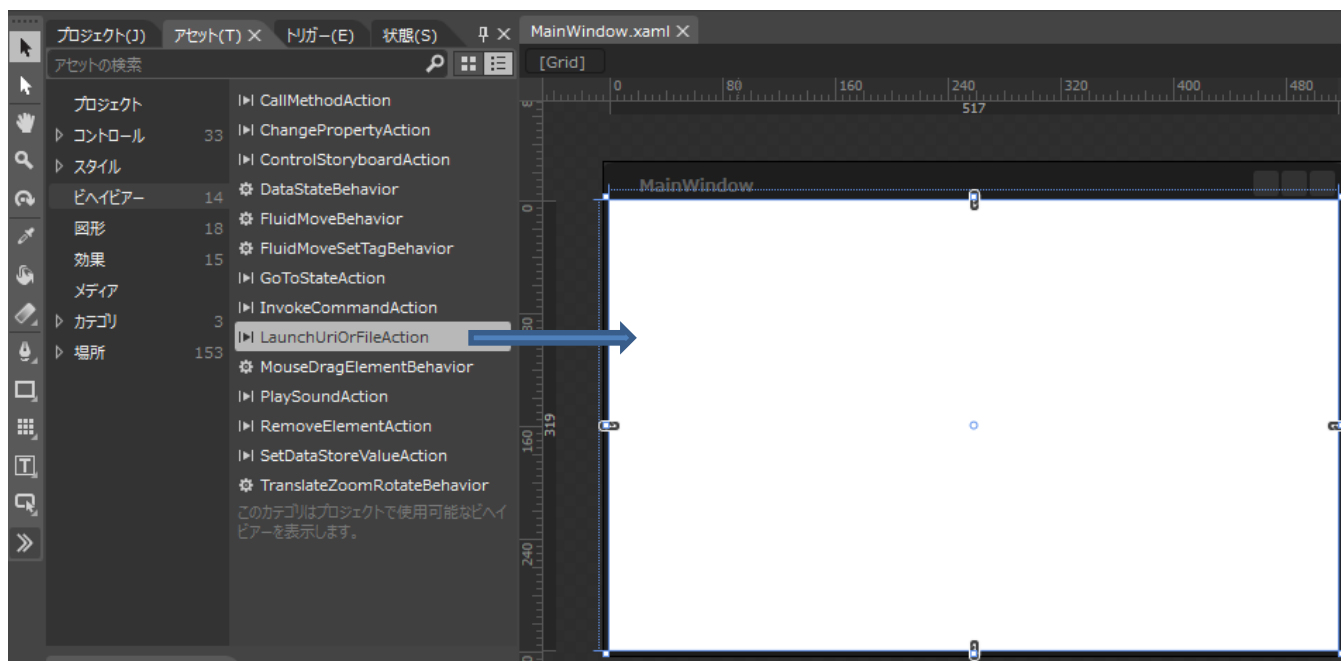
6.1.2. Trigger と Action の使い方

Trigger と Action を使うには、Blend for Visual Studio で、使用したい Action を、対象のコントロールにドロップします。その後、プロパティウィンドウの TriggerType（デフォルトで EventTrigger が設定されています）の横の新規作成ボタンで Trigger の種類を指定します。

例として、5 秒周期で指定した URL を開く処理を Trigger と Action で作成します。この例で使用する Trigger と Action は、以下のものになります。

- TimerTrigger
- LaunchUriOrFileAction

LaunchUriOrFileAction を、Window の Grid 上にドロップします。



そうすると、以下のような XAML が生成されます。デフォルトの Trigger として、EventTrigger が指定されていることが確認できます。

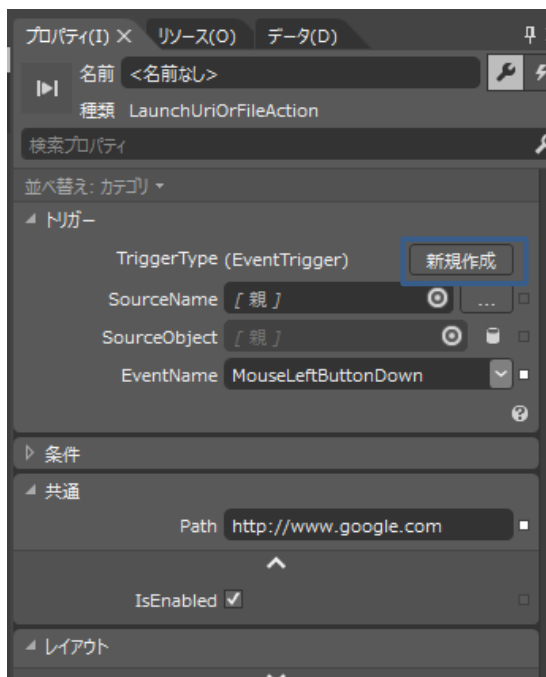
```

<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
  xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
  x:Class="BehaviorSample01.MainWindow"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <i:Interaction.Triggers>
      <i:EventTrigger EventName="MouseLeftButtonDown">
        <ei:LaunchUriOrFileAction/>
      </i:EventTrigger>
    </i:Interaction.Triggers>
  </Grid>
</Window>

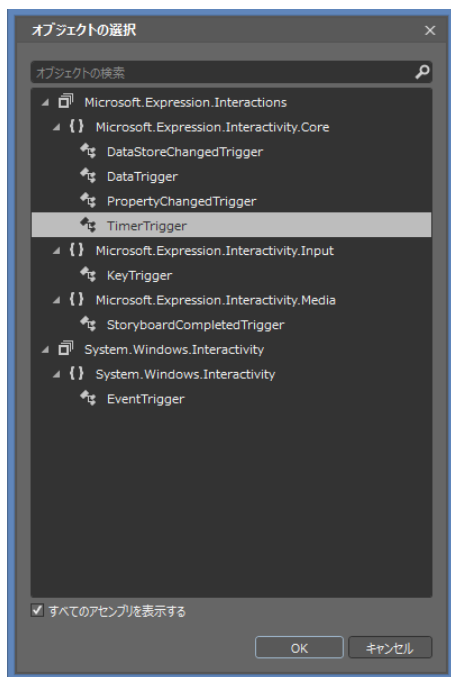
```

オブジェクトとタイムラインから、LaunchUriOrFileAction を選択して Path プロパティに

<http://www.google.com> などの URL を入力します。次に、Trigger を TimerTrigger に変更します。Trigger の変更は、プロパティウィンドウのトリガーにある TriggerType の新規作成を選択します。



そうすると、Trigger を選択画面が表示されます。ここで、TimerTrigger を選択します。



OK を選択すると Trigger が EventTrigger から TimerTrigger に変更されます。

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
    xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
    x:Class="BehaviorSample01.MainWindow"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <i:Interaction.Triggers>
            <ei:TimerTrigger>
                <ei:LaunchUriOrFileAction Path="http://www.google.com"/>
            </ei:TimerTrigger>
        </i:Interaction.Triggers>
    </Grid>
</Window>
```

TimerTrigger は、MillisecondsPerTick プロパティで Trigger が Action を実行する間隔を指定できます。ここでは 5000 を設定します。この状態で実行すると、5 秒間隔で Google のサイトが開かれます。

6.1.3. Behavior

Trigger と Action は、処理のきっかけと実際の処理が分離されていましたが、処理のきっかけと処理が分離されていないものが Behavior として提供されています。代表的なものとして以下のような Behavior が提供されています。

- **DataStateBehavior**
Binding プロパティと Value プロパティの値が等しいときに TrueState に遷移して等しくないときに FalseState に遷移します。
- **FluidMoveBehavior**
Panel 内の要素の移動をアニメーションさせることができます。
- **MouseDragElementBehavior**
タッチやマウス操作で、Behavior を設定したコントロールを移動させることができますようになります。
- **TranslateZoomRotateBehavior**
タッチやマウス操作で、Behavior を設定したコントロールを回転、移動させることができますようになります。

例として、DataStateBehavior を使用して、CheckBox のチェックの有無で VisualState を切り替える XAML を以下に示します。

```

<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
  xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
  xmlns:System="clr-namespace:System;assembly=mscorlib"
  x:Class="BehaviorSample02.MainWindow"
  Title="MainWindow" Height="350" Width="525">
  <Grid x:Name="grid">
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup x:Name="VisualStateGroup">
        <VisualState x:Name="TrueState">
          <Storyboard>
            <ObjectAnimationUsingKeyFrames
              Storyboard.TargetProperty="(Panel.Background)"
              Storyboard.TargetName="grid">
              <DiscreteObjectKeyFrame KeyTime="0">
                <DiscreteObjectKeyFrame.Value>
                  <SolidColorBrush Color="#FFFF8686"/>
                </DiscreteObjectKeyFrame.Value>
              </DiscreteObjectKeyFrame>
            </ObjectAnimationUsingKeyFrames>
          </Storyboard>
        </VisualState>
        <VisualState x:Name="FalseState"/>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <i:Interaction.Behaviors>
      <ei:DataStateBehavior
        TrueState="TrueState"
        FalseState="FalseState"
        Binding="{Binding IsChecked, ElementName=checkBox}"
        Value="True"/>
    </i:Interaction.Behaviors>
    <CheckBox x:Name="checkBox" Content="CheckBox" />
  </Grid>
</Window>

```

TrueState と FalseState という VisualState を定義して、DataStateBehavior から参照しています。

DataStateBehavior の Binding を CheckBox の IsChecked と非もづけて、Value プロパティに True を設定します。こうすることで、CheckBox にチェックが入った時に TrueState が、チェックが入っていないときに

FalseState に遷移するようになります。実行すると、CheckBox にチェックをつけたときに背景が薄い赤色になります。

このように、Behavior を使うと今までコードビハインドを使って書いていたような処理が部品化されているため Blend for Visual Studio 上でドラッグアンドドロップしてプロパティを設定するだけで作成できるようになります。

6.1.4. Behavior や Trigger や Action の作成

Behavior や Trigger と Action を使用することで、簡単なロジックが RAD 環境で構築できることがわかりました。ここでは、ありものの Behavior を使うのではなく自作の Behavior や Trigger/Action を作成する方法について示します。Behavior は、コードビハインドに何回も同じ処理を書いていた場合、作成するとよいことが多いです。では作成してみたいと思います。

6.1.4.1. Behavior の自作

ここで作成するのは、ボタンをクリックすると Hello world と表示する Behavior を自作したいと思います。

Behavior を作成するために以下のアセンブリを参照に追加します。（Behavior を使うときには Blend が自動で追加していたものになります）

- System.Windows.Interactivity
- Microsoft.Expression.Interactions

バージョンは 4.5 のものを追加してください。

Behavior の作成には、Behavior<T>クラスを継承します。Behavior<T>クラスの型引数には、Behavior を置くことができる型を指定します。特に指定がなければ DependencyObjectなどを指定すればよいです。今回は Button に置くので、Button を指定します。また TypeConstraint 属性でも Behavior を置ける型を指定します。

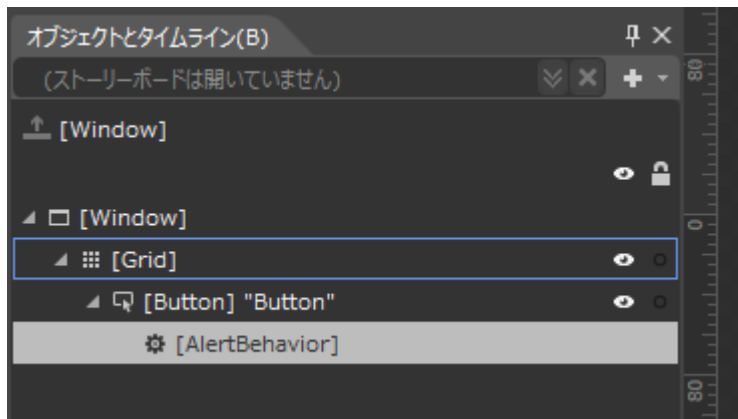
```
using System;
using System.Windows.Controls;
using System.Windows.Interactivity;

namespace BehaviorSample03
{
    [TypeConstraint(typeof(Button))]
    public class AlertBehavior : Behavior<Button>
    {
    }
}
```

Behavior は、対象に設定されたときに呼び出される OnAttached メソッドと、対象から外されるときに呼び出される OnDetaching メソッドをオーバーライドして処理を作成していきます。今回は、OnAttached メソッドでクリックイベントを購読して、OnDetaching メソッドでクリックイベントの購読を解除します。クリックイベントのイベントハンドラでは、MessageBox を表示しています。

```
[TypeConstraint(typeof(Button))]  
public class AlertBehavior : Behavior<Button>  
{  
    protected override void OnAttached()  
    {  
        // AssociatedObjectのイベントを購読する  
        this.AssociatedObject.Click += this.ButtonClicked;  
    }  
  
    protected override void OnDetaching()  
    {  
        // イベントの購読解除  
        this.AssociatedObject.Click -= this.ButtonClicked;  
    }  
  
    // イベントで処理をする  
    private void ButtonClicked(object sender, System.Windows.RoutedEventArgs e)  
    {  
        MessageBox.Show("Hello world");  
    }  
}
```

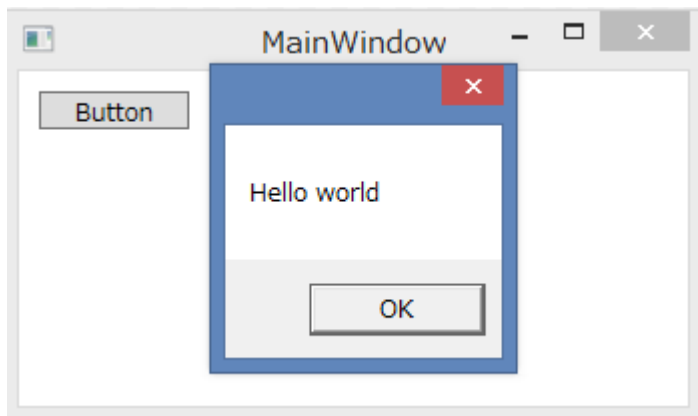
ビルドすると、Blend で AlertBehavior が使えるようになります。Button を画面に置いて、その上に AlertBehavior をドロップします。



XAML を以下に示します。

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
  xmlns:local="clr-namespace:BehaviorSample03"
  x:Class="BehaviorSample03.MainWindow"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <Button Content="Button" HorizontalAlignment="Left"
      Margin="10,10,0,0" VerticalAlignment="Top" Width="75">
      <i:Interaction.Behaviors>
        <local:AlertBehavior/>
      </i:Interaction.Behaviors>
    </Button>
  </Grid>
</Window>
```

実行してボタンをクリックすると、MessageBox が表示されます。



6.1.4.2. *Trigger と Action の自作*

Trigger を作成するには、System.Windows.Interactivity 名前空間の `TriggerBase<T>` クラスを継承します。Behavior と同様に `OnAttached` メソッドと `OnDetaching` メソッドがあるので、そこでイベントを購読するなどの Trigger が Action を実行するためのセットアップをします。`TriggerBase<T>` クラスの `InvokeActions` メソッドを呼び出すことで、Trigger に設定された Action を呼び出すことができます。その際に、引数を渡すことで Action に、情報を渡すこともできます。

ここでは、先ほどの `AlertBehavior` を、Trigger と Action に分解してみようと思います。Trigger のコードを以下に示します。

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

namespace BehaviorSample04
{
    [TypeConstraint(typeof(Button))]
    public class ButtonClickTrigger : TriggerBase<Button>
    {
        protected override void OnAttached()
        {
            this.AssociatedObject.Click += this.ButtonClick;
        }

        private void ButtonClick(object sender, RoutedEventArgs e)
        {
            this.InvokeActions(e);
        }

        protected override void OnDetaching()
        {
            this.AssociatedObject.Click -= this.ButtonClick;
        }
    }
}
```

Action は、TriggerAction<T>クラスを継承して、Invoke メソッドをオーバーライドしてそこに処理を記述します。今回のサンプルでは、MessageBox を表示しています。Behavior と同様に TypeConstraint 属性で Action を置く型の指定や、DefaultTrigger 属性で、デフォルトで設定する Trigger の型を指定できます。（指定しない場合は EventTrigger が使われます）今回の例では、Button に置いたときに ButtonClickTrigger クラスを使用するように設定しています。コードを以下に示します。

```

using System.Windows;
using System.Windows.Controls;
using System.Windows.Interactivity;

namespace BehaviorSample04
{
    [TypeConstraint(typeof(Button))]
    [DefaultTrigger(typeof(Button), typeof(ButtonClickTrigger))]
    public class AlertAction : TriggerAction<Button>
    {
        protected override void Invoke(object parameter)
        {
            MessageBox.Show("Hello world");
        }
    }
}

```

画面に Button を置いて AlertAction をドロップすると、以下のような XAML が生成されます。

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
    xmlns:local="clr-namespace:BehaviorSample04"
    x:Class="BehaviorSample04.MainWindow"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Button" HorizontalAlignment="Left"
            Margin="10,10,0,0" VerticalAlignment="Top" Width="75">
            <i:Interaction.Triggers>
                <local:ButtonClickTrigger>
                    <local:AlertAction/>
                </local:ButtonClickTrigger>
            </i:Interaction.Triggers>
        </Button>
    </Grid>
</Window>

```

実行すると、Behavior の時と同様に Button をクリックすると MessageBox が表示されます。

6.2. データバインディングを前提としたプログラミングモデル

WPF では、強力なデータバインディングを活かした設計パターンとして Model View ViewModel パターンというアプリケーションを設計するうえでの定石となる設計パターンがあります。Model View ViewModel パターンは MVVM パターンと略されます。MVVM パターンは、WPF だけでなく Web アプリ開発や、その他のアプリ開発にも波及していて、それぞれの状況に応じて形を変えて存在しています。

MVVM パターンは、MSDN マガジンの以下の記事をきっかけに世間に認知されるようになりました。

- Model-View-ViewModel デザイン パターンによる WPF アプリケーション

<http://msdn.microsoft.com/ja-jp/magazine/dd419663.aspx>

また、Microsoft はオープンソースで Prism という MVVM パターンをサポートするライブラリを提供しています。

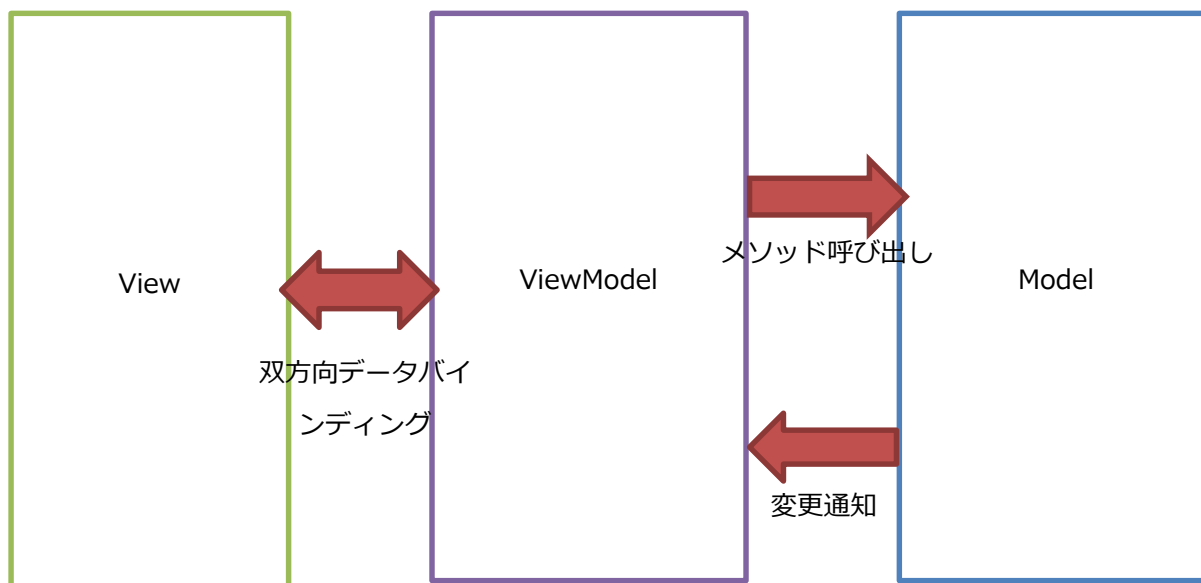
- Prism

<http://compositewpf.codeplex.com/>

ここでは、簡単に MVVM パターンの考えについて説明したあと、Prism の一部の機能を使って実際に MVVM パターンのサンプルプログラムを作成していきたいと思います。

6.2.1. MVVM パターンとは

MVVM パターンは、View (XAML + コードビハインド) と ViewModel と呼ばれる Model を View に適したインターフェースに変換するレイヤと、アプリケーションを記述する Model のレイヤからなります。View と ViewModel 間は、基本的にデータバインディングによって連携を行います。ViewModel は Model の変更を監視したり、必要に応じて Model のメソッドの呼び出しを行います。この関係を図で表すと以下のようになります。



6.2.2. 変更通知の仕組み

MVVM パターンの、変更通知や双方向データバインディングの ViewModel から View 方向の変更通知には `INotifyPropertyChanged` インターフェースを実装したクラスを使用します。`INotifyPropertyChanged` インターフェースは `PropertyChanged` イベントのみをもつシンプルなインターフェースです。このイベントを通じて Model から ViewModel、ViewModel から View への変更通知が行われます。

`INotifyPropertyChanged` インターフェースの実装をすべてのプロパティに実装するのは負荷が高いため、一般的に以下のようなヘルパークラスが作成されます。

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace MVVMSample01
{
    public class BindableBase : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual bool SetProperty<T>(ref T field, T value,
            [CallerMemberName]string propertyName = null)
        {
            if (Equals(field, value)) { return false; }
            field = value;
            var h = this.PropertyChanged;
            if (h != null) { h(this, new PropertyChangedEventArgs(propertyName)); }
            return true;
        }
    }
}
```

このクラスを継承すると、プロパティの変更通知機能を持ったクラスが以下のように簡単に作成できます。

```
namespace MVVMSample01
{
    public class Person : BindableBase
    {
        private int age;

        public int Age
        {
            get { return this.age; }
            set { this.SetProperty(ref this.age, value); }
        }

        private string name;

        public string Name
        {
            get { return this.name; }
            set { this.SetProperty(ref this.name, value); }
        }
    }
}
```

単一のクラスの変更通知は INotifyPropertyChanged インターフェースで行いますが、コレクションの変更通知には、これまでも使ってきた ObservableCollection<T>クラスを使用します。基本的に、この2通りの変更通知を通して Model から ViewModel と ViewModel から View の間のやり取りを行います。

6.2.3. ユーザーからの入力の処理

ボタンをクリックするなどのユーザーの処理を View から ViewModel に伝えるには、Command を使用します。この時使用する Command は RoutedCommand ではなく、デリゲートに Execute と CanExecute 処理を移譲する実装の DelegateCommand (RelayCommand という名前で作られることも多いです) クラスを使用します。

DelegateCommand を ViewModel クラスのプロパティとして定義して、View の Button や MenuItem などの Command プロパティとバインドして使用します。

6.2.4. 最初のアプリケーションの作成

各レイヤの連携方法がわかったので、簡単なアプリケーションを作成します。このアプリケーションは、入力した文字列を、ボタンを押したタイミングで大文字に変換して出力するものです。ボタンは、入力が空の場合は押すことが

できません。また、このサンプルプログラムは、処理が単純すぎるため Model に該当する部分は存在しません。あくまで View と ViewModel が連携した場合の動きを示すものです。

WPF アプリケーションを作成して NuGet で Prism.Mvvm のパッケージを追加します。Prism.Mvvm は BindableBase クラスや DelegateCommand クラスなどの MVVM パターンに必須のクラスだけを持ったシンプルなライブラリです。

ライブラリを追加したら、ViewModel を作成します。MainWindowViewModel という名前でクラスを作って以下のようなコードを作成します。入力用のプロパティと出力用のプロパティと変換用のコマンドを定義しています。コマンドの実行可否は、入力値が変化するたびに評価が必要なので DelegateCommand の CanExecute を再評価するためのメソッドを呼び出しています。

クラスを定義します。

```
using Microsoft.Practices.Prism.Commands;
using Microsoft.Practices.Prism.Mvvm;

namespace MVVMSample01
{
    public class MainWindowViewModel : BindableBase
    {
    }
}
```

入力、出力を受け取るプロパティを定義します。


```
private string input;
/// <summary>
/// 入力値
/// </summary>
public string Input
{
    get { return this.input; }
    set
    {
        this.SetProperty(ref this.input, value);
        // 入力値に変かがある度にコマンドのCanExecuteの状態が変わったことを通知する
        this.ConvertCommand.RaiseCanExecuteChanged();
    }
}

private string output;

/// <summary>
/// 出力値
/// </summary>
public string Output
{
    get { return this.output; }
    set { this.SetProperty(ref this.output, value); }
}
```

そして、Command を定義します。

```

/// <summary>
/// 変換コマンド
/// </summary>
public DelegateCommand ConvertCommand { get; private set; }

public MainWindowViewModel()
{
    // 変換コマンドに実際の処理をわたして初期化
    this.ConvertCommand = new DelegateCommand(
        this.ConvertExecute,
        this.CanConvertExecute);
}

/// <summary>
/// 大文字に変換
/// </summary>
private void ConvertExecute()
{
    this.Output = this.Input.ToUpper();
}

/// <summary>
/// 何か入力されてたら実行可能
/// </summary>
/// <returns></returns>
private bool CanConvertExecute()
{
    return !string.IsNullOrEmpty(this.Input);
}

```

ビルドして View (XAML) を作成します。ViewModel を XAML で参照できるように名前空間の定義を行います。

```
xmlns:l="clr-namespace:MVVMSample01"
```

そして、DataContext プロパティに先ほど作成した ViewModel クラスを設定します。

```

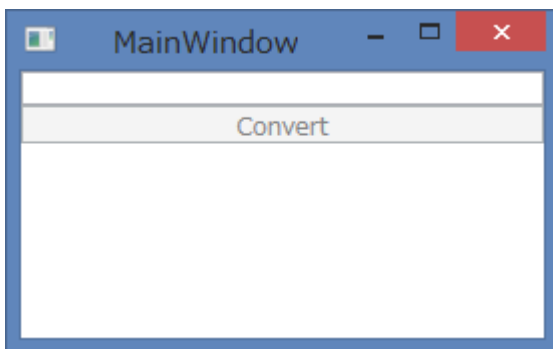
<Window.DataContext>
    <l:MainWindowViewModel />
</Window.DataContext>

```

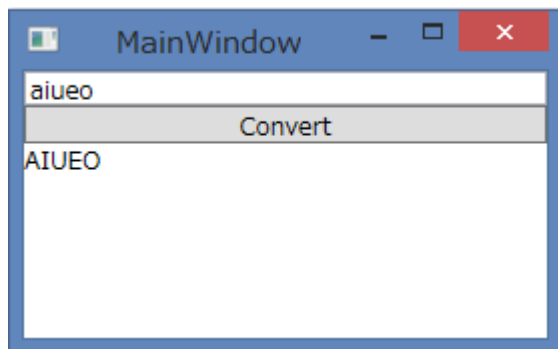
画面を作成していきます。入力用の TextBox と出力用の TextBlock とコマンドを実行するための Button を置いて、ViewModel の対応するプロパティとバインディングしています。

```
<Window
  x:Class="MVVMSample01.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:l="clr-namespace:MVVMSample01"
  Title="MainWindow" Height="350" Width="525">
  <Window.DataContext>
    <l:MainWindowViewModel />
  </Window.DataContext>
  <StackPanel>
    <TextBox Text="{Binding Input, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
    <Button Content="Convert" Command="{Binding ConvertCommand}" />
    <TextBlock Text="{Binding Output}" />
  </StackPanel>
</Window>
```

実行すると、以下のような画面が表示されます。



TextBox に文字を入力すると Convert ボタンが押せるようになり、押すと Convert ボタンの下に大文字に変換された結果が表示されます。



6.2.5. 四則演算アプリケーション

先ほどのアプリケーションはシンプルすぎて Model がありませんでしたが、今度はシンプルな四則演算アプリケーションで Model まで含んだコード例を示したいと思います。MVVM の基本クラスは、Prism.Mvvm のクラスを使用します。

6.2.5.1. Model の作成

Model はなるべくプレーンな C# のクラスになるように心がけます。そして状態の変更を `INotifyPropertyChanged` を通じて外部に通知します。今回は、左辺値、右辺値、計算方法、計算結果をステートとして持たせます。`INotifyPropertyChanged` を実装したクラスなので、Prism の `BindableBase` クラスを基本クラスとして使用します。

左辺値、右辺値、計算結果は `double` 型で保持して、計算結果は `OperationType` という `enum` 型を定義してそれを使用しています。コードを以下に示します。

```
public class Calc : BindableBase
{
    private double lhs;

    public double Lhs
    {
        get { return this.lhs; }
        set { this.SetProperty(ref this.lhs, value); }
    }

    private double rhs;

    public double Rhs
    {
        get { return this.rhs; }
        set { this.SetProperty(ref this.rhs, value); }
    }

    private OperatorType operatorType;

    public OperatorType OperatorType
    {
        get { return this.operatorType; }
        set { this.SetProperty(ref this.operatorType, value); }
    }

    private double answer;

    public double Answer
    {
        get { return this.answer; }
        set { this.SetProperty(ref this.answer, value); }
    }
}

public enum OperatorType
{
    Add,
    Sub,
    Mul,
    Div,
}
```

次に、アプリケーション全体を示すクラスを作成します。趣味の問題ですが、私は、このクラスから各 Model のクラスへたどれるように作っています。クラス名は AppContext という名前で作成しました。このクラスでは、アプリケーション全体でグローバルに持たせる状態を定義しています。今回は、アプリケーションのメッセージを表示させるようにしています。そして、先ほど定義した Calc クラスもプロパティとして定義します。Calc クラスから必要に応じてメッセージが設定できるように Calc クラスに自分自身を渡しています。

```
public class AppContext : BindableBase
{
    private string message;

    public string Message
    {
        get { return this.message; }
        set { this.SetProperty(ref this.message, value); }
    }

    public Calc Calc { get; private set; }

    public AppContext()
    {
        this.Calc = new Calc(this);
    }
}
```

Calc クラス側には、AppContext クラスを受け取るコンストラクタとフィールドを定義します。

```
private AppContext appContext;

public Calc(AppContext appContext)
{
    this.appContext = appContext;
}
```

そして、Calc クラスに計算ロジックを記述します。計算ロジックは 0 除算のときにエラーメッセージを出す以外は直に計算するだけにしました。

```
public void Execute()
{
    switch (this.OperatorType)
    {
        case OperatorType.Add:
            this.Answer = this.Lhs + this.Rhs;
            break;
        case OperatorType.Sub:
            this.Answer = this.Lhs - this.Rhs;
            break;
        case OperatorType.Mul:
            this.Answer = this.Lhs * this.Rhs;
            break;
        case OperatorType.Div:
            if (this.Rhs == 0)
            {
                this.appContext.Message = "0除算エラー";
                return;
            }
            this.Answer = this.Lhs / this.Rhs;
            break;
        default:
            throw new InvalidOperationException();
    }
}
```

6.2.5.2. ViewModel の作成

Model が完成したので ViewModel を作成します。ViewModel では、まず計算方法の OperationType を文字列と非もづけるための OperationTypeViewModel クラスを作成します。

```

public class OperatorTypeViewModel
{
    public OperatorType OperatorType { get; private set; }
    public string Label { get; private set; }

    public OperatorTypeViewModel(string label, OperatorType operatorType)
    {
        this.Label = label;
        this.OperatorType = operatorType;
    }

    public static OperatorTypeViewModel[] OperatorTypes = new[]
    {
        new OperatorTypeViewModel("足し算", OperatorType.Add),
        new OperatorTypeViewModel("引き算", OperatorType.Sub),
        new OperatorTypeViewModel("掛け算", OperatorType.Mul),
        new OperatorTypeViewModel("割り算", OperatorType.Div),
    };
}

```

計算方法の ViewModel ができたので、次は、MainWindow 用の ViewModel を作成します。クラス名は MainWindowViewModel にしました。MainWindowViewModel クラスには、左辺値、右辺値を受け取る string 型のプロパティを定義します。ここに入力値を受け取って、計算処理のときに double 型に変換して Model の左辺値と右辺値に設定します。そして、計算結果を格納するための Answer プロパティを定義します。これは、Model から正しい値が来ることが期待できるので、素直に double 型として定義します。

左辺値と右辺値は、あとで定義する計算をするための DelegateCommand 型の ExecuteCommand プロパティに対して呼び出し可能かどうかを変更されたというイベントを発生させるために RaiseCanExecuteChanged メソッドを呼び出しています。

最後に、画面に表示するメッセージを表示するプロパティも定義しています。


```
public class MainWindowViewModel : BindableBase
{
    private string lhs;

    public string Lhs
    {
        get { return this.lhs; }
        set
        {
            this.SetProperty(ref this.lhs, value);
            this.ExecuteCommand.RaiseCanExecuteChanged();
        }
    }

    private string rhs;

    public string Rhs
    {
        get { return this.rhs; }
        set
        {
            this.SetProperty(ref this.rhs, value);
            this.ExecuteCommand.RaiseCanExecuteChanged();
        }
    }

    private double answer;

    public double Answer
    {
        get { return this.answer; }
        set { this.SetProperty(ref this.answer, value); }
    }

    private string message;

    public string Message
    {
        get { return this.message; }
        set { this.SetProperty(ref this.message, value); }
    }
}
```

次に、計算方法のプロパティを定義します。これは先ほど作成した `OperationTypeViewModel` 型の配列と、実際に選択された `OperationTypeViewModel` 型のインスタンスを格納するプロパティを定義します。計算方式のプロパティは、コンストラクタで初期化を行います。

また、現在選択された `OperationTypeViewModel` 型を現すプロパティでは、変更されたときに、`ExecuteCommand` プロパティの `RaiseCanExecuteChanged` メソッドを呼び出して、コマンドが実行可能かどうかに変化があったことを伝えています。

```
public OperatorTypeViewModel[] OperatorTypes { get; private set; }

private OperatorTypeViewModel selectedOperatorType;

public OperatorTypeViewModel SelectedOperatorType
{
    get { return this.selectedOperatorType; }
    set
    {
        this.SetProperty(ref this.selectedOperatorType, value);
        this.ExecuteCommand.RaiseCanExecuteChanged();
    }
}

public MainWindowViewModel()
{
    this.OperatorTypes = OperatorTypeViewModel.OperatorTypes;
}
```

次に、`Model` を `ViewModel` と接続します。今回は 1 画面のアプリなので、`MainWindowViewModel` 内に `Model` のルートである `AppContext` クラスのインスタンスを持たせる形にしました。複数画面のアプリなどで複数の `ViewModel` から `AppContext` を参照するようなケースでは `AppContext` クラスのインスタンスをもう少しグローバルにアクセス可能な形で定義するとよいと思います。（例として `App` クラスとか）`AppContext` クラスをフィールドとして定義したら、コンストラクタで `PropertyChanged` を監視して、必要に応じて `Model` の変更を `ViewModel` に反映するコードを書きます。ここでは、`Model` のメッセージと計算結果を監視するコードを追加します。

```

private AppContext appContext = new AppContext();

public MainWindowViewModel()
{
    this.OperatorTypes = OperatorTypeViewModel.OperatorTypes;

    // Modelの監視
    this.appContext.PropertyChanged += this.AppContextPropertyChanged;
    this.appContext.Calc.PropertyChanged += this.CalcPropertyChanged;
}

private void CalcPropertyChanged(object sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName == "Answer")
    {
        this.Answer = this.appContext.Calc.Answer;
    }
}

private void AppContextPropertyChanged(object sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName == "Message")
    {
        this.Message = this.appContext.Message;
    }
}

```

今回のような Model と ViewModel が 1 対 1 の関係にあるアプリでは問題になりませんが、Model と ViewModel が 1 対 N の関係にあるようなケースの場合には、Model の PropertyChanged イベントの購読を Window がとしたタイミングなどで解除する必要がある点に注意してください。そうしないと、ViewModel のインスタンスがいつまでだっても GC の回収対象にならないという問題があります。

最後に計算を行う Command を定義します。ExecuteCommand という名前で DelegateCommand 型のプロパティを定義してコンストラクタで初期化します。DelegateCommand の Execute の処理では、Model の状態を ViewModel の状態をもとに最新化して、計算処理を呼び出しています。CanExecute の処理では、入力に応じて Command が実行可能かどうかを返しています。

```
public DelegateCommand ExecuteCommand { get; private set; }

public MainWindowViewModel()
{
    this.OperatorTypes = OperatorTypeViewModel.OperatorTypes;

    this.ExecuteCommand = new DelegateCommand(this.Execute, this.CanExecute);

    // Modelの監視
    this.appContext.PropertyChanged += this.AppContextPropertyChanged;
    this.appContext.Calc.PropertyChanged += this.CalcPropertyChanged;
}

private void Execute()
{
    this.appContext.Calc.Lhs = double.Parse(this.Lhs);
    this.appContext.Calc.Rhs = double.Parse(this.Rhs);
    this.appContext.Calc.OperatorType = this.SelectedOperatorType.OperatorType;
    this.appContext.Calc.Execute();
}

private bool CanExecute()
{
    double dummy;
    if (!double.TryParse(this.Lhs, out dummy))
    {
        return false;
    }

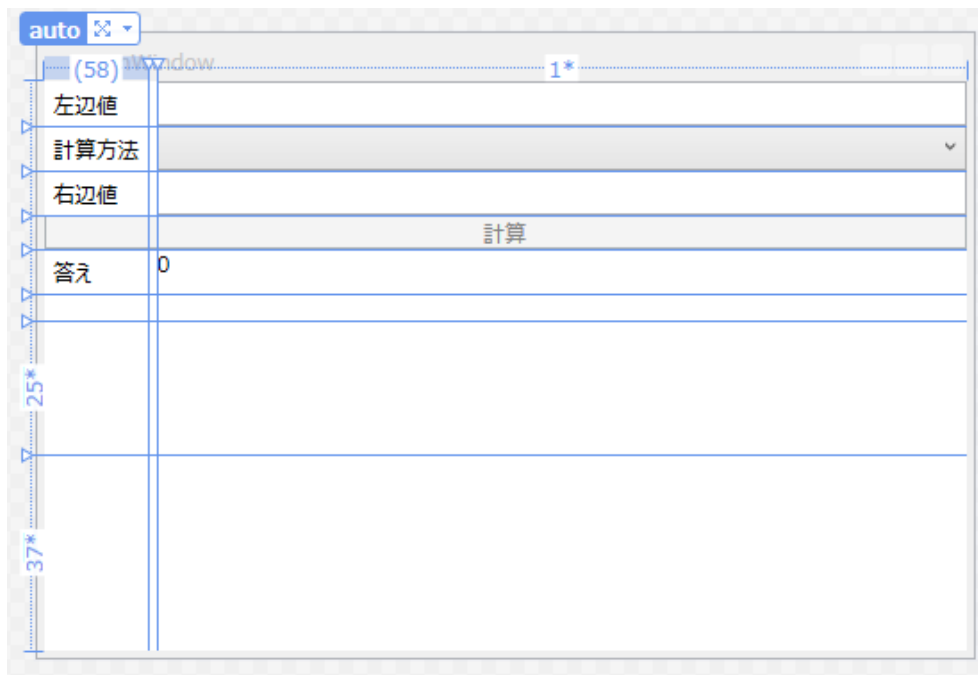
    if (!double.TryParse(this.Rhs, out dummy))
    {
        return false;
    }

    if (this.SelectedOperatorType == null)
    {
        return false;
    }

    return true;
}
```

6.2.5.3. View の作成

最後に ViewModel と View を接続します。View はシンプルに ViewModel に対応した入力項目と出力項目とボタンを持つだけの画面です。見た目は以下ようになります。



XAML を以下に示します。

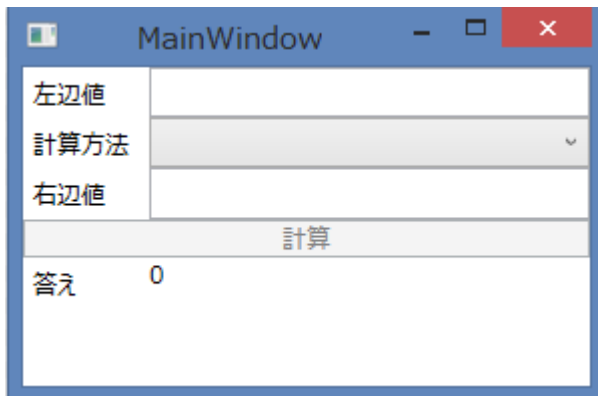
```

<Window
  x:Class="MVVMSample02.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:l="clr-namespace:MVVMSample02"
  Title="MainWindow" Height="350" Width="525">
  <Window.DataContext>
    <l:MainWindowViewModel />
  </Window.DataContext>
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="5"/>
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="25*" />
      <RowDefinition Height="37*" />
    </Grid.RowDefinitions>
    <Label Content="左辺値"/>
    <Label Content="計算方法" Grid.Row="1"/>
    <Label Content="右辺値" Grid.Row="2"/>
    <TextBox Grid.Column="2" TextWrapping="Wrap" Text="{Binding Lhs, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"/>
    <ComboBox Grid.Column="2" Grid.Row="1" ItemsSource="{Binding OperatorTypes}"
SelectedItem="{Binding SelectedOperatorType}" DisplayMemberPath="Label"/>
    <TextBox Grid.Column="2" Grid.Row="2" TextWrapping="Wrap" Text="{Binding Rhc,
Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"/>
    <Label Content="答え" Grid.Row="4"/>
    <TextBlock Grid.Column="2" Grid.Row="4" TextWrapping="Wrap" Text="{Binding Answer}"/>
    <TextBlock Grid.ColumnSpan="3" Grid.Row="5" TextWrapping="Wrap" Text="{Binding
Message}"/>
    <Button Grid.ColumnSpan="3" Content="計算" Grid.Row="3" Command="{Binding
ExecuteCommand, Mode=OneWay}"/>
  </Grid>
</Window>

```

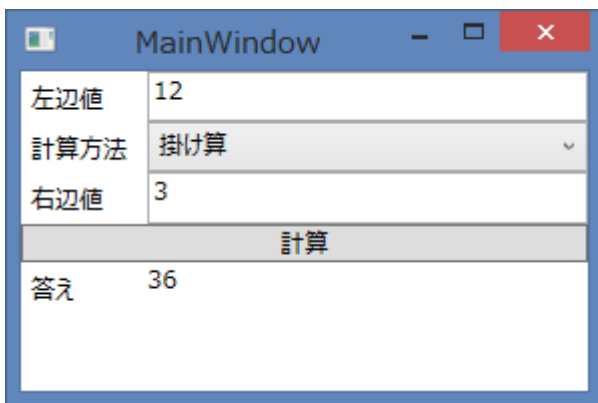
6.2.5.4. 実行して動作確認

実行すると以下のような画面が立ち上がります。



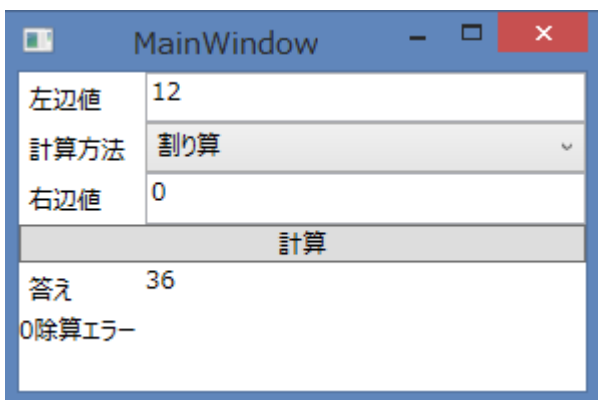
The screenshot shows a window titled "MainWindow" with a standard Windows title bar. Inside, there is a form with three input fields: "左辺値" (Left operand), "計算方法" (Calculation method), and "右辺値" (Right operand). The "計算方法" field is a dropdown menu. Below these fields is a "計算" (Calculate) button. At the bottom, there is a label "答え" (Answer) followed by the value "0".

左辺値と計算方法と右辺値を適当に入力して計算ボタンを押すと以下のように答えに計算結果が表示されます。



The screenshot shows the same "MainWindow" window. The "左辺値" field now contains "12", the "計算方法" dropdown is set to "掛け算" (Multiplication), and the "右辺値" field contains "3". The "計算" button is still present. The "答え" label now displays "36".

0 除算をしようとする以下のようにメッセージが表示されます。



The screenshot shows the "MainWindow" window with "左辺値" set to "12", "計算方法" set to "割り算" (Division), and "右辺値" set to "0". The "計算" button is visible. The "答え" label displays "36". Below the answer, a message "0除算エラー" (Division by zero error) is displayed.

7. さいごに

Windows Presentation Foundation 4.5 入門は、ひとまず終了です。WPF の入門書が少ないというか絶無な日本で少しでも、これから WPF をやろうと思うひとの助けになれば幸いです。

この中で使用しているコードは一部をのぞき以下の GitHub で管理しています。

<https://github.com/runceel/samples/tree/master/wpfedu/WPF4.5>