Windows Presentaion Foundation 4.5 入門

目次

[1. はじめに 5](#_Toc345112080)

[1.1. 本書の目的 5](#_Toc345112081)

[1.2. 本書の対象者 5](#_Toc345112082)

[1.3. 執筆環境 6](#_Toc345112083)

[2. WPFとは 6](#_Toc345112084)

[2.1. WPFのプログラミングモデル 7](#_Toc345112085)

[2.2. Hello world 7](#_Toc345112086)

[2.2.1. Appクラス 8](#_Toc345112087)

[2.2.2. MainWindow.xaml 9](#_Toc345112088)

[2.2.3. デザイナによる画面の設計 11](#_Toc345112089)

[2.2.4. イベントハンドラの追加とコードの記述 12](#_Toc345112090)

[2.2.5. コンパイルして実行 12](#_Toc345112091)

[2.2.6. Mainメソッドはどこにいった？ 13](#_Toc345112092)

[2.3. 全てC#でHello world 15](#_Toc345112093)

[2.4. WPFを構成するものを考えてみる 19](#_Toc345112094)

[2.4.1. WPFはクラスライブラリ？ 19](#_Toc345112095)

[2.4.2. XAMLは何？ 20](#_Toc345112096)

[2.4.3. WPFを構成するもののまとめ 21](#_Toc345112097)

[2.5. XAML 22](#_Toc345112098)

[2.5.1. オブジェクト要素とXAML名前空間 22](#_Toc345112099)

[2.5.2. オブジェクト要素のプロパティ 24](#_Toc345112100)

[2.5.3. コレクション構文 28](#_Toc345112101)

[2.5.4. コンテンツ構文 30](#_Toc345112102)

[2.5.5. マークアップ拡張 32](#_Toc345112103)

[2.5.6. 添付プロパティ 35](#_Toc345112104)

[2.5.7. 添付イベント 35](#_Toc345112105)

[2.5.8. TypeConverter 35](#_Toc345112106)

[2.5.9. その他の機能 36](#_Toc345112107)

[2.6. WPFのコントロール 36](#_Toc345112108)

[2.6.1. レイアウト 37](#_Toc345112109)

[2.6.2. ボタン 46](#_Toc345112110)

[2.6.3. データ表示 46](#_Toc345112111)

[2.6.4. 日付表示および選択 46](#_Toc345112112)

[2.6.5. メニュー 46](#_Toc345112113)

[2.6.6. Selection 46](#_Toc345112114)

[2.6.7. Navigation 46](#_Toc345112115)

[2.6.8. ダイアログ ボックス 46](#_Toc345112116)

[2.6.9. ユーザー情報 46](#_Toc345112117)

[2.6.10. ドキュメント 46](#_Toc345112118)

[2.6.11. 入力 46](#_Toc345112119)

[2.6.12. メディア 46](#_Toc345112120)

[2.6.13. デジタル インク 46](#_Toc345112121)

[3. WPF入門 47](#_Toc345112122)

[3.1. DispatcherObject 47](#_Toc345112123)

[3.2. DependencyObject 47](#_Toc345112124)

[3.3. プロパティシステム 47](#_Toc345112125)

[3.4. データバインド 47](#_Toc345112126)

[3.4.1. 単純なデータバインド 47](#_Toc345112127)

[3.4.2. コレクションのデータバインド 47](#_Toc345112128)

[3.4.3. 入力値の検証 47](#_Toc345112129)

[3.5. イベント 47](#_Toc345112130)

[3.6. レイアウトシステム 47](#_Toc345112131)

[3.7. コンテンツモデル 47](#_Toc345112132)

[3.8. 代表的なコントロール 48](#_Toc345112133)

[3.9. リソース 48](#_Toc345112134)

[3.10. スタイル 48](#_Toc345112135)

[3.11. テンプレート 48](#_Toc345112136)

[3.11.1. DataTemplate 48](#_Toc345112137)

[3.11.2. 階層構造を扱うテンプレート 48](#_Toc345112138)

[3.11.3. ControlTemplate 48](#_Toc345112139)

[3.11.4. DataTemplateSelector 48](#_Toc345112140)

[3.12. Visual State Manager 48](#_Toc345112141)

[4. 応用 48](#_Toc345112142)

[4.1. 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 48](#_Toc345112143)

# はじめに

## 本書の目的

2012年8月に.NET Framework 4.5がリリースされました。対応する開発環境としてVisual Studio 2012もリリースされ、Windows 8の時代に対応するアプリケーション開発の環境が整ってきています。Windows 8で動くアプリケーションには大別してWindows ストア アプリとデスクトップアプリケーションの2種類があります。Windows ストア アプリが注目されがちですが、デスクトップアプリケーションも従来と変わらず重要なファクターになります。今後は、デスクトップアプリケーションにもタッチ対応スクリーンへの対応や、拡大されたときの表示などに対応することが求められます。それに対応するためには従来のWindows FormよりもWindows Presentation Foundation(以下WPF)のほうが有利になります。

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

大田　一希  
 [k.ota.0130@gmail.com](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 4.5の学習をしたい方
* C#についての基本的な知識については知っている方  
  具体的には以下のキーワードについて知っている方
  + LINQ
  + async, await
  + ラムダ式
* Visual Studio 2012の基本的な操作方法について理解しているかた
  + プロジェクトの新規作成やクラスなどの新規作成方法
  + 参照の追加方法
  + NuGetを使った参照の追加方法など

## 執筆環境

本書は、以下の環境で作成しています。

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

# 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。

## 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。

## Hello world

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

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



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

### 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 2012には、WPFアプリケーションの画面デザインを行うためのデザイナがついています。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

## 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

### 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

XAML、分離コード、Applicationクラス、Windowクラス、XBAP、ナビゲーション  
http://msdn.microsoft.com/ja-jp/library/aa970268(v=vs.110).aspx

# WPF deep dive

## DispatcherObject

## DependencyObject

## プロパティシステム

## データバインド

### 単純なデータバインド

### コレクションのデータバインド

INotifyCollectionChanged

### 入力値の検証

## イベント

## レイアウトシステム

## コンテンツモデル

http://msdn.microsoft.com/ja-jp/library/bb613548.aspx

## 代表的なコントロール

## リソース

## スタイル

## テンプレート

### DataTemplate

### 階層構造を扱うテンプレート

### ControlTemplate

### DataTemplateSelector

## Visual State Manager

# 応用

## 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