Windows Presentaion Foundation 4.6 入門

目次

[1. はじめに 7](#_Toc396505381)

[1.1. 本書の目的 7](#_Toc396505382)

[1.2. 本書の対象者 7](#_Toc396505383)

[1.3. 執筆環境 8](#_Toc396505384)

[2. WPFとは 8](#_Toc396505385)

[2.1. WPFのプログラミングモデル 9](#_Toc396505386)

[2.1.1. WPFで作成可能なアプリケーション 9](#_Toc396505387)

[2.2. Hello world 12](#_Toc396505388)

[2.2.1. Appクラス 12](#_Toc396505389)

[2.2.2. MainWindow.xaml 14](#_Toc396505390)

[2.2.3. デザイナによる画面の設計 15](#_Toc396505391)

[2.2.4. イベントハンドラの追加とコードの記述 17](#_Toc396505392)

[2.2.5. コンパイルして実行 17](#_Toc396505393)

[2.2.6. Mainメソッドはどこにいった？ 18](#_Toc396505394)

[2.3. 全てC#でHello world 20](#_Toc396505395)

[2.4. WPFを構成するものを考えてみる 24](#_Toc396505396)

[2.4.1. WPFはクラスライブラリ？ 24](#_Toc396505397)

[2.4.2. XAMLは何？ 25](#_Toc396505398)

[2.4.3. WPFを構成するもののまとめ 26](#_Toc396505399)

[2.5. WPFのコンセプト 27](#_Toc396505400)

[2.5.1. コンテンツモデル 27](#_Toc396505401)

[2.6. スタイル 31](#_Toc396505402)

[2.7. データバインディング 34](#_Toc396505403)

[2.8. まとめ 36](#_Toc396505404)

[3. XAML 36](#_Toc396505405)

[3.1. オブジェクト要素とXAML名前空間 37](#_Toc396505406)

[3.2. オブジェクト要素のプロパティ 38](#_Toc396505407)

[3.3. コレクション構文 42](#_Toc396505408)

[3.4. コンテンツ構文 44](#_Toc396505409)

[3.5. マークアップ拡張 46](#_Toc396505410)

[3.5.1. ProvideValueメソッドのIServiceProviderって何者？ 48](#_Toc396505411)

[3.6. 添付プロパティ 49](#_Toc396505412)

[3.7. 添付イベント 49](#_Toc396505413)

[3.8. TypeConverter 49](#_Toc396505414)

[3.9. その他の機能 50](#_Toc396505415)

[4. WPFのコントロール 50](#_Toc396505416)

[4.1. レイアウト 51](#_Toc396505417)

[4.1.1. Borderコントロール 51](#_Toc396505418)

[4.1.2. BulletDecoratorコントロール 53](#_Toc396505419)

[4.1.3. Canvasコントロール 54](#_Toc396505420)

[4.1.4. StackPanelコントロール 56](#_Toc396505421)

[4.1.5. DockPanelコントロール 60](#_Toc396505422)

[4.1.6. WrapPanelコントロール 62](#_Toc396505423)

[4.1.7. ViewBoxコントロール 65](#_Toc396505424)

[4.1.8. ScrollViewerコントロール 67](#_Toc396505425)

[4.1.9. Gridコントロール 73](#_Toc396505426)

[4.1.10. Gridコントロールでのレイアウト例 79](#_Toc396505427)

[4.1.11. GridSplitterコントロール 83](#_Toc396505428)

[4.1.12. レイアウトに影響を与えるプロパティ 86](#_Toc396505429)

[4.2. ボタン 89](#_Toc396505430)

[4.2.1. Buttonコントロール 89](#_Toc396505431)

[4.2.2. RepeatButtonコントロール 91](#_Toc396505432)

[4.3. データ表示 92](#_Toc396505433)

[4.3.1. DataGridコントロール 92](#_Toc396505434)

[4.3.2. TreeViewコントロール 104](#_Toc396505435)

[4.4. 日付表示および選択 112](#_Toc396505436)

[4.4.1. Calendarコントロール 112](#_Toc396505437)

[4.4.2. DatePickerコントロール 118](#_Toc396505438)

[4.5. メニュー 121](#_Toc396505439)

[4.5.1. ContextMenuコントロール 121](#_Toc396505440)

[4.5.2. Menuコントロール 123](#_Toc396505441)

[4.5.3. ToolBarコントロール 125](#_Toc396505442)

[4.6. 選択系コントロール 128](#_Toc396505443)

[4.6.1. CheckBoxコントロール 128](#_Toc396505444)

[4.6.2. ComboBoxコントロール 130](#_Toc396505445)

[4.6.3. ListBoxコントロール 134](#_Toc396505446)

[4.6.4. RadioButtonコントロール 135](#_Toc396505447)

[4.6.5. Sliderコントロール 137](#_Toc396505448)

[4.7. ナビゲーションコントロール 141](#_Toc396505449)

[4.7.1. TabControl 141](#_Toc396505450)

[4.8. ファイルダイアログ 144](#_Toc396505451)

[4.9. 情報を表示するコントロール 148](#_Toc396505452)

[4.9.1. Labelコントロール 148](#_Toc396505453)

[4.9.2. ProgressBarコントロール 149](#_Toc396505454)

[4.9.3. StatusBarコントロール 150](#_Toc396505455)

[4.9.4. TextBlockコントロール 150](#_Toc396505456)

[4.9.5. Popupコントロール 151](#_Toc396505457)

[4.9.6. ToolTipコントロール 153](#_Toc396505458)

[4.10. 入力 154](#_Toc396505459)

[4.10.1. TextBoxコントロール 154](#_Toc396505460)

[4.11. メディア 156](#_Toc396505461)

[4.11.1. Imageコントロール 156](#_Toc396505462)

[4.11.2. MediaElementコントロール 159](#_Toc396505463)

[5. WPF deep dive 160](#_Toc396505464)

[5.1. DispatcherObject 161](#_Toc396505465)

[5.2. WPFのプロパティシステム 164](#_Toc396505466)

[5.2.1. 依存関係プロパティ 164](#_Toc396505467)

[5.2.2. 添付プロパティ 175](#_Toc396505468)

[5.3. WPFのイベントシステム 178](#_Toc396505469)

[5.3.1. ルーティングイベントの定義方法 179](#_Toc396505470)

[5.3.2. イベントのキャンセル 180](#_Toc396505471)

[5.3.3. 添付イベント 181](#_Toc396505472)

[5.4. コンテンツモデル 181](#_Toc396505473)

[5.5. コントロールテンプレート 181](#_Toc396505474)

[5.6. データバインド 181](#_Toc396505475)

[5.6.1. 単純なデータバインド 181](#_Toc396505476)

[5.6.2. コレクションのデータバインド 181](#_Toc396505477)

[5.6.3. 入力値の検証 182](#_Toc396505478)

[5.7. アニメーション 182](#_Toc396505479)

[5.8. Visual State Manager 182](#_Toc396505480)

[6. 応用 182](#_Toc396505481)

[6.1. 動的にスタイルやテンプレートを切り替える方法 182](#_Toc396505482)

[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 182](#_Toc396505483)

# はじめに

## 本書の目的

2015年7月に.NET Framework 4.6がリリースされました。対応する開発環境としてVisual Studio 2015もリリースされ、Windows 10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 10で動くアプリケーションには大別してUniversal Windows Platform 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。Universal Windows Platform 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](mailto:k.ota.0130@gmail.com)

## 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 2015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

## 執筆環境

本書は、以下の環境で作成しています。

* Windows 10 Pro
* Visual Studio 2015 Ultimate   
  (おそらくCommunity Edition/Express Editionでも同様に作成可能なものが主になると思います)

# 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も、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。
* Universal Windows Platform  
  Windows 10上のすべてのデバイスファミリ（Desktop、Mobile、IoTなど）で実行可能なアプリケーションが1バイナリで作成できるプラットフォーム。

## 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)

画面

UI

(XAML)

コードビハインド（C#）

ロジック

データモデル

Etc…

（C#）

データバインディング

メソッド呼び出し

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。

### 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。

#### 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。

#### ナビゲーション アプリケーション

Windowの中でブラウザー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。ブラウザー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ー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。



#### 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)というブラウザー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、ブラウザーの中に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。

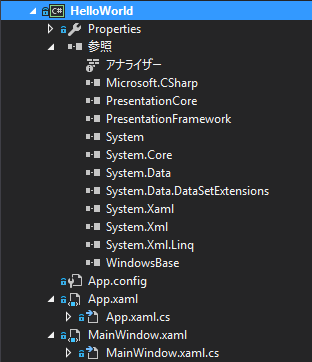


クライアントに.NET Frameworkが必要な点と、インターネットやイントラネットを経由して配備する際にセキュリティに関しての制限事項がある点を除くと、通常のWPFと同じように開発ができます。

## Hello world

ここでは、Visual Studio 2015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 2015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。

### 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。

### 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と設定してあり、高さと幅が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ドの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();

        }

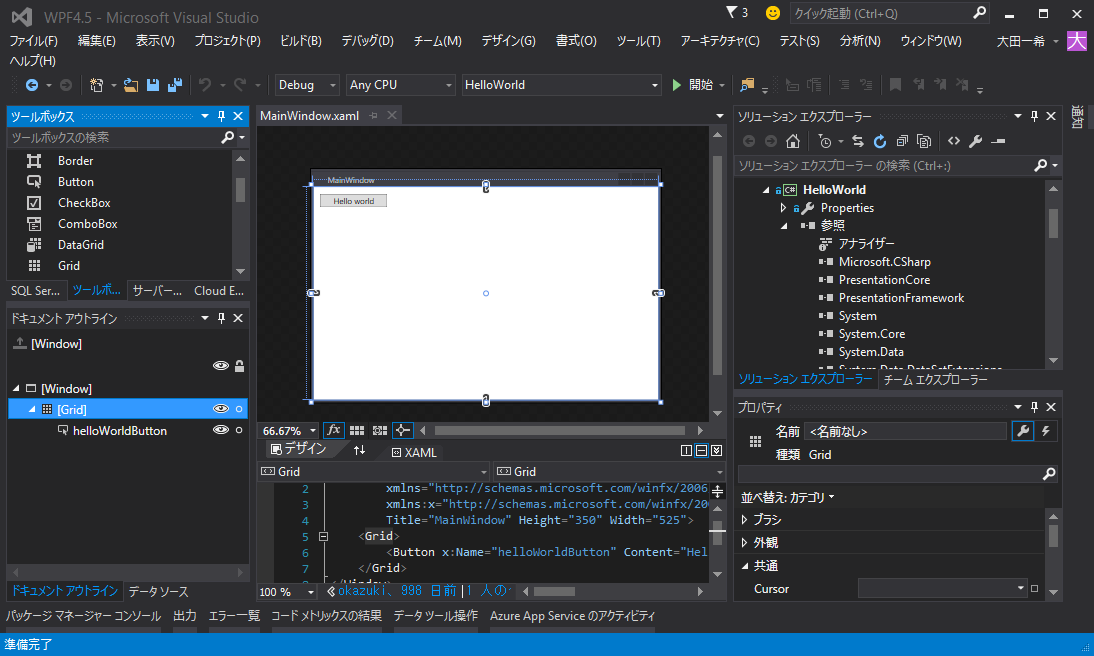
    }

}

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。

### デザイナによる画面の設計

Visual Studio 2015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がついています。主にツールボックスとデザイナ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。



WPFで使用できるコントロールの一覧です。

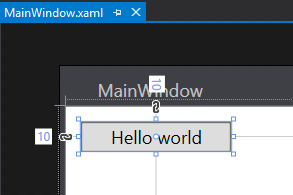
ドラッグアンドドロップでデザイナにコントロールを置くことができます。

デザイナのコントロールの親子関係をツリー状に表示します

画面デザインを行います

デザイナで選択した要素のプロパティを設定します

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。

### 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");

    }

}

### 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。



### 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が、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。

## 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。

* 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、どういうときにXAMLで記述し、どういうときに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。

## 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。

### 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。

### 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。

### 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。

.NET Frameworkのクラス群

WPFのクラス群  
Button, Window, Grid, etc…

記述言語

XAML  
宣言的（オブジェクトのインスタンス化に特化）

C#  
汎用的

相互補完

## 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をしたので、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などで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の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。

### 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のようになります。



##### 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。コードは以下のようになります(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のようになります。



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も柔軟に制御できることが感じていただけると思います。

## 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のようになります。



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を表示すると以下のようになります。



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。

## 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

コントロール  
(ターゲット)

オブジェクト  
(ソース)

バインディング

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。



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。

## 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。

# 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。

## 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。

## 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のようになります。

<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 . . .

## 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は以下のようになります。

<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。

## 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。

## 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う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}のように使用します。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 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 Item;

            Console.WriteLine(item2.Id);

        }

    }

}

実行すると、Idの値が毎回こと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

### 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>

## 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。

## 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。

## 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>

## 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>

# 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。

## 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。

### 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 BorderThickness { get; set; } | 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 BorderBrush { get; set; } | 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 CornerRadius { get; set; } | 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 Background { get; set; } | 背景を塗りつぶすためのブラシを指定します。 |
| Thickness Padding { get; set; } | 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内の余白のとられ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。

### 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を表示すると以下のようになります。



### 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ことロールです。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。



### 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に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 Orientation { get; set; } | 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" />

        <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。

### 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ってしまいます。



このXAMLを実行すると、以下のような結果になります。



### 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、Witdh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。

### 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 Stretc { get; set; } | 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Noneと、領域を埋めるようにサイズを調整するFill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Uniformと、縦横比を維持しながら領域いっぱいにサイズを調整するUniformFillがあります。デフォルトはUniform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。



UniformToFill

Uniform

Fill

None

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い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。

### 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 HorizontalScrollBarVisibility { 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 VerticalScrollBarVisibility { 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れます。

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あScrollableHeigt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。



スクロールバーが中央に移動しています。

### 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は以下のようになります。

<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を表示すると以下のようになります。



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。



5pxの幅の行と列ができます

RowDefinitionとColumnDefinitionの高さと幅の指定方法には、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。



1行1列目にボタンが置かれます。1行目と1列目のサイズはAutoなのでボタンの大きさと同じ大きさになります。

### 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。



DockPanelコントロールの例で作成したレイアウト

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。



Gridの線を表示している以外はDockPanelで作成したサンプルと同じ表示になっています。

### 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を表示すると以下にようになります。



この部分がGridSplitterコントロールです。

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。



マウスでサイズ変更ができます。

### 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て使うことで、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。

#### 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。 |

#### 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で配置されます。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でコントロールが表示されます。

#### 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

#### 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 VerticalAlighment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を表示すると以下のようになります。



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。

#### 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。

## 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。

### 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。



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。



### 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 Interal { 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しっぱなしにした画面を以下に示します。



## 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おさえておきましょう。

### 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。



奇数行・偶数行で色を変える

1つのセルに複数行表示

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。

#### 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

    }

    // DataGrid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

        }));

    // DataGridに設定する

    this.dataGrid.ItemsSource = data;

}

ここで使用しているObservableCollection<T>型は、DataGrid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INotifyColletionChanged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。



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。

#### 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(v=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出来ます。 |

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。



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。

#### 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プロパティやSelectedValuePath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。  表示するデータは、SelectedValueBindingプロパティでItemsSource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SelectedValuePathで指定したプロパティの値、SelectedItemBindingプロパティでItemsSourceプロパティに設定したコレクションで選択中のものと対応づけができます。 |
| DataGridTemplateColumn | DataTemplateを使って表示内容を自由にカスタマイズできます。DataGrid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。 |

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DataGridHeaderColumnクラスです。 |

#### 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。DtaGridTextColumnをNameプロパティに、DataGridCheckBoxColumnをAuthMembeプロパティにバインドしています。

<DataGrid Name="dataGrid" AutoGenerateColumns="False">

    <DataGrid.Columns>

        <DataGridTextColumn 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に指定しています。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="HorizontalContentAlignment" 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。



### 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。



#### 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現します。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; sest; } | 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.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。

#### 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、ルートの要素しか表示されないので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。



## 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。

### 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。

#### 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。



#### 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; } | 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。   * 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; } | 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。 |
| DateTime? SelectedDate { get; set; } | 単一の選択された日付を取得または設定します。 |

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; } | 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。  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。 |

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。



#### 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; se;t } | 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。



### 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。



テキストボックスの横のカレンダーマークを選択すると、テキストボックスの下にカレンダーが表示されます。



カレンダーから選択した日付がテキストボックス部分に表示されます。

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

#### 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。



## 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。

### 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。



#### 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");

}

#### 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出来ます。

|  |  |
| --- | --- |
| 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。 |

#### 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。



### 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。



#### 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="ファイル(\_F)" />

<MenuItem Header="編集(\_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。 |

### 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出来ます。

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

ToggleButton

Separator

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>

#### 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。

## 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。

### 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 |
| public Nullable<bool> IsChecked { get; set; } | 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。 |
| public bool IsThreeState { get; set; } | 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出来ます。

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。



### 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>



#### 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。

### 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。基本的な使用方法は、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; } | 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。   * 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。 |

### 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（StaciPanel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。



#### 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。

### 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 |
| public double Value { get; set; } | Sliderコントロールの現在の値を取得または設定します。 |

#### 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 |
| public double Minimum { get; set; } | Sliderコントロールの最小値を取得または設定します。 |
| public double Maximum { get; set; } | 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。

#### 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。 |

#### 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。   * 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つきのSliderコントロールが表示されます。



## 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>

### 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>

#### 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。



## 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用します。

* 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。



## 情報を表示するコントロール

### 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。

### 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。



### 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>

実行すると以下のようになります。



### 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をつかっ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。



### 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>
* ポップアップの配置動作  
  <http://msdn.microsoft.com/ja-jp/library/bb613596(v=vs.110).aspx>

### 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。



## 入力

### 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。



#### 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。   * 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。 * 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。



## メディア

### 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。



### 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。



# 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。

## 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。

## WPFのプロパティシステム

DispatcherObjectの1段下に継承階層をおりると、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。

### 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

#### 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);

#### 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で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。

#### 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。

#### 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(ValuePeoperty);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。

#### 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);

}

#### 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。

#### 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と役に立ちます。

* FrameworkPropertyMetadataクラス  
  <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>

### 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メソッドを使って行います。

// 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));

## 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の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。

### 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、それの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RoutedEvent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。

### 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。

### 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));

## 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" };

### 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

{

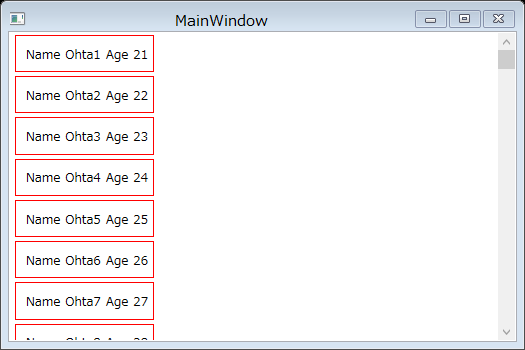
public string Name { get; set; }

public int Age { get; set; }

}

}

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



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の中の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ItemTempalte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>

#### 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ようなことが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" />

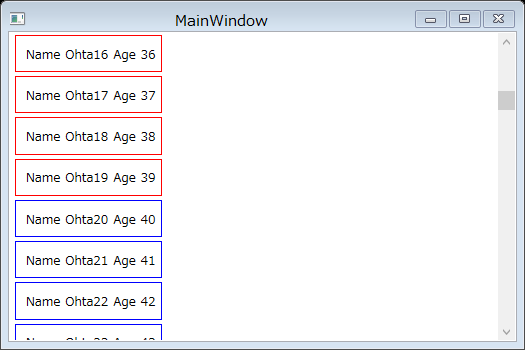
</DataTrigger>

</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。



### DateTemplateSelector

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">

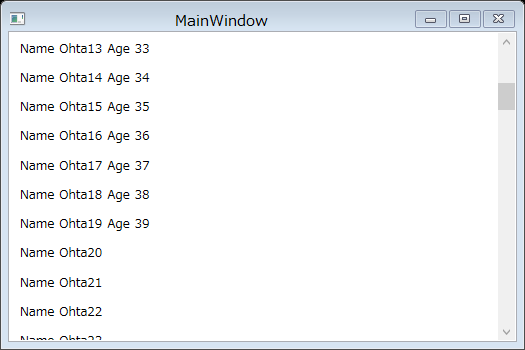
<ListBox.ItemTemplateSelector>

<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。



## 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く値を設定します。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"

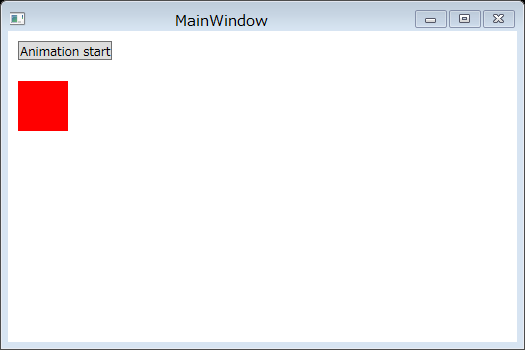
Canvas.Top="50" Canvas.Left="10"

Width="50" Height="50" Fill="Red"/>

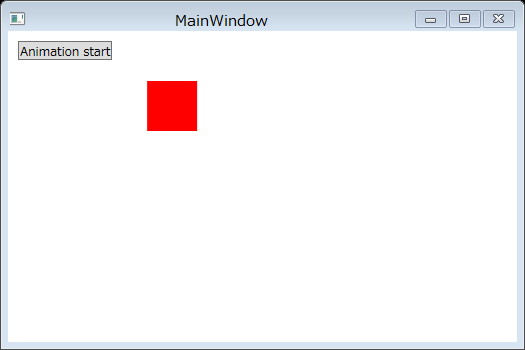
</Canvas>

</Window>

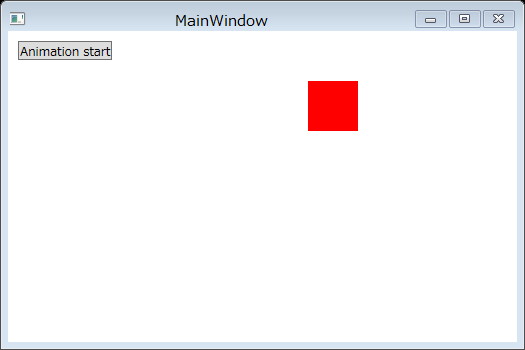
実行結果を以下に示します。起動直後は以下のような配置になっています。



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。



### 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。

### 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。

### 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>

### 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。

### 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>

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。

### イージング関数

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。

* イージング関数  
  <http://msdn.microsoft.com/ja-jp/library/ee308751(v=vs.110).aspx>

イージング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>

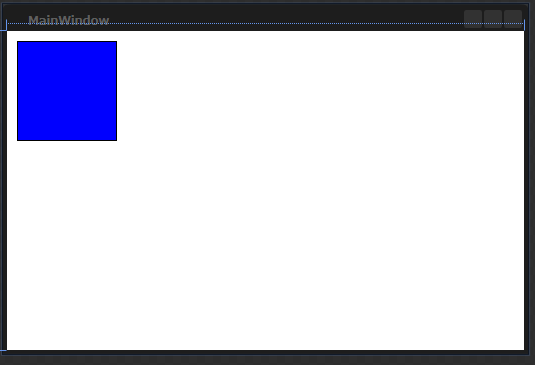
</Storyboard>

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。

### Blendによるアニメーションの作成

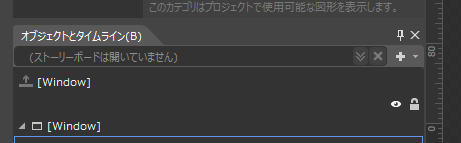
XAMLを使用してアニメーションを作成することは出来ますが、複雑なアニメーションになってくると手書きによるアニメーションは現実的ではなくなってきます。WPFには、Blend for Visual Studioというデザイナ向けのツールがあります。このツールには、Visual Studioにない機能として、アニメーションの作成機能があります。ここでは、簡単にアニメーションの作成機能について紹介します。

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。

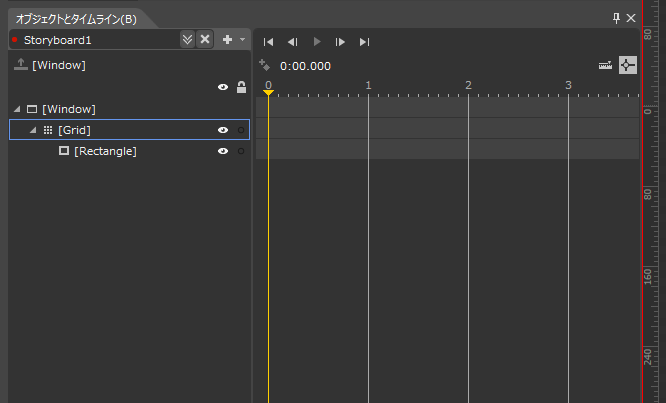


#### 単純なアニメーション

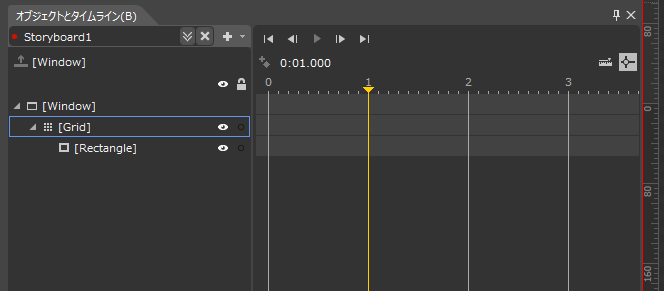
アニメーションを作成するには、「オブジェクトとタイムライン」ウィンドウの＋ボタンを押してStoryboardを新規作成します。



＋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。

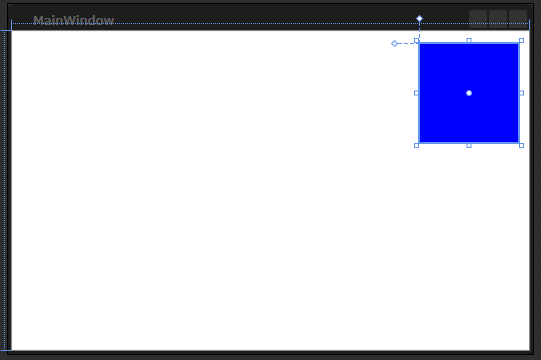


例として1秒後に画面に置いたRectangleを右に移動させる場合のアニメーションの作成方法を示します。まず、オブジェクトとタイムラインウィンドウのタイムラインで、1秒後を選択します。

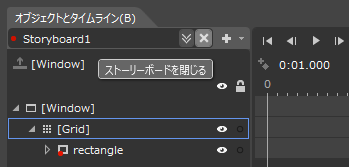


1秒後をタイムライン上で選択する

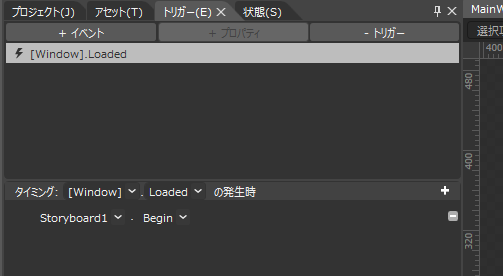
その状態で、デザイナ上でRectangleを右に移動させます。



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。

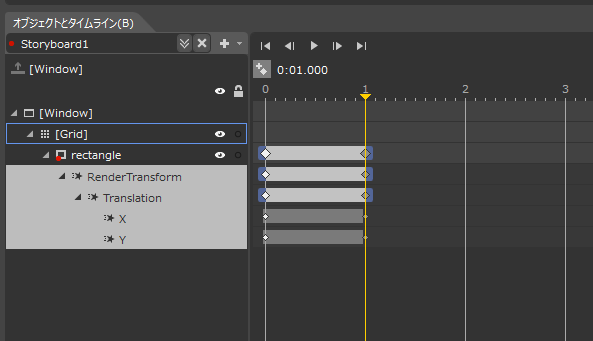


Storyboardの起動はトリガーウィンドウで設定されています。デフォルトではWindowのLoadedイベントに割り当てられています。画面にボタンを置いてButtonのClickなどに割り当てることでアニメーションの起動タイミングを変えることが出来ます。



#### キーフレームアニメーション

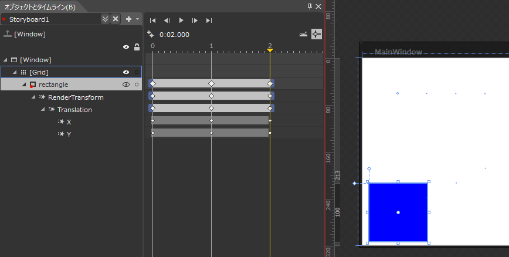
オブジェクトとタイムラインウィンドウで、キーフレームを設定したいアニメーション（この場合RectangleのXとY方向の移動）を選んで、キーフレームを記録するボタンを押すと、その時点でキーフレームが作成されます。



1.キーフレームを設定したアニメーションを選ぶ

2.キーフレームを記録するボタンを押す

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、左下に移動するアニメーションが実行されます。

#### 繰り返し設定

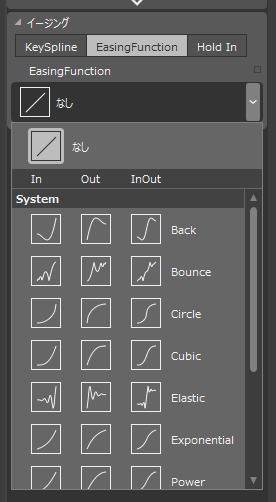
オブジェクトとタイムラインウィンドウで、左上のStoryboardを選択すると、プロパティウィンドウでAutoReverseプロパティとRepeatBehaviorを設定できるようになります。ここで、自動で繰り返すアニメーションや、永久的に実行されるアニメーションの設定などが出来ます。

#### 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。



目的のキーフレームを選択



Hold Inを選択すると、Discreate型名KeyFrameになります。

イージング関数を選択できます。

## 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。

### 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。

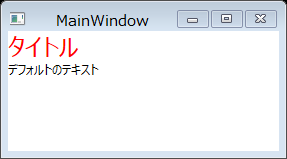
<StackPanel>

<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。



### 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">

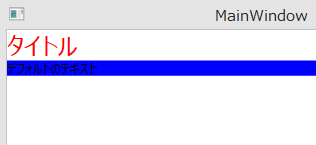
<Setter Property="Background" Value="Blue" />

</Trigger>

</Style.Triggers>

</Style>

この例では、Triggerの中にSetteri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。

## 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、ResourceDicinonary型のResourcesというプロパティが定義されています。ResourceDicinonary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。

### 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。

### 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。

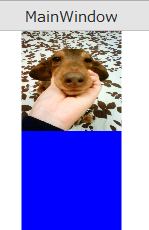
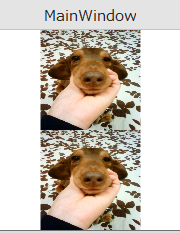
<Border x:Name="border" Width="100" Height="100" Background="{StaticResource AnthemBrush}" />

<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。



StaticResource

DynamicResource

差し替え前

差し替え後

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。存在するかどうかがわ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。

## 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}" />

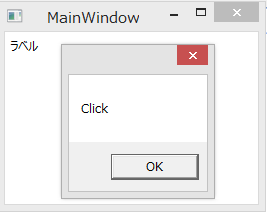
</ControlTemplate>

</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のようになります。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>

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}">

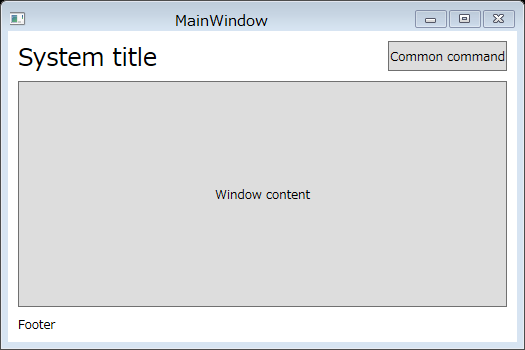
<Grid>

<Button Content="Window content"/>

</Grid>

</Window>

表示すると以下のようになります。



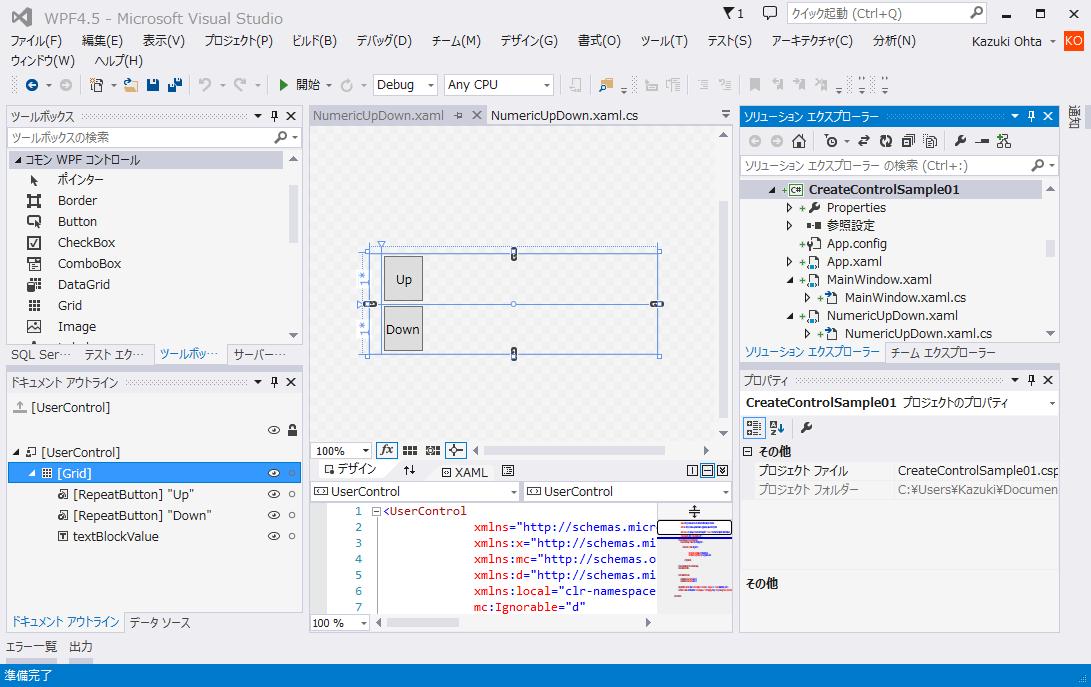
## コントロールの作成

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。

### 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てNumericUpDown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++;

}

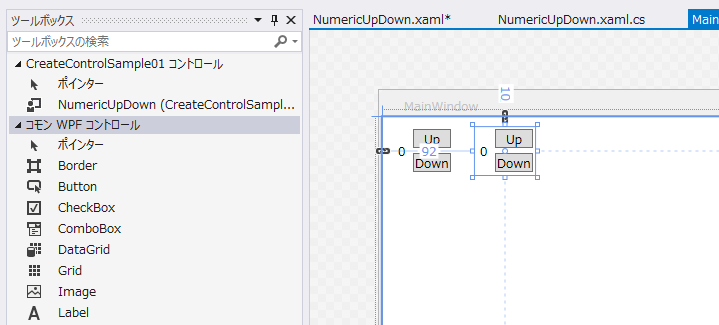
private void DownButton\_Click(object sender, RoutedEventArgs e)

{

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。



#### 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

{

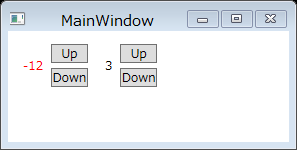
VisualStateManager.GoToState(this, "Negative", useTransition);

}

}

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。

NumeridUpDown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。

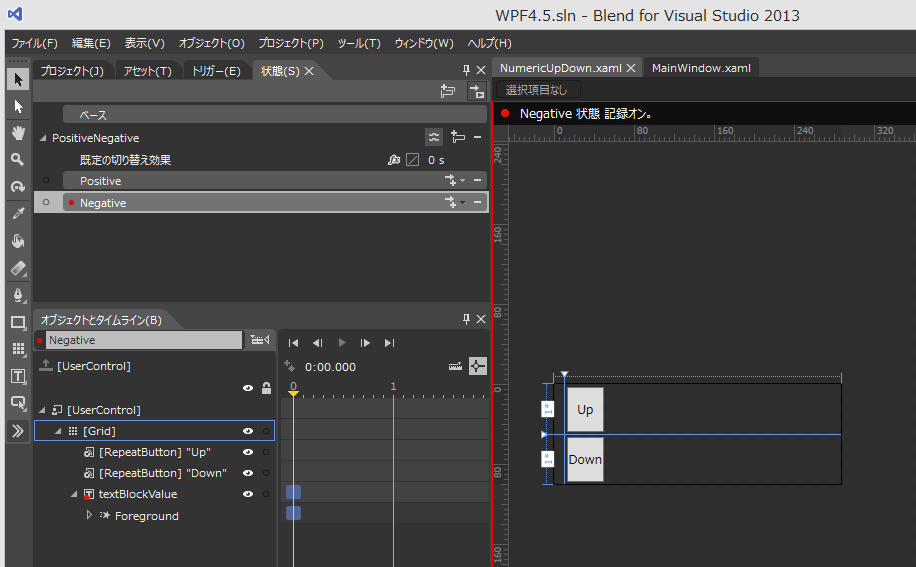


VisualStateManagerは、フォーカスの状態に応じた見た目の管理など組み込みのコントロールの様々な個所で使用されています。

#### 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。



### 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ます。

<StackPanel>

<!-- 通常の見た目 -->

<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>

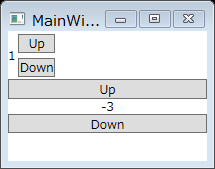
</ControlTemplate>

</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ControlTemplate内に存在することを期待しているか表します。TemplateVisualState属性で、どんなVisualStateGroup内にVisualStateが必要なのか表します。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

## 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いります。

### 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>

### 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>

### 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}" />

### 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。

### 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>

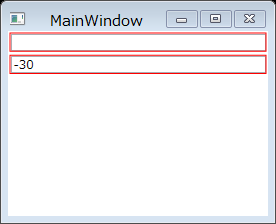
<TextBox Text="{Binding Name}" Margin="2.5" />

<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"

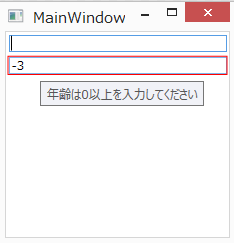
ToolTip="{Binding (Validation.Errors)/ErrorContent, RelativeSource={RelativeSource Self}}" />

<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のようになります。



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、そこの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}" />

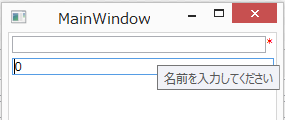
</DockPanel>

</ControlTemplate>

</Validation.ErrorTemplate>

</TextBox>

実行すると以下のように表示されます。



### 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。

#### 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)

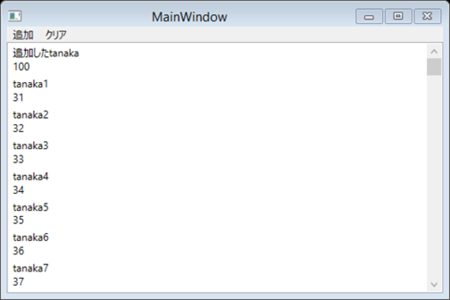
{

// 全削除

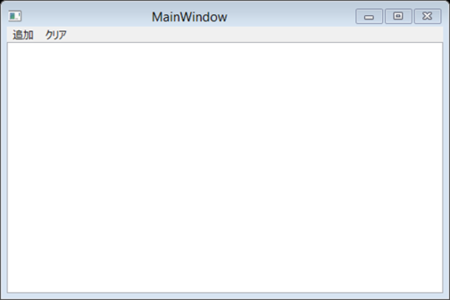
this.people.Clear();

}

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。



#### 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、SortDescritptions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>

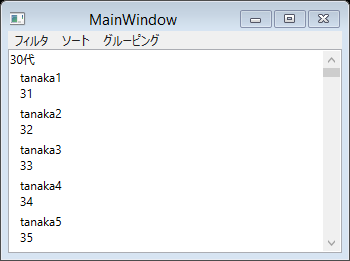
</GroupStyle.HeaderTemplate>

</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。



#### 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。

#### 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随して、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>

#### 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、それの選択中の項目の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 -->

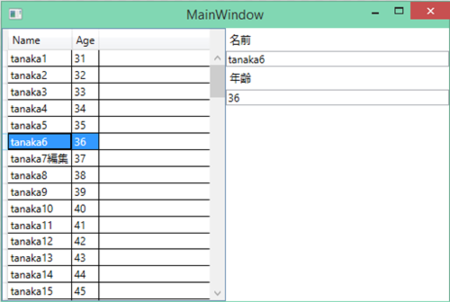
<Label Content="名前"/>

<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。



#### 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て、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。

## 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。

IInputElement 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、以下のように、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)

{

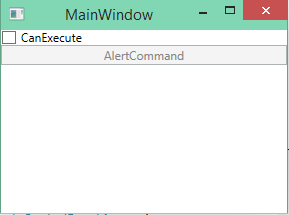
e.CanExecute = this.checkBox.IsChecked.Value;

}

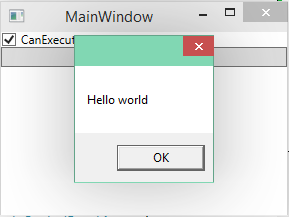
}

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>

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。

# 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。

## 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ると考え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。

### 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。

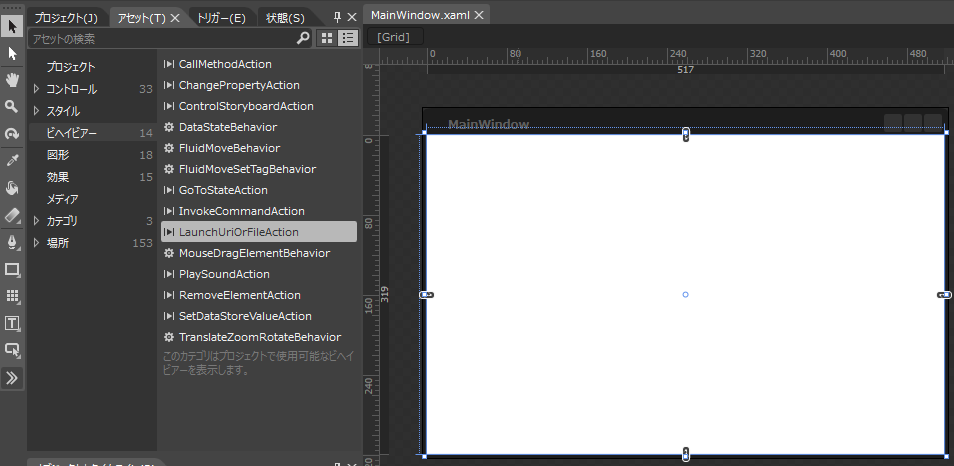
### TriggerとActionの使い方

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。

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。

<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">

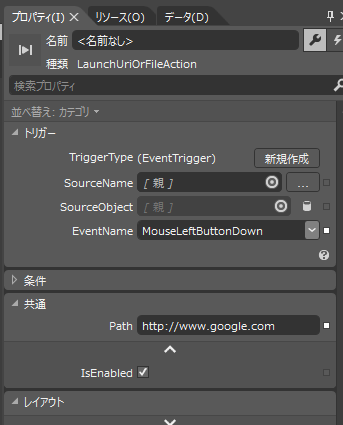
<ei:LaunchUriOrFileAction/>

</i:EventTrigger>

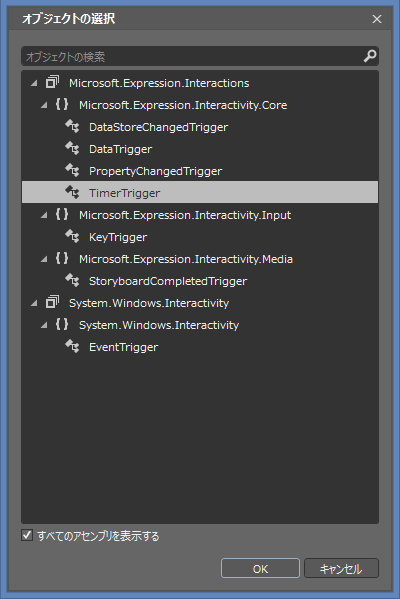
</i:Interaction.Triggers>

</Grid>

</Window>



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。

### 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るようになります。
* 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るようになります。

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。

### 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。では作成したみたいと思います。

#### 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)

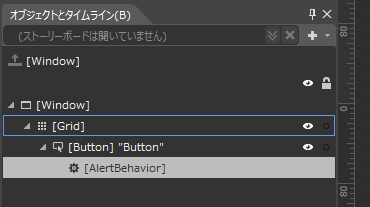
{

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/>

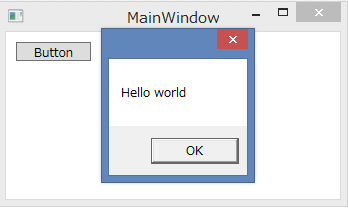
</i:Interaction.Behaviors>

</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。



#### 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を置ける型の指定や、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。

## 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。

### 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。

View

ViewModel

Model

双方向データバインディング

メソッド呼び出し

変更通知

### 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。

### 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。

### 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。

そして、Command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); }

}

/// <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.IsNullOrWhiteSpace(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}" />

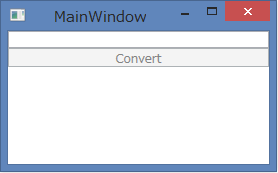
<Button Content="Convert" Command="{Binding ConvertCommand}" />

<TextBlock Text="{Binding Output}" />

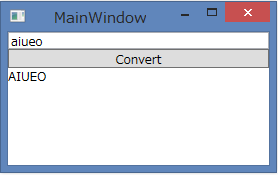
</StackPanel>

</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。



### 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。

#### 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();

}

}

#### 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が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がとじ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;

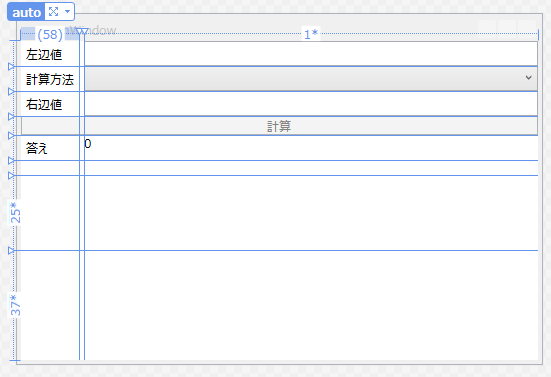
}

return true;

}

#### 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。見た目は以下のようになります。



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 Rhs, 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}"/>

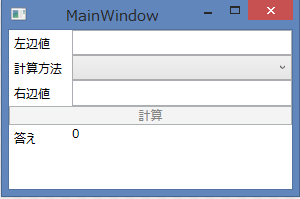
<Button Grid.ColumnSpan="3" Content="計算" Grid.Row="3" Command="{Binding ExecuteCommand, Mode=OneWay}"/>

</Grid>

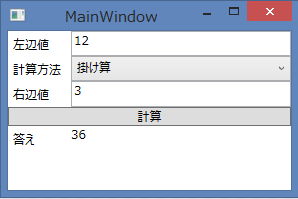
</Window>

#### 実行して動作確認

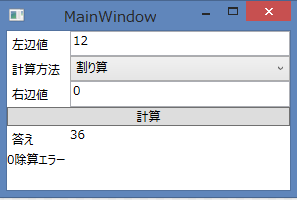
実行すると以下のような画面が立ち上がります。



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。



0除算をしようとすると以下のようにメッセージが表示されます。



### 画面遷移をするアプリケーション

### ダイアログボックスなどの扱いについて