# YKToolkit.Controls

# 取扱説明書

Ver.1.3.1

# 目次

# 図/表/コード目次

# 1 はじめに

この章では本書の目的および執筆環境を掲載します。

## 1.1 目的

本書は WPF 向けコントロールライブラリ YKToolkit.Controls.dll の使用方法をまとめるとともに、開発グループ内での技術共有ならびに WPF の普及を目的としています。

## 1.2 開発環境

本書は以下の環境で執筆しています。

・Windows7 Professional SP1 32 ビットオペレーティングシステム
・Visual Studio Professional 2013 Update4

## 1.3 YKToolkit ファイル群

YKToolkit は下記のファイル群で構成されています。本書では特に YKToolkit.Controls.dll についての取り扱いを説明しています。本書に掲載されていない詳細情報については HelpDocument.chm ヘルプドキュメントファイルをご参照ください。

**表 1.1：YKToolkit ファイル構成一覧表**

| フィル名 | 概 要 |
|---|---|
| YKToolkit.dll | 数学的演算に関するメソッドを提供するための DLL ファイルです。 |
| YKToolkit.AAF1170.dll | 主に NLS 法による推定計算に関するメソッドを拡張するための DLL ファイルです。 |
| YKToolkit.Controls.dll | WPF テーマおよびカスタムコントロールを提供するための DLL ファイルです。 |
| HelpDocument.chm | YKToolkit 全体のヘルプドキュメントファイルです。 |

# 2 WPF の基本的な開発手順

この章では WPF の基本的な開発手順として、まず MVVM パターンを意識した内部構造の作成方法を紹介します。その後、データバインディング機能の要となる INotifyPropertyChanged インターフェースおよび ICommand インターフェースの実装例を紹介し、データバインディング機能の基礎について簡単に紹介します。

## 2.1 MVVM パターンを意識した基本プロジェクト作成方法

WPF によるデスクトップアプリケーションを開発する場合、Visual Studio では "WPF アプリケーション" というプロジェクトを選択して開発を進めます。新規プロジェクトを作成するとき、図 2.1 のように Visual C# の下にある "WPF アプリケーション" を選択し、ソリューション名を決定して下さい。

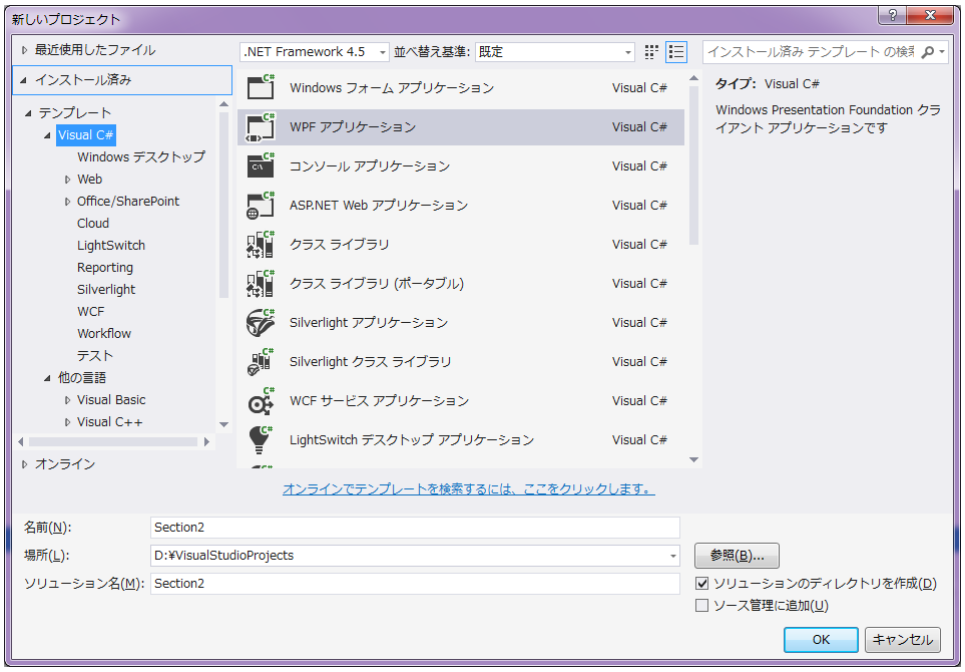



図 2.1：新しいプロジェクト作成ダイアログ

WPF アプリケーションのデフォルトの内部構造は図 2.2 のように App クラスと MainWindow クラスのみとなっています。また、参照設定を見ると、コンソールアプリケーションに比べて PresentationCore、PresentationFramework、System.Xaml、WindowsBase の外部参照が追加されています。

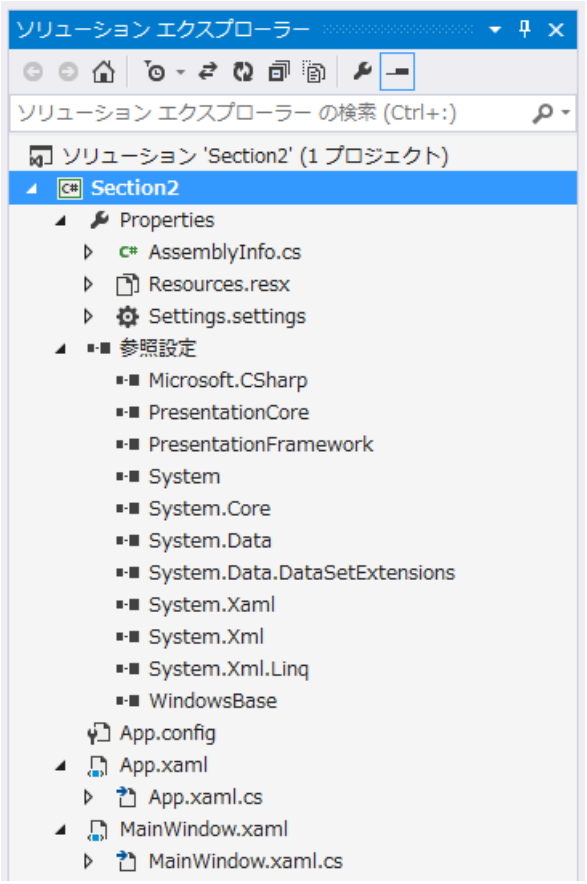



**図 2.2：デフォルトの内部構造**

一方、WPF は UI とロジックを明確に切り分けて開発を進めることができる MVVM パターンによるプログラミングを支援するシステムとして知られています。これを有効活用するにはデフォルトの内部構造では少し不便ですので、使いやすいように変更して使います。

具体的には次のような作業となります。

・MainWindow.xaml（および MainWindow.xaml.cs）を削除
・"Views"、"ViewModels"、"Models" という名前のフォルダをツリーに追加
・"Views" フォルダに "MainView.xaml" をウィンドウとして追加
・"ViewModels" フォルダに "MainViewModel.cs" をクラスとして追加
・App.xaml 内で定義されている StartupUri 属性を削除
・App.xaml.cs 内で OnStartup() メソッドをオーバーライド、編集

以上の作業をおこなった後の内部構造は図 2.3 のようになります。また、編集した App.xaml および App.xaml.cs はコード 2.1、コード 2.2 のようになります。

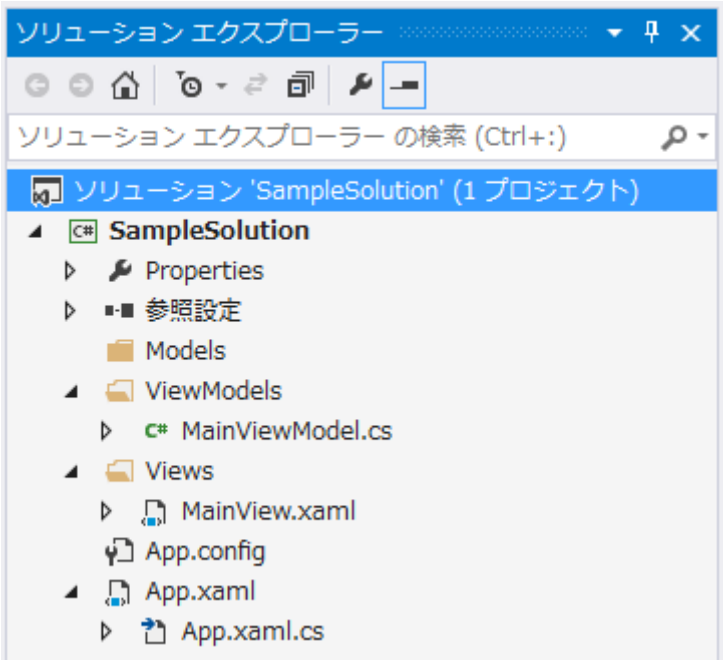



図 2.3：MVVM パターンを意識した内部構造

コード 2.1：StartupUri プロパティを削除した App.xaml

App.xaml

```
1  <Application x:Class="Section2.App"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
4      <Application.Resources>
5
6      </Application.Resources>
7  </Application>
```

コード 2.2：OnStartup メソッドを override した App.xaml.cs

App.xaml.cs

```
1  namespace Section2
2  {
3      using System.Windows;
4      using Section2.Views;
5      using Section2.ViewModels;
6
7      /// <summary>
8      /// App.xaml の相互作用ロジック
9      /// </summary>
10     public partial class App : Application
11     {
12         protected override void OnStartup(StartupEventArgs e)
13         {
14             base.OnStartup(e);
15
16             var w = new MainView();
17             var vm = new MainViewModel();
18
19             w.DataContext = vm;
20             w.Show();
21         }
22     }
23 }
```

このように MainView の DataContext に対して MainViewModel を指定することで、MainView と MainViewModel の間でデータバインディング機能を用いたデータのやり取りができるようになります。しかし、データバインディング機能を使用するためには、ViewModel 側は INotifyPropertyChanged インターフェースや ICommand インターフェースを実装したプロパティを公開する必要があります。

## 2.2 簡単な UI の作成

データバインディング機能を説明する前に、簡単な UI を作成します。MainView.xaml をコード 2.3 のように編集します。縦にコントロールを並べる StackPanel コントロールの中に、TextBox コントロール、TextBlock コントロール、Button コントロールをひとつずつ配置しただけのシンプルな画面です。これをコンパイルすると図 2.4 のような画面が表示されます。

**コード 2.3：コントロールを配置した MainView**

**MainView.xaml**

```
1  <Window x:Class="Section2.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="300" Width="300">
5      <StackPanel>
6          <TextBox Text="Hello world." />
7          <TextBlock Text="Hello world." />
8          <Button Content="Click me." />
9      </StackPanel>
10 </Window>
```

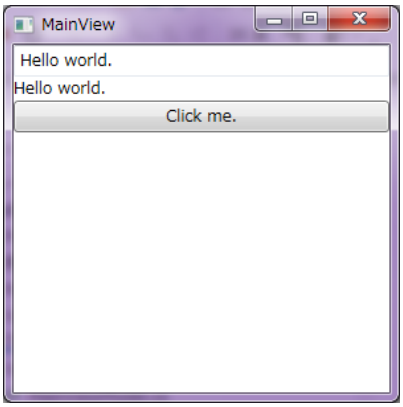



**図 2.4：サンプル画面**

## 2.3 INotifyPropertyChanged インターフェースの自前実装と具体例

データバインディング機能を使うために、ViewModel 側に INotifyPropertyChanged インターフェースを実装したプロパティを定義します。YKToolkit.Controls.dll を利用することで INotifyPropertyChanged インターフェースが既に実装されたクラスを扱うことができますが、ここでは敢えて INotifyPropertyChanged インターフェースを自前で実装して、その内部構造を知ってもらおうと思います。

INotifyPropertyChanged インターフェースを MainViewModel に実装すると次のようなコードになります。

**コード 2.4：INotifyPropertyChanged を実装した MainViewModel クラス**

**MainViewModel.cs**

```
1  namespace Section2.ViewModels
2  {
3      using System.ComponentModel;
4
5      public class MainViewModel : INotifyPropertyChanged
6      {
```

```
7           #region INotifyPropertyChanged のメンバ
8           /// <summary>
9           /// プロパティ変更時に発生します。
10          /// </summary>
11          public event PropertyChangedEventHandler PropertyChanged;
12          #endregion INotifyPropertyChanged のメンバ
13
14          /// <summary>
15          /// PropertyChanged イベントを発行します。
16          /// </summary>
17          /// <param name="propertyName">プロパティ名を指定します。</param>
18          protected virtual void RaisePropertyChanged(string propertyName)
19          {
20              var h = PropertyChanged;
21              if (h != null)
22                  h(this, new PropertyChangedEventArgs(propertyName));
23          }
24      }
25  }
```

INotifyPropertyChanged インターフェースの目的は、View 側に ViewModel のプロパティが変更されたことを通知することです。したがって、ViewModel 側で公開しているプロパティに変更があったときに、RaisePropertyChanged() メソッドをコールする必要があります。

具体例として、Text プロパティおよび Result プロパティを次のように定義します。それぞれの get アクセサでは、private フィールドの内容をそのまま返しています。set アクセサでは、private フィールドで保持している値と異なる値がセットされようとしたとき、private フィールドの内容を更新するとともに RaisePropertyChanged() メソッドをコールすることで、対応するプロパティが変更されたことを View 側に通知しています。

ここでは Text プロパティが変更されたとき、すべて大文字にした文字列を Result プロパティに設定するようにしています。

**コード 2.5：プロパティの追加とその変更通知**

**MainViewModel.cs の一部**

```
1   private string text;
2   /// <summary>
3   /// 文字列を取得または設定します。
4   /// </summary>
5   public string Text
6   {
7       get { return text; }
8       set
9       {
10          if (text != value)
11          {
12              text = value;
13              Result = text.ToUpper();
14              RaisePropertyChanged("Text");
15          }
16      }
17  }
18
19  private string result;
20  /// <summary>
21  /// 処理結果を取得または設定します。
22  /// </summary>
23  public string Result
24  {
25      get { return result; }
26      set
```

```
27      {
28          if (result != value)
29          {
30              result = value;
31              RaisePropertyChanged("Result");
32          }
33      }
34  }
```

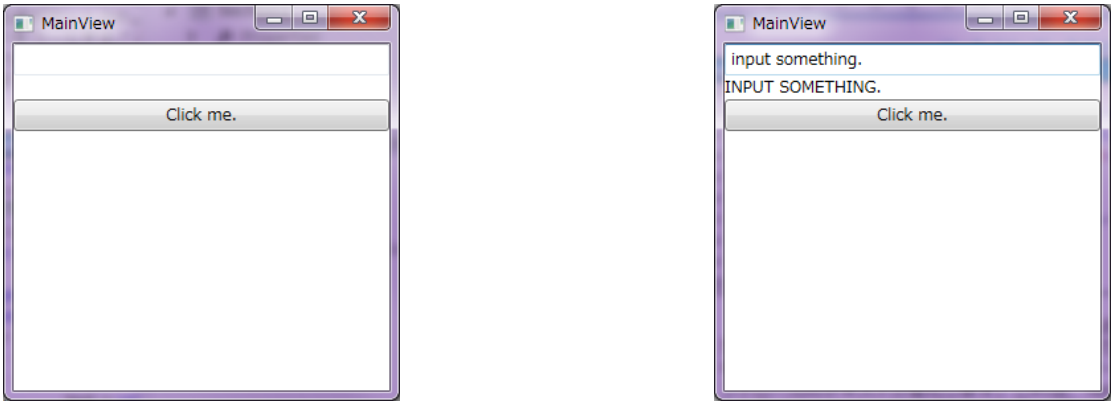
次に、これらのプロパティを参照するように、MainView を次のように変更します。

**コード 2.6：データバインディング機能を利用した MainView**

**MainView.xaml**

```
1  <Window x:Class="Section2.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="300" Width="300">
5      <StackPanel>
6          <TextBox Text="{Binding Text}" />
7          <TextBlock Text="{Binding Result}" />
8          <Button Content="Click me." />
9      </StackPanel>
10 </Window>
```

コンパイル、実行すると、Buton コントロールおよび TextBox コントロールの中身が MainViewModel で定義した Text プロパティの内容と一致するようになるため、起動直後は空白となります。TextBox コントロール内のテキストを変更した後、TextBox コントロールのキーボードフォーカスを外す（Tab キーを押す）と、TextBlock コントロールのテキストが変化します。これは、Text プロパティの変更によって Result プロパティが変更され、その変更が TextBlock コントロールの Text プロパティに伝わっているからです。




(a) 起動直後　　(b) テキスト変更後

**図 2.5：Text プロパティがバインディングされた画面**

View 側からのプロパティ変更通知のタイミングは設定によって変更できます。デフォルト値は LostFocus といって、フォーカスを失ったときに通知されるようになっています。テキスト内容を変更した時点で MainViewModel にその変更が通知されるようにする場合は、PropertyChanged という設定値にする必要があります。具体的なコードは次のようになります。

**コード 2.7：UpdateSourceTrigger を指定したデータバインディング**

**MainView.xaml の一部**

```
1  <TextBox Text="{Binding Text, UpdateSourceTrigger=PropertyChanged}" />
```

## 2.4 ICommand インターフェースの自前実装と具体例

次に、ボタンを押すことによって実行されるコマンドを、データバインディング機能を用いて ViewModel 側に記述する方法を紹介します。コマンドを使用するには、ICommand インターフェースを実装したプロパティが必要になります。前節と同様に、YKToolkit.Controls.dll を利用することで ICommand インターフェースが既に実装されたクラスを扱うことができますが、ここでは敢えて ICommand インターフェースを自前で実装して、その内部構造を知ってもらおうと思います。

DelegateCommand クラスという ICommand インターフェースを実装するクラスを次のように定義します。

**コード 2.8：DelegateCommand クラスの定義**

**DelegateCommand.cs**

```
1  namespace Section2
2  {
3      using System;
4      using System.Windows.Input;
5
6      public class DelegateCommand : ICommand
7      {
8          /// <summary>
9          /// コマンドの実体を保持します。
10         /// </summary>
11         private Action<object> _execute;
12
13         /// <summary>
14         /// コマンドの実行可能判別処理の実態を保持します。
15         /// </summary>
16         private Func<object, bool> _canExecute;
17
18         /// <summary>
19         /// 新しいインスタンスを生成します。
20         /// </summary>
21         /// <param name="execute">コマンドの実体を指定します。</param>
22         public DelegateCommand(Action<object> execute)
23             : this(execute, null)
24         {
25         }
26
27         /// <summary>
28         /// 新しいインスタンスを生成します。
29         /// </summary>
30         /// <param name="execute">コマンドの実体を指定します。</param>
31         /// <param name="canExecute">コマンドの実行可能判別処理の実体を指定します。</param>
32         public DelegateCommand(Action<object> execute, Func<object, bool> canExecute)
33         {
34             _execute = execute;
35             _canExecute = canExecute;
36         }
37
38         /// <summary>
39         /// CanExecuteChanged イベントを発行します。
40         /// </summary>
41         public static void RaiseCanExecuteChanged()
42         {
43             CommandManager.InvalidateRequerySuggested();
44         }
45
46         #region ICommand のメンバ
47         /// <summary>
48         /// コマンドの実行可能判別処理を実行します。
```

```
49              /// </summary>
50              /// <param name="parameter">コマンドパラメータを指定します。</param>
51              /// <returns>コマンドが実行可能であるとき true を返します。</returns>
52              public bool CanExecute(object parameter)
53              {
54                  return _canExecute == null ? true : _canExecute(parameter);
55              }
56
57              /// <summary>
58              /// コマンドの実行可能判別条件が変更されたときに発生します。
59              /// </summary>
60              public event System.EventHandler CanExecuteChanged
61              {
62                  add { CommandManager.RequerySuggested += value; }
63                  remove { CommandManager.RequerySuggested += value; }
64              }
65
66              /// <summary>
67              /// コマンドを実行します。
68              /// </summary>
69              /// <param name="parameter">コマンドパラメータを指定します。</param>
70              public void Execute(object parameter)
71              {
72                  if (_execute != null)
73                      _execute(parameter);
74              }
75              #endregion ICommand のメンバ
76          }
77      }
```

ICommand インターフェースは、実際にコマンドの内容を実施する Execute() メソッドと、このコマンド自体が実行可能かどうかを判別するための CanExecute() メソッドを実装する必要があります。コマンドの内容や実行可能かどうかの条件をここで固定させる必要はないため、_execute および _canExecute という private フィールドを用意し、コマンドの中身や実行可能判別の処理を外部から指定できるようにしています。

実行可能かどうかに影響するような変更があった場合、RaiseCanExecuteChanged() メソッドを呼び出すことで、CanExecuteChanged イベントを発行し、その変更を View 側に伝えます。ここでは、CanExecuteChanged イベントを CommandManager.RequerySuggested イベントに委任しているため、こちらのイベントを発生させることで CanExecuteChanged イベントを発生させています。ただし、同じ View（ViewModel）に複数のコマンドが存在し、この RaiseCanExecuteChanged() メソッドを呼び出した場合、すべてのコマンドに対して再評価がおこなわれるため、各コマンドに対して呼ぶ必要はありません。このことから、RaiseCanExecuteChanged() メソッドは static なメソッドとして定義しています。

具体例として、ClearCommand プロパティを次のように定義します。get アクセサによるプロパティ値取得タイミングで、private フィールドである clearCommand 変数が null のときのみ DelegateCommand クラスをインスタンス化しています。コマンドの中身は Text プロパティを空にするという処理で、Text プロパティが既に空である場合は実行不可であるという判別をおこなっています。

**コード 2.9：ClearCommand プロパティの定義**

**MainViewModel.cs の一部**

```
1   private DelegateCommand clearCommand;
2   /// <summary>
3   /// 文字列をクリアするコマンドを取得します。
4   /// </summary>
5   public DelegateCommand ClearCommand
6   {
7       get
8       {
9           if (clearCommand == null)
10              clearCommand = new DelegateCommand(
```

```
11                  _ => Text = string.Empty,
12                  _ => !string.IsNullOrEmpty(Text)
13                  );
14              return clearCommand;
15          }
16      }
```

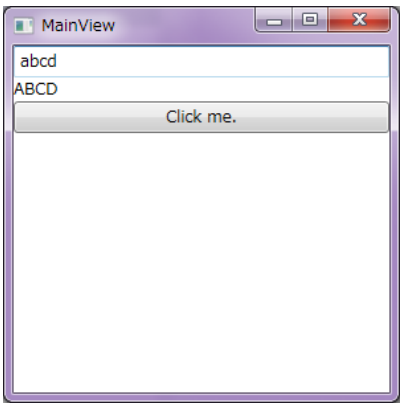
次に、このプロパティを参照するように、MainView を次のように変更します。

**コード 2.10：データバインディング機能を利用した MainView**

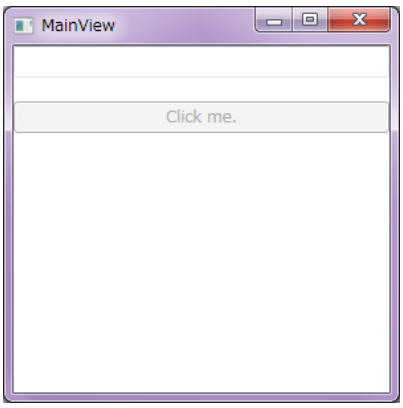
**MainView.xaml**

```
1   <Window x:Class="Section2.Views.MainView"
2           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4           Title="MainView" Height="300" Width="300">
5       <StackPanel>
6           <TextBox Text="{Binding Text, UpdateSourceTrigger=PropertyChanged}" />
7           <TextBlock Text="{Binding Result}" />
8           <Button Content="Click me." Command="{Binding ClearCommand}" />
9       </StackPanel>
10  </Window>
```

コンパイル、実行すると、起動直後は Text プロパティが空であるため、Button コントロールの Command プロパティの実行可能判別処理によって Button コントロールの IsEnabled プロパティが false となっています。TextBox コントロールに任意の文字列を入力して Text プロパティに文字列が設定されると、Button コントロールの実行可能判別処理によって Button コントロールの IsEnabled プロパティが true となり、ボタンを押せるようになります。また、ボタンを押すと ClearCommand のコマンド実行処理がおこなわれ、Text プロパティが空になるため、TextBox コントロールのテキストが空になります。




(a) テキストを入力するとボタンが有効になる




(b) ボタンを押すと TextBox コントロールが空になる

**図 2.6：ClearCommand プロパティがバインディングされた画面**

## 2.5 サンプルプロジェクト

| ソリューション名 | 概　要 |
|---|---|
| Section2 | INotifyPropertyChanged インターフェースおよび ICommand インターフェースを自前で実装したサンプルプログラム |

# 3 YKToolkit.Controls.dll の導入方法

この章では、プロジェクトに YKToolkit.Controls.dll を導入し、これを利用する簡単な例を紹介します。

## 3.1 参照設定

ここでは、前章で紹介した MVVM パターンを意識した基本プロジェクトを作成し、これに YKToolkit.Controls.dll を導入します。

前章で紹介した MVVM パターンを意識した基本プロジェクトを作成した後、ソリューションエクスプローラのツリー上にある "参照設定" を右クリックし、"参照の追加" を選択します。すると "参照マネージャー" ダイアログが開くので、ダイアログ右下にある "参照" ボタンを押して YKToolkit.Controls.dll を選択して下さい。こうすることで、プロジェクトに YKToolkit.Controls.dll が埋め込まれるようになります。

## 3.2 NotificationObject クラス

データバインディング機能によって View あるいは ViewModel にプロパティ値の変更を通知するには INotifyPropertyChanged インターフェースを実装する必要がありますが、YKtoolkit.Controls.dll では、これをあらかじめ実装した NotificationObject クラスが用意されています。これを利用すると、前節の Text プロパティや Result プロパティは次のように記述できます。

**コード 3.1：NotificationObject クラスを利用したプロパティ定義**

**MainViewModel.cs**

```
1  namespace Section3_2.ViewModels
2  {
3      using YKToolkit.Bindings;
4
5      public class MainViewModel : NotificationObject
6      {
7          private string text;
8          /// <summary>
9          /// 文字列を取得または設定します。
10         /// </summary>
11         public string Text
12         {
13             get { return text; }
14             set
15             {
16                 if (SetProperty(ref text, value))
17                 {
18                     Result = text.ToUpper();
19                 }
20             }
21         }
22
23         private string result;
24         /// <summary>
25         /// 処理結果を取得または設定します。
26         /// </summary>
27         public string Result
28         {
29             get { return result; }
```

```
30                  set { SetProperty(ref result, value); }
31              }
32          }
33      }
```

SetProperty() メソッドは NotificationObject クラスで定義されており、第一引数に変更対象となる private フィールド変数、第二引数に変更後の値、第三引数に変更されるプロパティ名を指定します。また、プロパティ値が変更された場合は true を返し、変更が無い場合は false を返します。ただし、第三引数は省略でき、この場合はひとつのプロパティ値が変更されるとそのクラス全体のプロパティ値が変更されたことを通知するようになります。このようなプロパティ変更通知のヘルパが用意されていることで、前節と比べてプロパティ定義が非常に簡便になっていることが窺えます。

## 3.3 DelegateCommand クラス

データバインディング機能によって Command プロパティを紐付けるには ICommand インターフェースを実装したプロパティを公開する必要がありますが、YKToolkit.Controls.dll では、ICommand インターフェースを実装した DelegateCommand クラスが用意されています。これを利用すると、前節の ClearCommand プロパティは次のように同じように記述できます。

**コード 3.2：DelegateCommand クラスを利用したプロパティ定義**

**MainViewModel.cs の一部**

```
1   private DelegateCommand clearCommand;
2   /// <summary>
3   /// 文字列をクリアするコマンドを取得します。
4   /// </summary>
5   public DelegateCommand ClearCommand
6   {
7       get
8       {
9           if (clearCommand == null)
10              clearCommand = new DelegateCommand(
11              _ => Text = string.Empty,
12              _ => !string.IsNullOrEmpty(Text)
13              );
14          return clearCommand;
15      }
16  }
```

実行結果は前節で示したサンプルプロジェクトと同じとなるため省略します。

## 3.4 コントロール外観のテーマ

WPF では、コントロールに対してテーマを設定することで、その配色や形等を統一したものに変更することができます。テーマを作成するためには、ControlTemplate によって各コントロールの基本外観を自作する必要があります。YKToolkit.Controls.dll では濃色および淡色のテーマを用意しています。ここでは、それらのテーマをどのようにして導入するかを説明します。

YKToolkit.Controls.dll では、標準の Window クラスとは異なるウィンドウを用意しており、テーマはこのウィンドウ上にコントロールを配置することを想定して作成されています。したがって、まず System.Windows.Window クラスを YKToolkit.Controls.Window クラスに置き換える必要があります。

例えば MainView クラスの System.Windows.Window クラスを YKToolkit.Controls.Window クラスに置き換える場合、MainView.xaml と MainView.xaml.cs の両方を次のように編集します。

**コード 3.3：XAML における YKToolkit.Controls.Window クラスへの変更**

**MainView.xaml**

```
1   <YK:Window x:Class="Section3_4.Views.MainView"
```

```
2                 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3                 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                 xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                 Title="MainView" Height="300" Width="300">
6       <Grid>
7
8       </Grid>
9   </YK:Window>
```

**コード 3.4：コードビハインドにおける YKToolkit.Controls.Window クラスへの変更**

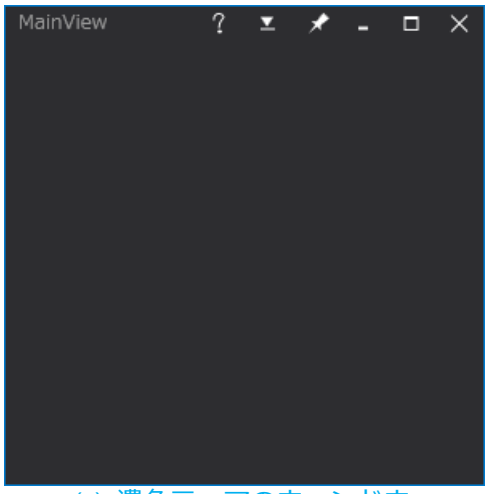
**MainView.xaml.cs**

```
1   namespace Section3_4.Views
2   {
3       using YKToolkit.Controls;
4
5       /// <summary>
6       /// MainView.xaml の相互作用ロジック
7       /// </summary>
8       public partial class MainView : Window
9       {
10          public MainView()
11          {
12              InitializeComponent();
13          }
14      }
15  }
```

XAML の方は、YKtoolkit.Controls 名前空間を使用するために xmlns:YK="..." としてエイリアス "YK" を定義しています。そのエイリアスを使用して、先頭の Window タグを YK:Window に変更することで、System.Windows.Window クラスから YKToolkit.Controls.Window クラスへ変更しています。

コードビハインドである MainView.xaml.cs は、MainView.xaml で定義した MainView クラスの partial クラスであるため、そのクラス名や基底クラスは XAML で記述したものと一致していなければなりません。したがって、コードビハインドのほうで基底クラスとして System.Windows.Window クラスを指定していた部分を、上記のように YKToolkit.Controls.Window クラスに変更しなければなりません。

コンパイル、実行すると、下図のような外観のウィンドウが表示されます。右上には最小化、最大化（元に戻す）、閉じるボタン以外に、YKToolkit.Controls のバージョン確認ボタン、テーマ変更ボタン、常に手前に表示ボタンが追加されています。これらの機能は YKToolkit.Controls.Window クラスで定義されているため、特別なコーディング作業をすることなくこれらの機能が使用できます。




(a) 濃色テーマのウィンドウ


(b) 淡色テーマのウィンドウ

**図 3.1：YKToolkit.Controls のテーマとその切り替え**

YKToolkit.Controls.Window クラスの外観はデフォルトで濃色テーマとなります。起動時のテーマを淡色テーマにする場合は、App.xaml.cs 等で次のようなコードを記述します。

**コード 3.5：淡色テーマをデフォルトにする**

**MainView.xaml.cs**

```
1  namespace Section3_4
2  {
3      using Section3_4.Views;
4      using System.Windows;
5      using YKToolkit.Controls;
6
7      /// <summary>
8      /// App.xaml の相互作用ロジック
9      /// </summary>
10     public partial class App : Application
11     {
12         protected override void OnStartup(StartupEventArgs e)
13         {
14             base.OnStartup(e);
15
16             // 淡色テーマをデフォルトにする
17             ThemeManager.Initialize("Light");
18
19             var w = new MainView();
20             w.Show();
21         }
       }
   }
```

Button コントロールや TextBox コントロール等、ほとんどの標準コントロールに対するテーマを用意しているため、XAML でこれらのコントロールを配置するだけで YKToolkit.Controls.dll で定義されたテーマが反映されます。例えば次のような XAML の実行結果を下図に示します。

**コード 3.6：標準コントロールの配置**

**MainView.xaml.cs**

```
1  <YK:Window x:Class="Section3_5.Views.MainView"
2              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4              xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5              Title="MainView" Height="300" Width="300">
6      <StackPanel>
7          <TextBox Text="Input text." />
8          <Button Content="Click me." />
9          <CheckBox Content="Check me." IsThreeState="True" />
10         <ComboBox SelectedIndex="0">
11             <ComboBoxItem>Item1</ComboBoxItem>
12             <ComboBoxItem>Item2</ComboBoxItem>
13             <ComboBoxItem>Item3</ComboBoxItem>
14         </ComboBox>
15     </StackPanel>
16 </YK:Window>
```

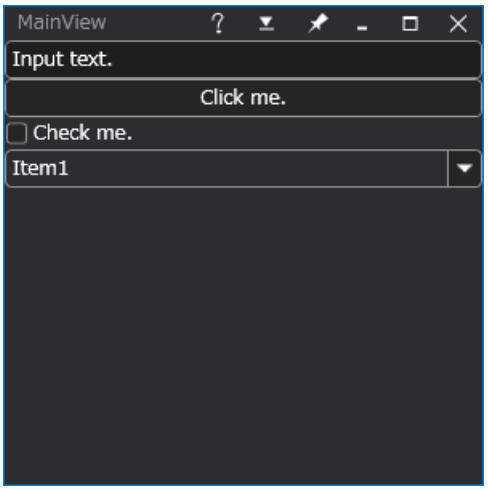



図 3.2：標準コントロールの配置

## 3.5 サンプルプロジェクト

| ソリューション名 | プロジェクト名 | 概　要 |
|---|---|---|
| Section3 | Section3_2 | Section2 サンプルを YKToolkit.Controls に置き換えたもの |
| | Section3_4 | YKToolkit.Controls.Window コントロールを使用し、テーマ変更をおこなうサンプルプログラム |
| | Section3_5 | YKToolkit.Controls.Window コントロール上に標準コントロールを配置したサンプルプログラム |

# 4 Extended Controls

この章では、YKToolkit.Controls.dll で公開されている、標準コントロール以外の Extended Controls を紹介します。各コントロールのプロパティやイベント等の詳細については YKToolkit 付属のヘルプドキュメントファイルをご参照ください。

## 4.1 概要

YKToolkit.Controls.dll では、標準コントロールにはない機能を持ったコントロールを Extended Controls として公開しています。下表に Extended Controls 一覧を掲載します。

**表 4.1：Extended Controls 一覧表**

| コントロール名 | 概 要 |
|---|---|
| BarGraph | 棒グラフを表示するコントロールです。 |
| BusyIndicator | ビジー状態を示すインジケータを表示するコントロールです。 |
| ColorPicker | 色選択をおこなうためのコントロールです。よく DropDownButton コントロールと併用されます。 |
| DirectorySelectDialog | ディレクトリ選択ダイアログコントロールです。 |
| DropDownButton | 任意のコンテンツをドロップダウン形式で表示させることができるボタンコントロールです。 |
| FileTreeView | ローカルのファイル構造をツリー形式で表示するコントロールです。 |
| LineGraph | 折れ線グラフを表示するコントロールです。 |
| MessageBox | メッセージダイアログを表示し、ユーザと対話するためのコントロールです。 |
| SpinInput | 数値入力のためのコントロールです。入力支援としてインクリメントボタンおよびデクリメントボタンを備えています。 |
| TextBox | テキストの表示および編集をおこなうためのコントロールです。ウォーターマーク表示に対応しています。 |
| Transition | View の Type 情報を利用して画面遷移をおこなうコントロールです。 |
| TransitionControl | 指定する ViewModel を切り替えることで対応する View を切り替える画面遷移コントロールです。 |

## 4.2 BarGraph

BarGraph コントロールは、BarGraphItem クラスのコレクションをデータとする棒グラフを描画します。棒グラフを表示する XAML サンプルプログラムを下記コードに示します。

**コード 4.1：BarGraph コントロールのサンプルコード**

**MainView.xaml**

```
1  <YK:Window x:Class="Section4_2.Views.MainView"
2                  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3                  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                  xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                  xmlns:sys="clr-namespace:System;assembly=mscorlib"
6                  Title="MainView" Height="400" Width="600"
7                  WindowStartupLocation="CenterScreen">
8      <Grid>
9          <YK:BarGraph Title="サンプルグラフ"
10                          YLabel="データ数">
11             <YK:BarGraph.ItemsSource>
12                 <x:Array Type="{x:Type YK:BarGraphItem}">
```

```
13                          <YK:BarGraphItem Title="Data1" Fill="Coral" Stroke="Coral" Values="{Binding Data1}" />
14                          <YK:BarGraphItem Title="Data2" Fill="Cyan" Stroke="Cyan" Values="{Binding Data2}" />
15                      </x:Array>
16                  </YK:BarGraph.ItemsSource>
17                  <YK:BarGraph.XAxisItemsSource>
18                      <x:Array Type="{x:Type sys:String}">
19                          <sys:String>みかん</sys:String>
20                          <sys:String>りんご</sys:String>
21                      </x:Array>
22                  </YK:BarGraph.XAxisItemsSource>
23                  <YK:BarGraph.XAxisItemTemplate>
24                      <DataTemplate>
25                          <TextBlock Text="{Binding}" TextAlignment="Center" />
26                      </DataTemplate>
27                  </YK:BarGraph.XAxisItemTemplate>
28                  <YK:BarGraph.BorderBrush>
29                      <SolidColorBrush Color="{DynamicResource BorderColor}" />
30                  </YK:BarGraph.BorderBrush>
31                  <YK:BarGraph.Background>
32                      <SolidColorBrush Color="{DynamicResource WindowColor}" />
33                  </YK:BarGraph.Background>
34              </YK:BarGraph>
35          </Grid>
36      </YK:Window>
```

Grid コントロールの中に BarGraph コントロールをひとつだけ配置しています。グラフデータは ItemsSource プロパティに BarGraphItem クラスを IEnumerable を実装した形で指定します。上記サンプルでは配列として指定するため、12 行目のように <x:Array /> を使用しています。横軸の項目も同じように IEnumerable を実装した形で指定します。上記サンプルでは 17 行目から指定しています。ただし、グラフデータである BarGraphItem の数と、横軸の項目である XAxisItemsSource に指定する項目数が一致していないと、グラフが表示されないことにご注意ください。

上記サンプルでは、グラフデータである BarGraphItem に対して、Title プロパティや Fill プロパティは XAML から直接指定していますが、 Values プロパティについてはデータバインディングの機能を使用することで、ViewModel から値をもらうようにしています。このように、表示に関するプロパティは XAML から、データに関するプロパティは ViewModel、引いては Model から参照するようにすることができます。

次に、上記サンプルに対する ViewModel のサンプルコードを示します。

**コード 4.2：BarGraph コントロールのための ViewModel のサンプルコード**

**MainViewModel.cs**

```
1   namespace Section4_2.ViewModels
2   {
3       using System.Collections;
4       using System.Linq;
5       using YKToolkit.Bindings;
6
7       public class MainViewModel : NotificationObject
8       {
9           /// <summary>
10          /// 新しいインスタンスを生成します。
11          /// </summary>
12          public MainViewModel()
13          {
14              SetRandomData();
15          }
16
17          #region 公開プロパティ
18          private IEnumerable data1;
19          /// <summary>
```

```
20              /// 1 つ目のサンプルデータのデータ列を取得または設定します。
21              /// </summary>
22              public IEnumerable Data1
23              {
24                  get { return data1; }
25                  set { SetProperty(ref data1, value); }
26              }
27
28              private IEnumerable data2;
29              /// <summary>
30              /// 2 つ目のサンプルデータのデータ列を取得または設定します。
31              /// </summary>
32              public IEnumerable Data2
33              {
34                  get { return data2; }
35                  set { SetProperty(ref data2, value); }
36              }
37              #endregion 公開プロパティ
38
39              /// <summary>
40              /// ランダムなグラフデータを生成します。
41              /// </summary>
42              private void SetRandomData()
43              {
44                  var rnd = new System.Random();
45
46                  Data1 = Enumerable.Range(0, 3).Select(_ => rnd.Next(20, 80));
47                  Data2 = Enumerable.Range(0, 3).Select(_ => rnd.Next(20, 80));
48              }
49          }
50      }
```

2 つのグラフデータ Data1 および Data2 プロパティを公開し、それぞれにランダムなデータを与えています。上記のサンプルコードの実装結果は下図のようになります。

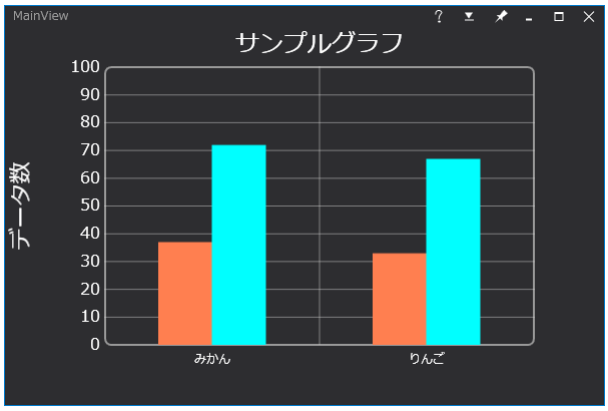



**図 4.1：BarGraph コントロール**

BarGraph コントロールにはこの他にも様々なプロパティが用意されています。また、グラフ上を右クリックすることでメニューが表示され、ユーザ操作によってグラフの設定を変更することができます。

## 4.3 BusyIndicator

BusyIndicator コントロールは、UI が処理中であることを示すためのインジケータを表示するためのコントロールです。バックグラウンドで処理をおこないながら UI を更新するため、主に非同期処理をおこなっている場合に使用します。ここでは async/await による簡単な非同期処理を使用します。

まず、重たい処理を表現するために、次のような SampleModel クラスを定義します。本当におこないたい処理は DoHeavyWorkCore() メソッドですが、これを実行するタスクを返すメソッドが DoHeavyWorkAsync() メソッドです。そして、DoHeavyWorkAsync() メソッドのほうを公開することで、外側からは同じタスクで重たい処理をさせないようにしています。

**コード 4.3：SampleModel クラスのコード**

**SampleModel.cs**

```
1  namespace Section4_3.Models
2  {
3      using System.Threading;
4      using System.Threading.Tasks;
5
6      public class SampleModel
7      {
8          /// <summary>
9          /// 重たい処理を別タスクでおこないます。
10         /// </summary>
11         /// <returns></returns>
12         public Task DoHeavyWorkAsync()
13         {
14             return Task.Run(() =>
15             {
16                 DoHeavyWorkCore();
17             });
18         }
19
20         /// <summary>
21         /// 重たい処理をおこないます。
22         /// </summary>
23         private void DoHeavyWorkCore()
24         {
25             // 仮想的な重たい処理として 10[s] 間待機する
26             Thread.Sleep(10000);
27         }
28     }
29 }
```

次に BusyIndicator コントロールを使った簡単なサンプルコードを示します。Button コントロールと TextBlock コントロールを縦に並べた StackPanel コントロールの上に被さるように BusyIndicator コントロールを配置しています。BusyIndicator コントロールは、IsBusy プロパティが true にならない限り表示されないため、普段は StackPanel コントロールの内容のみが表示される形となります。

なお、BusyIndicator には Message プロパティがあります。下記のサンプルコードでは TextBlock コントロールを別に用意していますが、BusyIndicator コントロールのインジケータ下部にメッセージを表示させることもできます。こちらの場合、メッセージ表示用に UI のスペースを確保する必要がなくなります。

**コード 4.4：BusyIndicator コントロールのサンプルコード**

**MainView.xaml**

```
1  <YK:Window x:Class="Section4_3.Views.MainView"
2              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4              xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5              Title="MainView" Height="500" Width="500"
6              WindowStartupLocation="CenterScreen">
7      <Grid>
```

```
8           <StackPanel VerticalAlignment="Center">
9               <Button Content="Click me." Command="{Binding ButtonCommand}" Margin="20" />
10              <TextBlock Margin="20">
11                  <Run Text="Message :" />
12                  <Run Text="{Binding Message, Mode=OneWay}" />
13              </TextBlock>
14          </StackPanel>
15
16          <YK:BusyIndicator IsBusy="{Binding IsBusy}" />
17      </Grid>
18  </YK:Window>
```

上記の MainView と SampleModel をつなぐための MainViewModel クラスのコードを次に示します。
MainView のボタンを押すと重たい処理が開始されるようにするため、ButtonCommand プロパティから非同期処理が開始されるようにしています。非同期処理をおこなうには、処理をおこなうタスクの前に await 演算子を置きます。また、await 演算子を使用することを許可するために、そのメソッドの修飾子に async を追加します。

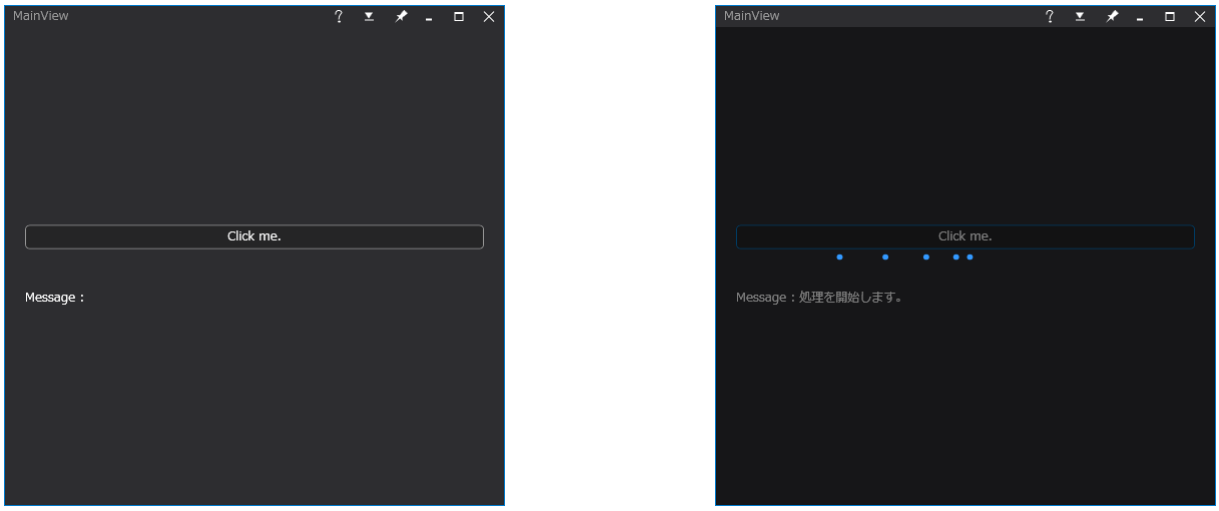
**コード 4.5：BusyIndicator コントロールのための ViewModel のサンプルコード**

**SampleModel.cs**

```
1   namespace Section4_3.ViewModels
2   {
3       using Section4_3.Model;
4       using YKToolkit.Bindings;
5
6       public class MainViewModel : NotificationObject
7       {
8           /// <summary>
9           /// サンプルモデルのインスタンス
10          /// </summary>
11          private SampleModel model = new SampleModel();
12
13          #region 公開プロパティ
14          private bool isBusy;
15          /// <summary>
16          /// ビジー状態かどうかを取得します。
17          /// </summary>
18          public bool IsBusy
19          {
20              get { return isBusy; }
21              private set { SetProperty(ref isBusy, value); }
22          }
23          private string message;
24          /// <summary>
25          /// メッセージを取得または設定します。
26          /// </summary>
27          public string Message
28          {
29              get { return message; }
30              set { SetProperty(ref message, value); }
31          }
32
33          private DelegateCommand buttonCommand;
34          /// <summary>
35          /// ボタンのコマンドを取得します。
36          /// </summary>
37          public DelegateCommand ButtonCommand
38          {
39              get
```

```
40                  {
41                      if (buttonCommand == null)
42                          buttonCommand = new DelegateCommand(_ =>
43                          {
44                              DoCommand();
45                          });
46                      return buttonCommand;
47                  }
48              }
49              #endregion 公開プロパティ
50
51              /// <summary>
52              /// 処理をおこないます。
53              /// </summary>
54              private async void DoCommand()
55              {
56                  Message = "処理を開始します。";
57                  IsBusy = true;
58
59                  // 重たい処理を非同期でおこなう
60                  await model.DoHeavyWorkAsync();
61
62                  IsBusy = false;
63                  Message = "処理を終了しました。";
64              }
65          }
66      }
```

上記のサンプルコードの実装結果は下図のようになります。




(a) 起動直後 (b) ボタン操作後

**図 4.2 : BusyIndicator コントロールのサンプル画面**

## 4.4 ColorPicker

ColorPicker コントロールは、UI の操作によってユーザに色を選択できるようにするためのコントロールです。DropDownButton コントロールと併用することでより使いやすいコントロールとなります。

ColorPicker コントロールを使ったサンプル UI のコードを次に示します。

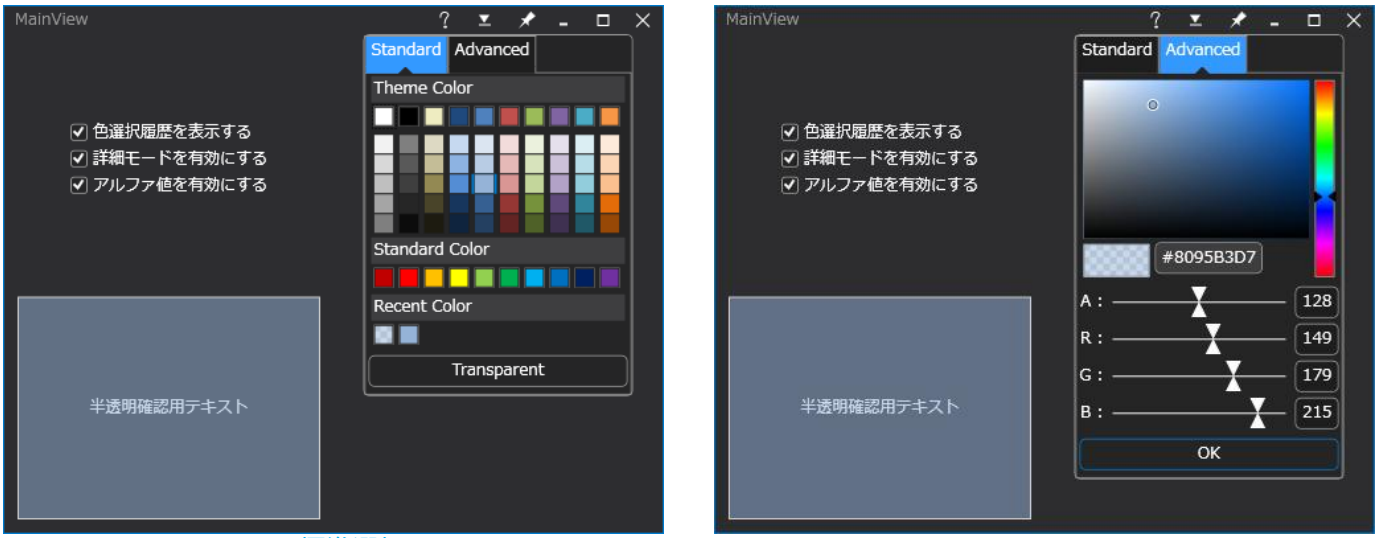
8 行目の BooleanToVisibilityConverter クラスは、true/false である Boolean 型を Visible/Hidden/Collapsed である Visibility クラスに変換するためのコンバータで、データバインドの際に使用するためにこのように StaticResource として定義しています。

コード 4.6：ColorPicker コントロールのサンプルコード

**MainView.xaml**

```
<YK:Window x:Class="Section4_4.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
        Title="MainView" Height="400" Width="500"
        WindowStartupLocation="CenterScreen">
    <YK:Window.Resources>
        <BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter" />
    </YK:Window.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
            <CheckBox x:Name="IsRecentColorEnabledChange" Content="色選択履歴を表示する" />
            <CheckBox x:Name="IsAdvancedModeEnabledChange" Content="詳細モードを有効にする" />
            <CheckBox x:Name="IsAlphaValueEnabledChange" Content="アルファ値を有効にする" />
        </StackPanel>

        <Grid Grid.Row="1" Margin="10">
            <TextBlock Text="半透明確認用テキスト"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center"
                       />
            <Rectangle Stroke="LightGray">
                <Rectangle.Fill>
                    <SolidColorBrush Color="{Binding SelectedColor, ElementName=colorPicker}" />
                </Rectangle.Fill>
            </Rectangle>
        </Grid>

        <YK:ColorPicker Grid.RowSpan="2" Grid.Column="1"
                        x:Name="colorPicker"
                        HorizontalAlignment="Center"
                        VerticalAlignment="Top"
                        RecentColorsVisibility="{Binding IsChecked,
ElementName=IsRecentColorEnabledChange, Converter={StaticResource BooleanToVisibilityConverter}}"
                        IsAdvancedModeEnabled="{Binding IsChecked,
ElementName=IsAdvancedModeEnabledChange}"
                        AlphaValueVisibility="{Binding IsChecked,
ElementName=IsAlphaValueEnabledChange, Converter={StaticResource BooleanToVisibilityConverter}}"
                        />
    </Grid>
</YK:Window>
```

上記のサンプルコードの実装結果は下図のようになります。




(a) 標準選択 (b) 拡張選択

**図 4.3：ColorPicker コントロールのサンプル画面**

## 4.5 DirectorySelectDialog

DirectorySelectDialog コントロールは、ディレクトリを選択し、そのフルパスを取得するためのダイアログで、Window クラスから派生したコントロールです。

DirectorySelectDialog コントロールを使用したサンプルコードを以下に示します。

**コード 4.7：DirectorySelectDialog コントロールのサンプルコード**

**MainView.xaml**

```
<YK:Window x:Class="Section4_5.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
        xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
        Title="MainView" Height="300" Width="300">
    <Grid>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center" VerticalAlignment="Center">
            <TextBox Text="{Binding DirectoryName}" Width="200" Margin="10,0" />
            <Button Content="..."
                    YKb:DirectorySelectDialogBehavior.RoutedEvent="Button.Click"
                    YKb:DirectorySelectDialogBehavior.Title="フォルダ選択"
                    YKb:DirectorySelectDialogBehavior.DirectoryName="{Binding DirectoryName}"
                    />
        </StackPanel>
    </Grid>
</YK:Window>
```

**コード 4.8：DirectorySelectDialog コントロールのための ViewModel のサンプルコード**

**MainViewModel.cs**

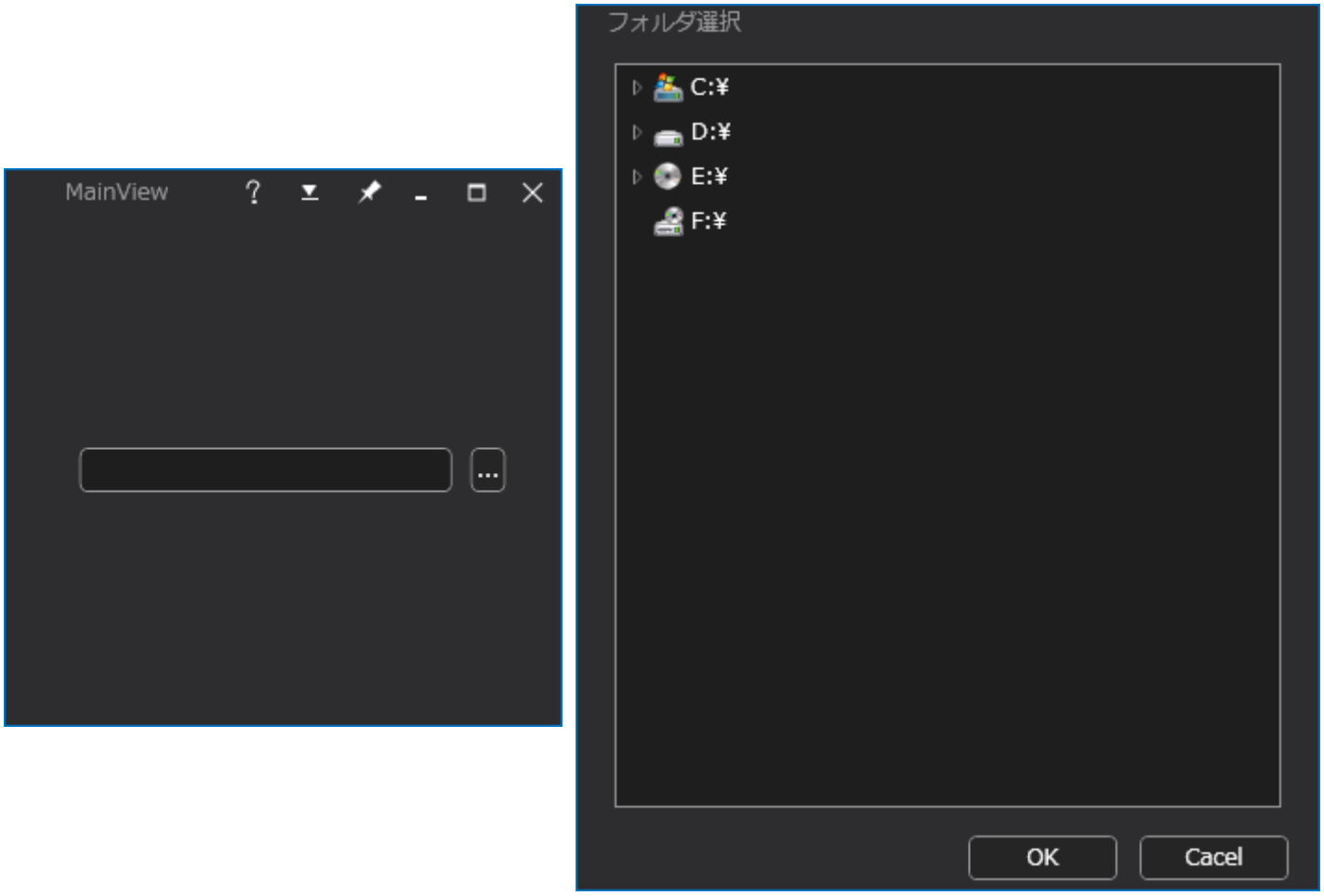
```
namespace Section4_5.ViewModels
{
    using YKToolkit.Bindings;

    public class MainViewModel : NotificationObject
    {
        private string directoryName;
        /// <summary>
        /// ディレクトリ名を取得または設定します。
        /// </summary>
```

```
11              public string DirectoryName
12              {
13                  get { return directoryName; }
14                  set { SetProperty(ref directoryName, value); }
15              }
16          }
17      }
```

DirectorySelectDialog コントロールは Window クラスから派生したコントロールであるため、XAML に直接記述するものではなく、子ウィンドウとして呼び出して使います。上記のサンプルでは、YKToolkit.Controls.Behavior 名前空間に用意されている DirectorySelectDialogBehavior 添付ビヘイビアを使用することで、ビヘイビア内部から呼び出して使用しています。

DirectorySelectDialogBehavior は RoutedEvent、Title、DirectoryName という 3 つのプロパティを持っています。RoutedEvent プロパティで指定されたルートイベントが発生すると、Title プロパティで指定された文字列をタイトルとして持つ DirectorySelectDialog コントロールを呼び出します。呼び出された DirectorySelectDialog で OK ボタンが押されたときのみ DirectorySelectDialogBehavior の DirectoryName プロパティが変更されるようになっています。

サンプルコードでは、DirectorySelectDialog コントロールの OK ボタンが押されると、MainView にある TextBox コントロールに選択されたディレクトリのフルパスが入力されるようになっています。




(a) MainView の外観 (b) DirectorySelectDialog コントロールの外観

**図 4.4：DirectorySelectDialog コントロールのサンプル画面**

## 4.6 DropDownButton

DropDownButton コントロールは、形状は Button コントロールですが、押すと内部コンテンツが下部に表示される特殊なボタンです。例えば ColorPicker コントロールと併用することで色選択をおこなうドロップダウンリストコントロールを作成できます。

以下に DropDownButton コントロールを使用したサンプルコードを示します。

**コード 4.9：DropDownButton コントロールのサンプルコード**

**MainView.xaml**

```
<YK:Window x:Class="Section4_6.Views.MainView"
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
           Title="MainView" Height="300" Width="300">
    <YK:Window.Resources>
        <!-- 透明色用背景 -->
        <DrawingBrush x:Key="AlphaBackgroundBrush" ViewportUnits="Absolute" Viewport="0,0,10,10" TileMode="Tile">
            <DrawingBrush.Drawing>
                <DrawingGroup>
                    <GeometryDrawing Brush="White">
                        <GeometryDrawing.Geometry>
                            <RectangleGeometry Rect="0,0,100,100" />
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                    <GeometryDrawing Brush="LightGray">
                        <GeometryDrawing.Geometry>
                            <GeometryGroup>
                                <RectangleGeometry Rect="0,0,50,50" />
                                <RectangleGeometry Rect="50,50,50,50" />
                            </GeometryGroup>
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                </DrawingGroup>
            </DrawingBrush.Drawing>
        </DrawingBrush>
    </YK:Window.Resources>

    <StackPanel>
        <TextBlock Text="色選択 :" />
        <YK:DropDownButton Width="56" CloseTriggerValue="{Binding SelectedColor, ElementName=colorPicker}">
            <YK:DropDownButton.ButtonContent>
                <Border Background="{StaticResource AlphaBackgroundBrush}" HorizontalAlignment="Center" VerticalAlignment="Center">
                    <Rectangle Width="16" Height="16">
                        <Rectangle.Stroke>
                            <SolidColorBrush Color="{DynamicResource BorderColor}" />
                        </Rectangle.Stroke>
                        <Rectangle.Fill>
                            <SolidColorBrush Color="{Binding SelectedColor, ElementName=colorPicker}" />
                        </Rectangle.Fill>
                    </Rectangle>
                </Border>
            </YK:DropDownButton.ButtonContent>
            <YK:ColorPicker x:Name="colorPicker" RecentColorsVisibility="Visible" />
        </YK:DropDownButton>

        <Separator Margin="0,20" />
        <TextBlock Text="任意のコントロールを配置 :" />
        <YK:DropDownButton>
            <YK:DropDownButton.ButtonContent>
                <TextBlock TextAlignment="Center">
```

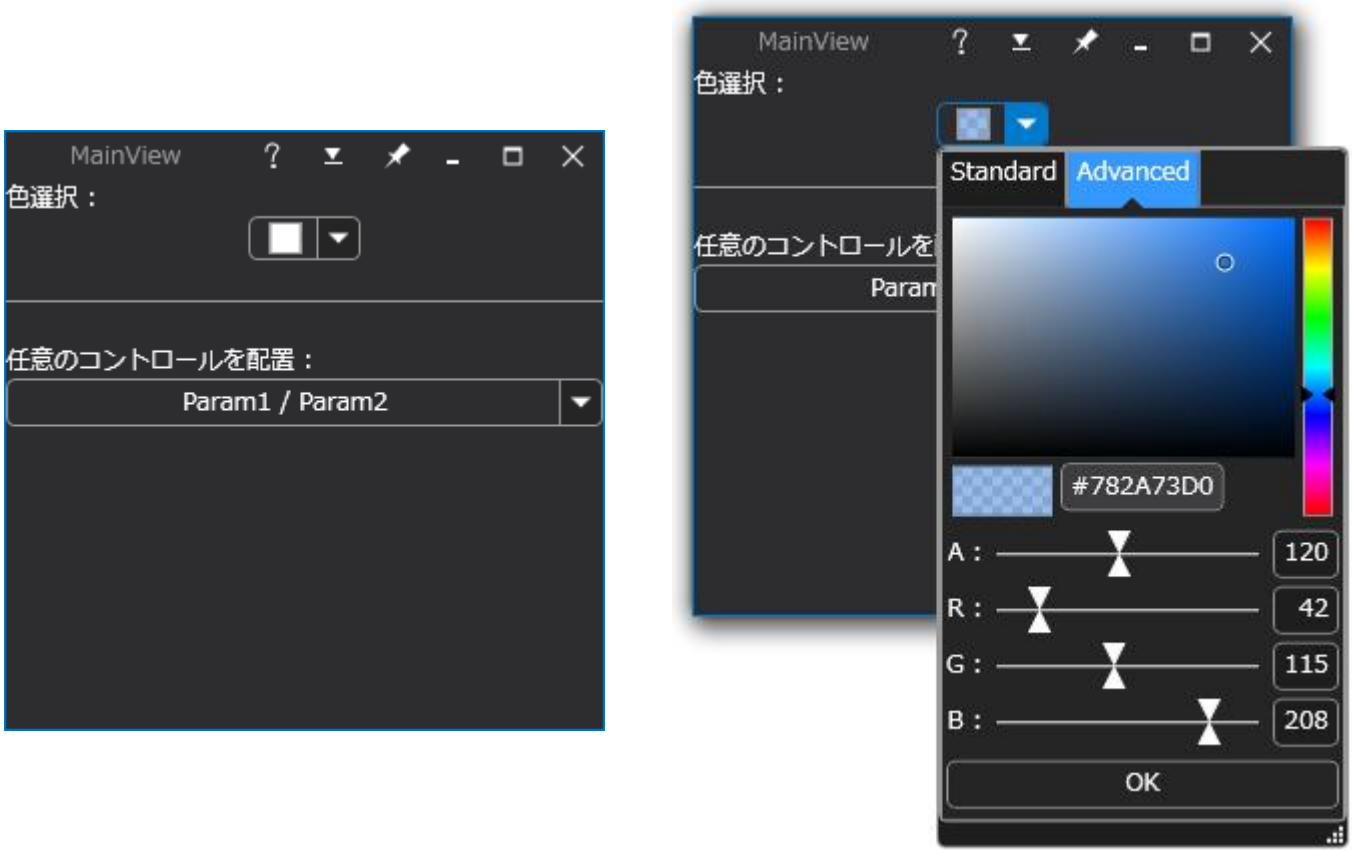
```
52                          <Run Text="{Binding Param1, Mode=OneWay}" />
53                          <Run Text="/" />
54                          <Run Text="{Binding Param2, Mode=OneWay}" />
55                      </TextBlock>
56                  </YK:DropDownButton.ButtonContent>
57
58                  <StackPanel Margin="10,10,0,0">
59                      <StackPanel Orientation="Horizontal">
60                          <TextBlock Text="Param1 : " VerticalAlignment="Center" />
61                          <TextBox Text="{Binding Param1}" MinWidth="200" />
62                      </StackPanel>
63                      <StackPanel Orientation="Horizontal" Margin="0,10,0,0">
64                          <TextBlock Text="Param2 : " VerticalAlignment="Center" />
65                          <TextBox Text="{Binding Param2}" MinWidth="200" />
66                      </StackPanel>
67                  </StackPanel>
68              </YK:DropDownButton>
69          </StackPanel>
70      </YK:Window>
```

**コード 4.10 : DropDownButton コントロールのための ViewModel のサンプルコード**

**MainViewModel.cs**

```
1   namespace Section4_6.ViewModels
2   {
3       using YKToolkit.Bindings;
4
5       public class MainViewModel : NotificationObject
6       {
7           private string param1 = "Param1";
8           /// <summary>
9           /// パラメータ 1 を取得または設定します。
10          /// </summary>
11          public string Param1
12          {
13              get { return param1; }
14              set { SetProperty(ref param1, value); }
15          }
16
17          private string param2 = "Param2";
18          /// <summary>
19          /// パラメータ 2 を取得または設定します。
20          /// </summary>
21          public string Param2
22          {
23              get { return param2; }
24              set { SetProperty(ref param2, value); }
25          }
26      }
27  }
```

DropDownButton コントロールは、ButtonContent プロパティにボタン内部のコンテンツを、Content プロパティにドロップダウン表示されるコンテンツを指定します。ドロップダウン表示されるコンテンツが DropDownButton コントロールの子供としてツリーにぶら下がるイメージになるため、コードとして非常にスッキリします。




(a) MainView の外観 (b) ColorPicker を内部に持つ DropDownButton コントロール

**図 4.5：DropDownButton コントロールのサンプル画面**

## 4.7 FileTreeView

FileTreeView コントロールは、ローカルのファイル構成をツリー形式で閲覧または選択できるコントロールです。選択したファイルはドラッグ＆ドロップすることもできます。

FileTreeView コントロールを使用したサンプルアプリケーションの外観を以下に示します。画面は大きく分けて 3 行の構成となっており、1 行目は FileTreeView コントロールに対する設定、2 行目は FileTreeView コントロールと選択されたディレクトリ内のファイルを示すための ListBox コントロール、3 行目は選択されたディレクトリまたはファイルの各種情報が表示されています。このサンプルコードは非常に長いため、その一部を次に示します。

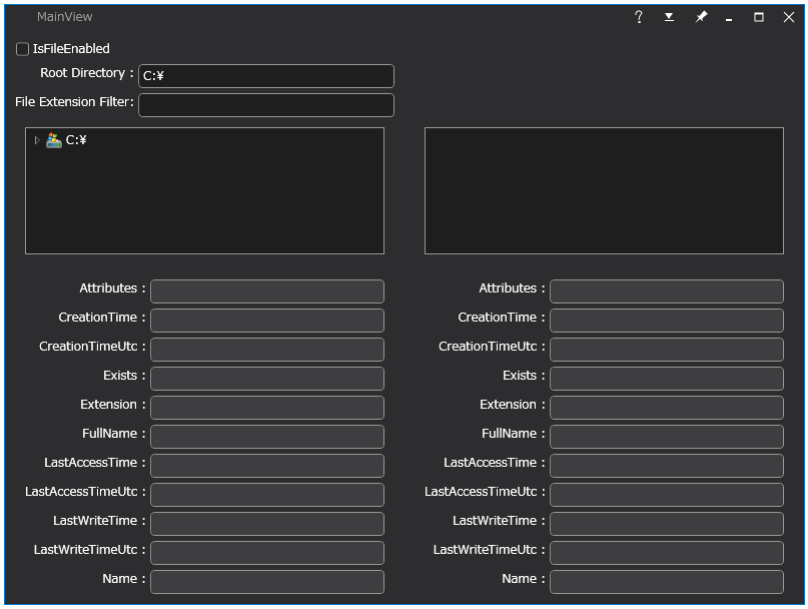



図 4.6：FileTreeView コントロールのサンプルアプリケーション

コード 4.11：FileTreeView コントロールのサンプルコードの一部

**MainView.xaml**

```
49          <YK:FileTreeView Grid.Row="1"
50                           x:Name="FileTreeView"
51                           Margin="20,10,20,10"
52                           IsFileEnabled="{Binding IsChecked, ElementName=IsFileEnabledCheckBox}"
53                           RootNode="{Binding Text, ElementName=RootNodeTextBox}"
54                           ExtensionFilter="{Binding Text, ElementName=ExtensionFilterTextBox}"
55                           />
```

FileTreeView コントロールには、IsFileEnabled プロパティ、RootNode プロパティ、ExtensionFilter プロパティがあります。IsFileEnabled プロパティは、FileTreeView コントロール上にディレクトリのみを表示するか、ファイルも表示するかを設定できます。RootNode プロパティは、FileTreeView コントロールのルートノードのフルパスを設定できます。null または Empty の場合はすべてのドライブが表示されます。ExtensionFiliter プロパティは、表示するファイルの種類を拡張子でフィルタリングするためのプロパティです。これは FileTreeView コントロール内に表示するファイルだけでなく、選択されたディレクトリに対するファイルリストにも有効となります。

さらに、FileTreeView コントロールは、選択されたアイテムをドラッグ＆ドロップすることができます。ドラッグされたオブジェクトは YKToolkit.Controls.FileTreeNode クラスとなります。これを受け取るためのコードビハインドを以下に示します。

コード 4.12：FileTreeView コントロールのドラッグ＆ドロップためのコードビハインド

**MainView.xaml.cs**

```
1   namespace Section4_7.Views
2   {
3       using System.Windows.Controls;
4       using YKToolkit.Controls;
5
6       /// <summary>
7       /// MainView.xaml の相互作用ロジック
8       /// </summary>
9       public partial class MainView : Window
10      {
11          public MainView()
12          {
13              InitializeComponent();
```

```
14              }
15
16              /// <summary>
17              /// Drop イベントハンドラ
18              /// </summary>
19              /// <param name="sender"></param>
20              /// <param name="e"></param>
21              private void RootNodeTextBox_Drop(object sender, System.Windows.DragEventArgs e)
22              {
23                  var item = e.Data.GetData(typeof(TreeViewItem)) as TreeViewItem;
24                  if (item != null)
25                  {
26                      var data = item.Header as FileTreeNode;
27                      if (data != null)
28                      {
29                          // YKToolkit.Controls.FileTreeNode がドロップされたらそのフルパスを受け取る
30                          RootNodeTextBox.Text = data.NodeInfo.FullName;
31                      }
32                  }
33              }
34
35              /// <summary>
36              /// PreviewDragOver イベントハンドラ
37              /// </summary>
38              /// <param name="sender"></param>
39              /// <param name="e"></param>
40              private void RootNodeTextBox_PreviewDragOver(object sender, System.Windows.DragEventArgs e)
41              {
42                  // DragOver イベントをここで中断させる
43                  e.Handled = true;
44              }
45          }
46      }
```

また、YKToolkit.Controls.FileTreeNode クラスは System.IO,FileSystemInfo クラスである NodeInfo プロパティを持っているため、選択されたディレクトリまたはファイルの情報は NodeInfo プロパティから取得できます。サンプルアプリケーションではこの NodeInfo プロパティから様々な情報を表示しています。

## 4.8 LineGraph

LineGraph コントロールは折れ線グラフを表示するためのコントロールです。LineGraph コントロールを使用したサンプルアプリケーションの外観を以下に示します。

グラフには複数のデータを指定でき、縦軸として第 1 主軸と第 2 主軸の選択ができます。また、グラフ上で右クリックすることで表示されるメニューから様々な設定ができます。

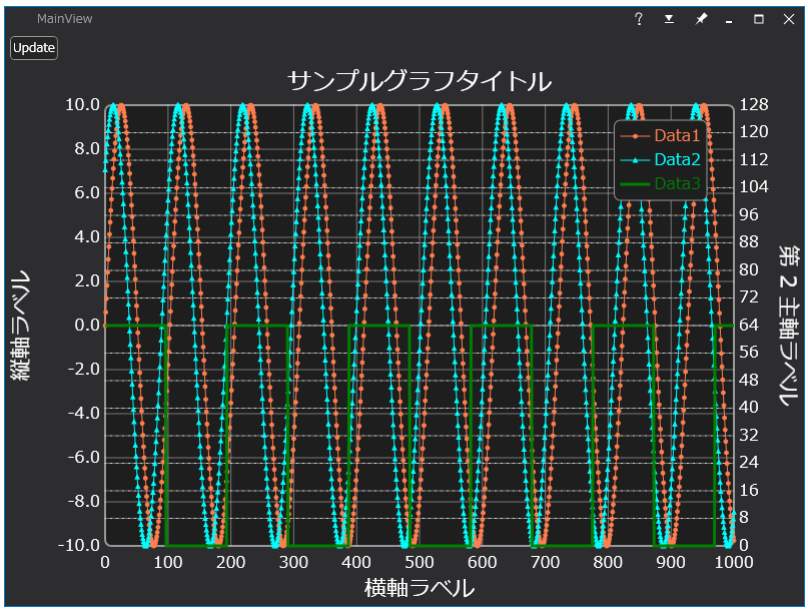
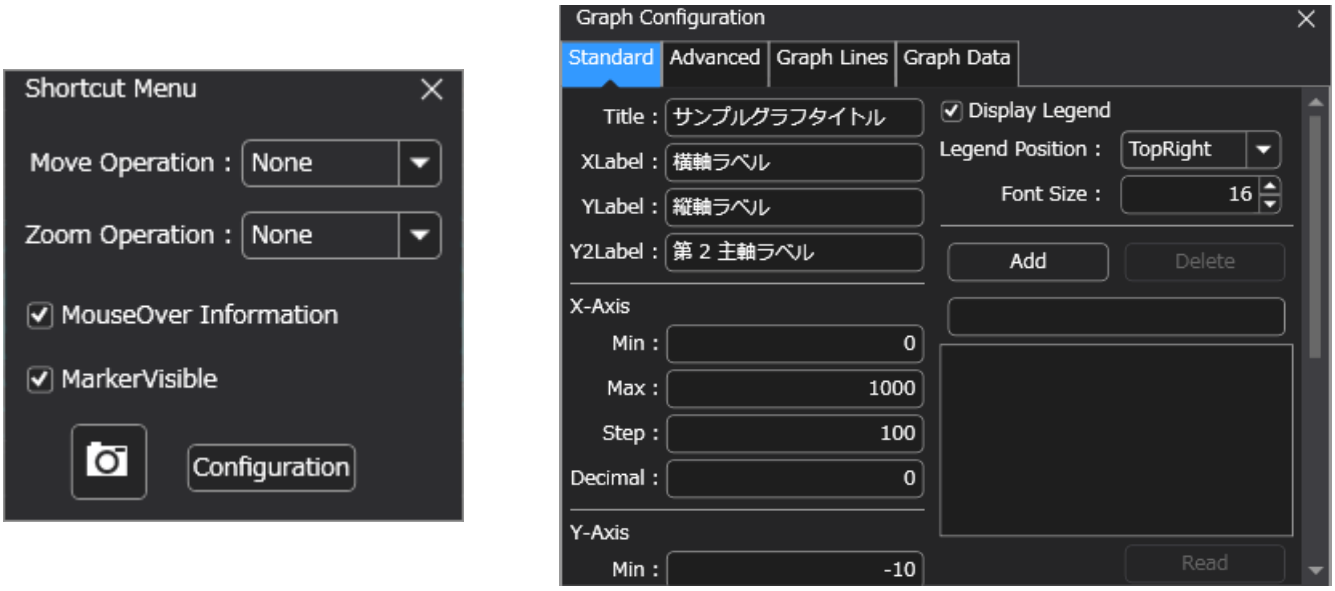



図 4.7：LineGraph コントロールのサンプルアプリケーション




(a) ショートカットメニュー (b) グラフ設定メニュー

図 4.8：LineGraph コントロールの右クリックメニュー

右クリックで表示されるショートカットメニューには、移動/拡大操作をするための "Move Operation"、"Zoom Operation" というメニューがあります。これらを "None" 以外にすると、グラフをドラッグ＆ドロップで移動したり拡大したりできるようになります。拡大操作で縮小したい場合はグラフをダブルクリックします。

"MouseOver Information" のチェックボックスを ON にすると、グラフにマウスポインタを乗せると、その横軸の位置周辺のグラフデータの数値が凡例にそれぞれ表示されるようになります。凡例が表示されていない場合は、"Configuration" ボタンを押してグラフ設定メニューを開き、"Display Legend" チェックボックスを ON にします。

"MarkerVisible" チェックボックスを OFF にすると、すべてのグラフデータのマーカーが非表示となります。一部のグラフデータのみマーカーを非表示にしたい場合は、サンプルプログラムでもおこなっているように、グラフデータである YKToolkit.Controls.LineGraphItem クラスの IsMarkerEnabled プロパティを false にして下さい。

グラフ設定メニューでは、各軸目盛の設定や、線種の設定などがおこなえるようになっています。また、"Graph Data" タブでは、各グラフデータの数値を表形式で閲覧することができます。

サンプルプログラムのコードを以下に示します。

コード 4.13：LineGraph コントロールのサンプルコード

MainView.xaml

```
<YK:Window x:Class="Section4_8.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
        Title="MainView"
        Width="800" Height="600"
        WindowStartupLocation="CenterScreen">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <Button Content="Update" Command="{Binding UpdateCommand}" HorizontalAlignment="Left" Margin="5" />

        <YK:LineGraph Grid.Row="1"
                      Title="サンプルグラフタイトル"
                      XLabel="横軸ラベル"
                      YLabel="縦軸ラベル"
                      Y2Label="第 2 主軸ラベル"
                      XMin="0" XMax="100" XStep="10" XDecimal="0"
                      YMin="-10" YMax="10" YStep="2" YDecimal="1"
                      Y2Min="0" Y2Max="128" Y2Step="8" Y2Decimal="0"
                      IsSecondEnabled="True"
                      IsLegendEnabled="True"
                      IsMouseOverInformationEnabled="True">
            <YK:LineGraph.ItemsSource>
                <x:Array Type="{x:Type YK:LineGraphItem}">
                    <YK:LineGraphItem Title="Data1" Stroke="Coral" MarkerType="Ellipse" Points="{Binding Data1}" />
                    <YK:LineGraphItem Title="Data2" Stroke="Cyan" MarkerType="Triangle" Points="{Binding Data2}" />
                    <YK:LineGraphItem Title="Data3" Stroke="Green" IsMarkerEnabled="False" Points="{Binding Data3}" IsSecond="True" StrokeThickness="3" />
                </x:Array>
            </YK:LineGraph.ItemsSource>
        </YK:LineGraph>
    </Grid>
</YK:Window>
```

コード 4.14：LineGraph コントロールのための ViewModel のサンプルコード

MainViewModel.cs

```
namespace Section4_8.ViewModels
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Windows;
    using YKToolkit.Bindings;

    public class MainViewModel : NotificationObject
    {
        /// <summary>
        /// 新しいインスタンスを生成します。
```

```
13              /// </summary>
14              public MainViewModel()
15              {
16                  UpdateGraphData();
17              }
18
19              private IEnumerable<Point> data1;
20              /// <summary>
21              /// 1 つ目のサンプルデータのデータ列を取得または設定します。
22              /// </summary>
23              public IEnumerable<Point> Data1
24              {
25                  get { return data1; }
26                  set { SetProperty(ref data1, value); }
27              }
28
29              private IEnumerable<Point> data2;
30              /// <summary>
31              /// 2 つ目のサンプルデータのデータ列を取得または設定します。
32              /// </summary>
33              public IEnumerable<Point> Data2
34              {
35                  get { return data2; }
36                  set { SetProperty(ref data2, value); }
37              }
38
39              private IEnumerable<Point> data3;
40              /// <summary>
41              /// 3 つ目のサンプルデータのデータ列を取得または設定します。
42              /// </summary>
43              public IEnumerable<Point> Data3
44              {
45                  get { return data3; }
46                  set { SetProperty(ref data3, value); }
47              }
48
49              private DelegateCommand updateCommand;
50              /// <summary>
51              /// グラフデータ更新コマンドを取得します。
52              /// </summary>
53              public DelegateCommand UpdateCommand
54              {
55                  get
56                  {
57                      return updateCommand ?? (updateCommand = new DelegateCommand(_ =>
58                      {
59                          UpdateGraphData();
60                      }));
61                  }
62              }
63
64              /// <summary>
65              /// グラフデータを更新します。
66              /// </summary>
67              private void UpdateGraphData()
68              {
69                  var rnd = new Random();
70                  var N = 1001;
```

```
71                  var fMax = 15.0;
72                  double f;
73
74                  f = rnd.NextDouble() * fMax;
75                  Data1 = Enumerable.Range(0, N).Select(i =>
76                  {
77                      // f[Hz] の正弦波
78                      var x = (double)i;
79                      var y = 10.0 * Math.Sin(2.0 * Math.PI * f * x / 1000.0);
80                      return new Point(x, y);
81                  });
82
83                  f = rnd.NextDouble() * fMax;
84                  Data2 = Enumerable.Range(0, N).Select(i =>
85                  {
86                      // f[Hz] で π/4 だけ位相がずれた正弦波
87                      var x = (double)i;
88                      var y = 10.0 * Math.Sin(2.0 * Math.PI * f * x / 1000.0 + Math.PI / 4.0);
89                      return new Point(x, y);
90                  });
91
92                  f = rnd.NextDouble() * fMax;
93                  double d = 0.0;
94                  Data3 = Enumerable.Range(0, N).Select(i =>
95                  {
96                      // 矩形波
97                      if (i % (int)(f * 10.0) == 0)
98                          d = d > 0.0 ? 0.0 : 64.0;
99                      var x = (double)i;
100                     var y = d;
101                     return new Point(x, y);
102                 });
103             }
104     }
105 }
```

## 4.9 MessageBox

MessageBox コントロールは、メッセージを表示し、それに対するユーザのレスポンスを取得するダイアログです。したがって、ビヘイビアなどでよく使われます。

MessageBox コントロールを使用したサンプルコードを以下に示します。理想的な MVVM パターンでは、ViewModel から View を呼び出すことはマナー違反ですが、簡単なメッセージを出力するのに一々複雑なビヘイビアなどを作成することは煩わしいため、サンプルコードのように ViewModel から直接 MessageBox コントロールを呼び出すこともあります。

**コード 4.15：MessageBox コントロールのサンプルコード**

**MainView.xaml**

```
1  <YK:Window x:Class="Section4_9.Views.MainView"
2              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4              xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5              Title="MainView" Height="300" Width="300">
6      <StackPanel>
7          <Button Content="Ok or Cancel" Command="{Binding OkCancelCommand}" />
8          <TextBlock Text="{Binding OkCancelText}" />
9          <Button Content="Yes, No or Cancel" Command="{Binding YesNoCancelCommand}" />
10         <TextBlock Text="{Binding YesNoCancelText}" />
11     </StackPanel>
```

```
12  </YK:Window>
```

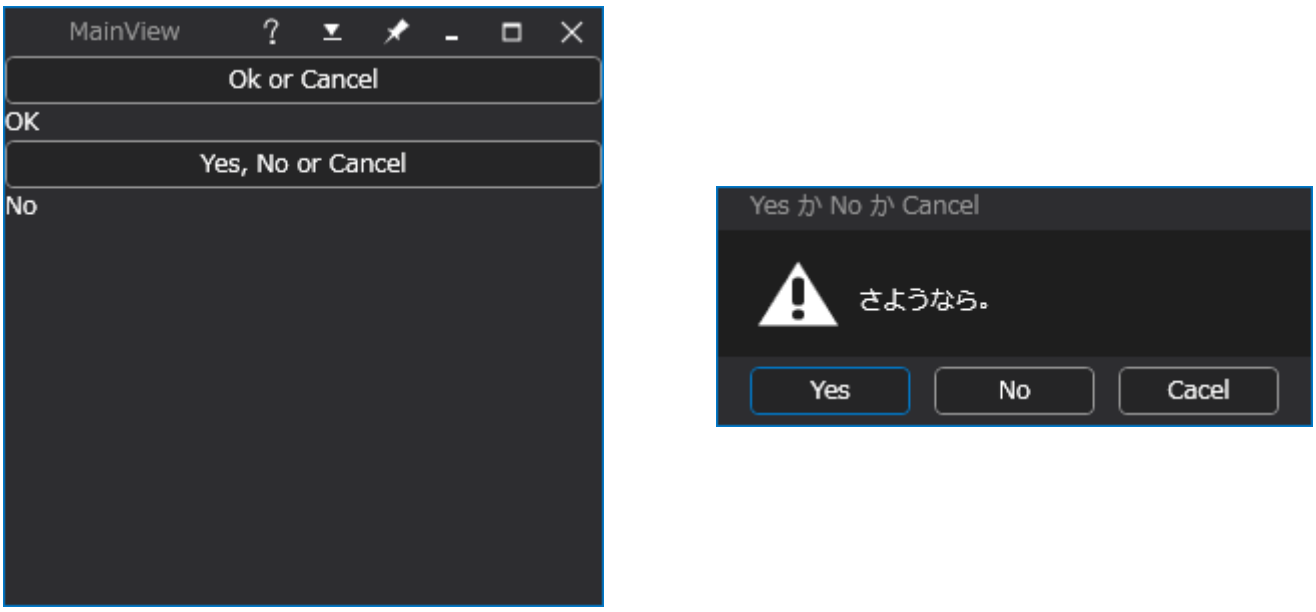
コード 4.16：Message コントロールのための ViewModel のサンプルコード

MainViewModel.cs

```
1   namespace Section4_9.ViewModels
2   {
3       using YKToolkit.Bindings;
4       using YKToolkit.Controls;
5
6       public class MainViewModel : NotificationObject
7       {
8           private string okCancelText;
9           /// <summary>
10          /// OkCancel メッセージボックスの結果を取得します。
11          /// </summary>
12          public string OkCancelText
13          {
14              get { return okCancelText ?? "null"; }
15              private set { SetProperty(ref okCancelText, value); }
16          }
17
18          private string yesNoCancelText;
19          /// <summary>
20          /// YesNoCancel メッセージボックスの結果を取得します。
21          /// </summary>
22          public string YesNoCancelText
23          {
24              get { return yesNoCancelText ?? "null"; }
25              private set { SetProperty(ref yesNoCancelText, value); }
26          }
27
28          private DelegateCommand okCancelCommand;
29          /// <summary>
30          /// Ok/Cancel メッセージボックス表示コマンドを取得します。
31          /// </summary>
32          public DelegateCommand OkCancelCommand
33          {
34              get
35              {
36                  return okCancelCommand ?? (okCancelCommand = new DelegateCommand(_ =>
37                  {
38                      // ViewModel から Window クラスを呼び出すのは
39                      // MVVM 的にはあまり美しいコードではないが
40                      // 現実的にはこれが一番楽
41                      var result = MessageBox.Show("こんにちは。", "OK か Cancel",
    MessageBoxButton.OKCancel, MessageBoxImage.Question);
42                      OkCancelText = result.ToString();
43                  }));
44              }
45          }
46
47          private DelegateCommand yesNoCancelCommand;
48          /// <summary>
49          /// Yes/No/Cancel メッセージボックス表示コマンドを取得します。
50          /// </summary>
51          public DelegateCommand YesNoCancelCommand
52          {
53              get
```

```
54                  {
55                      return yesNoCancelCommand ?? (yesNoCancelCommand = new DelegateCommand(_ =>
56                      {
57                          // ViewModel から Window クラスを呼び出すのは
58                          // MVVM 的にはあまり美しいコードではないが
59                          // 現実的にはこれが一番楽
60                          var result = MessageBox.Show("さようなら。", "Yes か No か Cancel",
    MessageBoxButton.YesNoCancel, MessageBoxImage.Warning);
61                          YesNoCancelText = result.ToString();
62                      }));
63                  }
64              }
65          }
66      }
```

MessageBox クラスには Show() 静的メソッドがあるため、これを使用してダイアログを表示します。また、結果として YKToolkit.Controls.MessageBoxResult 列挙体が返ってくるため、これを判定して次の処理につなげることができます。サンプルコードでは返ってきた結果をそのまま string 型に変換して表示しています。




(a) メイン画面 (b) MessageBox コントロール

**図 4.9：MessageBox コントロールのサンプルアプリケーション**

## 4.10 SpinInput

SpinInput コントロールは、スピンボタン付きの数値入力用コントロールです。スピンボタンを押すことで数値を決められた間隔で変更することができます。

SpinInput コントロールを使用したサンプルコードを以下に示します。

**コード 4.17：SpinInput コントロールのサンプルコード**

**MainView.xaml**

```
1   <YK:Window x:Class="Section4_10.Views.MainView"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4               xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5               Title="MainView" Height="300" Width="400">
6       <Grid>
7           <Grid.Resources>
8               <Style TargetType="{x:Type TextBlock}">
```

```
                    <Setter Property="VerticalAlignment" Value="Center" />
                    <Setter Property="TextAlignment" Value="Right" />
                </Style>
                <Style TargetType="{x:Type YK:SpinInput}" BasedOn="{StaticResource {x:Type YK:SpinInput}}">
                    <Setter Property="Margin" Value="5" />
                </Style>
            </Grid.Resources>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition />
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>

            <TextBlock Grid.Row="0" Grid.Column="0" Text="Min : " />
            <TextBlock Grid.Row="1" Grid.Column="0" Text="Max : " />
            <TextBlock Grid.Row="2" Grid.Column="0" Text="Decimal : " />
            <TextBlock Grid.Row="0" Grid.Column="2" Text="Tick : " />
            <TextBlock Grid.Row="1" Grid.Column="2" Text="Delay : " />
            <TextBlock Grid.Row="2" Grid.Column="2" Text="Interval : " />

            <YK:SpinInput Grid.Row="0" Grid.Column="1" x:Name="Min" />
            <YK:SpinInput Grid.Row="1" Grid.Column="1" x:Name="Max" Value="100" />
            <YK:SpinInput Grid.Row="2" Grid.Column="1" x:Name="Decimal" Min="0" />
            <YK:SpinInput Grid.Row="0" Grid.Column="3" x:Name="Tick" Value="1" />
            <YK:SpinInput Grid.Row="1" Grid.Column="3" x:Name="Delay" Min="0" Value="500" />
            <YK:SpinInput Grid.Row="2" Grid.Column="3" x:Name="Interval" Min="0" Value="100" />

            <TextBlock Grid.Row="3" Text="FontSize : " />
            <YK:SpinInput Grid.Row="3" Grid.Column="1" Grid.ColumnSpan="3"
                          x:Name="input"
                          Min="{Binding Value, ElementName=Min}"
                          Max="{Binding Value, ElementName=Max}"
                          Decimal="{Binding Value, ElementName=Decimal}"
                          Tick="{Binding Value, ElementName=Tick}"
                          Delay="{Binding Value, ElementName=Delay}"
                          Interval="{Binding Value, ElementName=Interval}"
                          Value="{Binding Input}"
                          />

            <TextBlock Grid.Row="4" Grid.ColumnSpan="4"
                       Text="文字の大きさが変わります。"
                       TextAlignment="Left"
                       FontSize="{Binding Value, ElementName=input}"
                       />
        </Grid>
    </YK:Window>
```


図 4.10：SpinInput コントロールのサンプルアプリケーション

## 4.11 TextBox

TextBox コントロールは、標準の System.Windows.Controls.TextBox コントロールに加え、ウォーターマークを表示できるコントロールです。

TextBox コントロールを使用したサンプルコードを以下に示します。

コード 4.18：TextBox コントロールのサンプルコード

**MainView.xaml**

```
1  <YK:Window x:Class="Section4_11.Views.MainView"
2              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4              xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5              Title="MainView" Height="300" Width="300">
6      <Grid VerticalAlignment="Center">
7          <Grid.RowDefinitions>
8              <RowDefinition Height="Auto" />
9              <RowDefinition Height="10" />
10             <RowDefinition Height="Auto" />
11             <RowDefinition Height="10" />
12             <RowDefinition Height="Auto" />
13         </Grid.RowDefinitions>
14
15         <YK:TextBox Watermark="Input somthing." />
16         <YK:TextBox Grid.Row="2">
17             <YK:TextBox.Watermark>
18                 <StackPanel Orientation="Horizontal">
19                     <Ellipse Fill="Red" Width="10" Height="10" Margin="4,0" />
20                     <TextBlock Text="未入力">
21                         <TextBlock.Foreground>
22                             <SolidColorBrush Color="{DynamicResource DisabledSymbolColor}" />
23                         </TextBlock.Foreground>
24                     </TextBlock>
25                 </StackPanel>
26             </YK:TextBox.Watermark>
27         </YK:TextBox>
28         <YK:TextBox Grid.Row="4" Watermark="入力して下さい。">
```

```
29                  <YK:TextBox.WatermarkTemplate>
30                      <DataTemplate>
31                          <Border BorderBrush="Green" BorderThickness="1">
32                              <TextBlock Text="{Binding}" />
33                          </Border>
34                      </DataTemplate>
35                  </YK:TextBox.WatermarkTemplate>
36              </YK:TextBox>
37          </Grid>
38      </YK:Window>
```

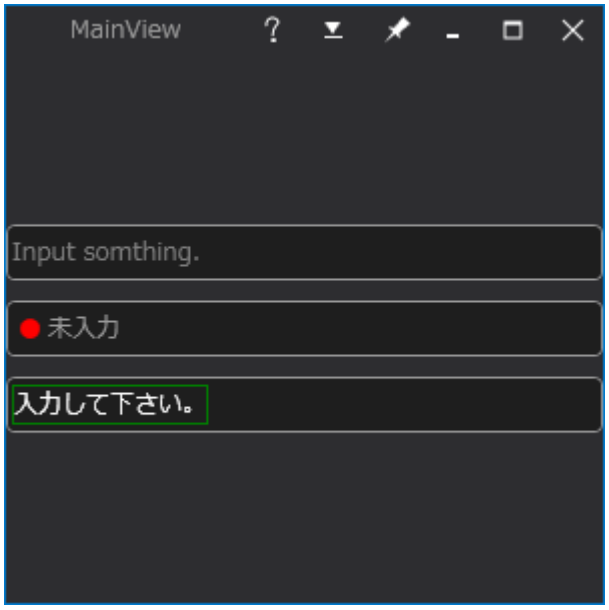



図 4.11：TextBox コントロールのサンプルアプリケーション

## 4.12 Transition

Transition コントロールは、画面遷移アニメーションを実現するためのコントロールです。複数のユーザコントロールをアニメーションとともに入れ替えることができます。表示結果は TransitionControl と同じですが、その内部構造の違いから、コンテンツの指定方法が異なります。

Transition コントロールは、Source プロパティに指定されたオブジェクトを表示するコントロールです。ただし、Source プロパティには必ず ITransitionListItem インターフェースが実装されたクラスを割り当てる必要があります。ITransitionListItem インターフェースは次表に示すプロパティを持っています。Name プロパティに画面名、Type プロパティに表示するオブジェクトの System.Type 情報を設定する必要があります。

表 4.2：ITransitionListItem インターフェース

| プロパティ名 | 概　要 |
|---|---|
| Name | 表示する名前を取得または設定します。 |
| Type | object の Type 情報を取得または設定します。 |

以下に Transition コントロールを使用したサンプルコードを示します。

コード 4.19：Transition コントロールのサンプルコード

**MainView.xaml**

```
1   <YK:Window x:Class="Section4_12.Views.MainView"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4               xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5               xmlns:vw="clr-namespace:Section4_12.Views"
```

```
6               Title="MainView" Height="300" Width="300">
7       <Grid>
8           <Grid.RowDefinitions>
9               <RowDefinition Height="Auto" />
10              <RowDefinition />
11          </Grid.RowDefinitions>
12
13          <StackPanel>
14              <ComboBox x:Name="ScreenSelector" SelectedIndex="0">
15                  <ComboBox.ItemsSource>
16                      <x:Array Type="{x:Type YK:TransitionListItem}">
17                          <YK:TransitionListItem Name="Page1" Type="{x:Type vw:Page1View}" />
18                          <YK:TransitionListItem Name="Page2" Type="{x:Type vw:Page2View}" />
19                          <YK:TransitionListItem Name="Page3" Type="{x:Type vw:Page3View}" />
20                      </x:Array>
21                  </ComboBox.ItemsSource>
22                  <ComboBox.ItemTemplate>
23                      <DataTemplate>
24                          <TextBlock Text="{Binding Name}" />
25                      </DataTemplate>
26                  </ComboBox.ItemTemplate>
27              </ComboBox>
28
29              <ComboBox x:Name="direction" SelectedIndex="0">
30                  <ComboBox.ItemsSource>
31                      <x:Array Type="{x:Type YK:TransitionDirections}">
32                          <YK:TransitionDirections>ToLeft</YK:TransitionDirections>
33                          <YK:TransitionDirections>ToTop</YK:TransitionDirections>
34                          <YK:TransitionDirections>ToRight</YK:TransitionDirections>
35                          <YK:TransitionDirections>ToBottom</YK:TransitionDirections>
36                      </x:Array>
37                  </ComboBox.ItemsSource>
38              </ComboBox>
39          </StackPanel>
40
41          <YK:Transition Grid.Row="1"
42                         Source="{Binding SelectedItem, ElementName=ScreenSelector}"
43                         Direction="{Binding SelectedItem, ElementName=direction}"
44                         />
45      </Grid>
46  </YK:Window>
```

1 つ目の ComboBox コントロールで ITransitionListItem インターフェースを実装した TransitionListItem クラスが選択できるようになっています。それぞれのアイテムには Name プロパティと Type プロパティを設定しています。Type プロパティで設定している "vw:Page*View" というオブジェクトは、あらかじめ作成してある Page*View という名前のユーザコントロールです。この ComboBox コントロールで選択されたアイテムが Transition コントロールの Source プロパティに設定されるようにデータバインドされています。

また、Transition コントロールにはアニメーション方向を指定するための Direction プロパティがあります。こちらの設定は 2 つ目の ComboBox コントロールで選択できるようにしています。

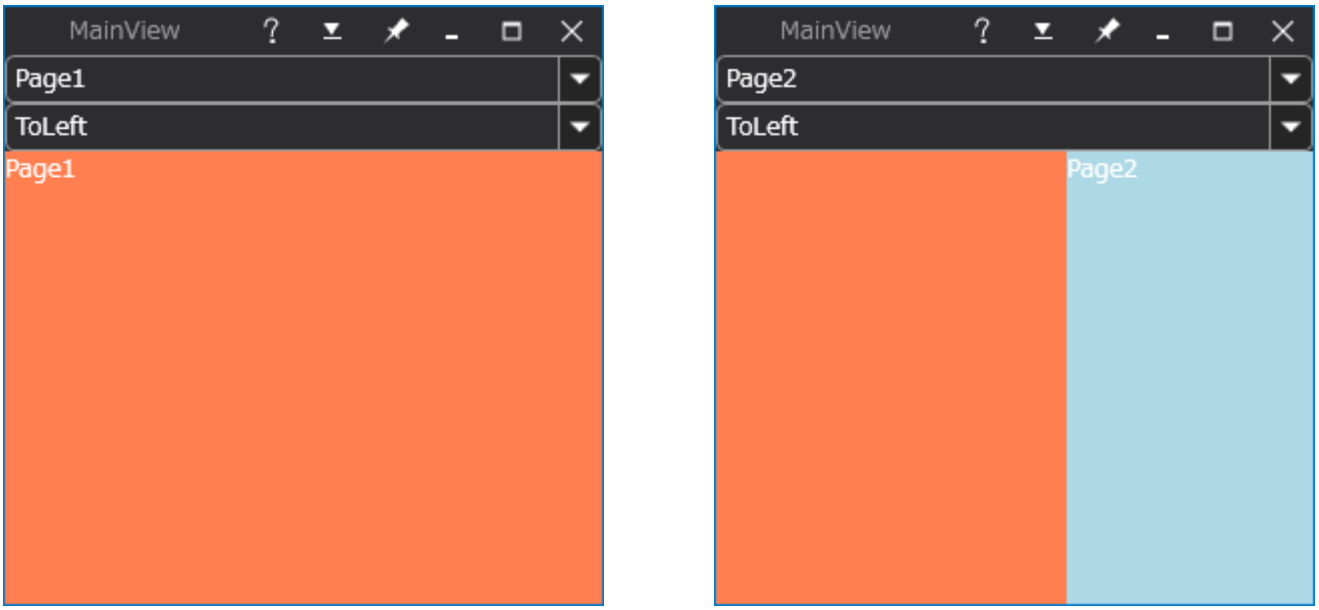
ここで、Page1View のコードについて見てみます。UserControl なので、必要が無い限り YKToolkit.Controls 名前空間を定義する必要はありません。また、DataContext プロパティに ViewModel を割り付ける必要がありますが、ここでは XAML 上で対応する Page1ViewModel を割り付けています。

**コード 4.20：Transition コントロールで遷移される画面のサンプルコード**

**Page1View.xaml**

```
1   <UserControl x:Class="Section4_12.Views.Page1View"
2                xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
3                   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
5                   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
6                   mc:Ignorable="d"
7                   d:DesignHeight="300" d:DesignWidth="300"
8                   xmlns:vm="clr-namespace:Section4_12.ViewModels"
9                   Background="Coral">
10      <UserControl.DataContext>
11          <vm:Page1ViewModel />
12      </UserControl.DataContext>
13
14      <Grid>
15          <TextBlock Text="{Binding Title}" />
16      </Grid>
17  </UserControl>
```




(a) 起動時の画面 (b) 画面遷移中

**図 4.12：Transition コントロールのサンプルアプリケーション**

## 4.13 TransitionControl

TransitionControl コントロールは、画面遷移アニメーションを実現するためのコントロールです。表示結果は Transition コントロールと同じですが、その内部構造の違いから、コンテンツの指定方法が異なります。

TransitionControl コントロールは、Content プロパティに指定されたオブジェクトを画面に表示するコントロールで、Content プロパティが変更されるタイミングで画面遷移アニメーションが実行されます。以下に TransitionControl コントロールを使用したサンプルコードを示します。

**コード 4.21：TransitionControl コントロールのサンプルコード**

**MainView.xaml**

```
1   <YK:Window x:Class="Section4_13.Views.MainView"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4               xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5               xmlns:vw="clr-namespace:Section4_13.Views"
6               xmlns:vm="clr-namespace:Section4_13.ViewModels"
7               Title="MainView" Height="300" Width="300">
```

```
8       <Grid>
9           <Grid.RowDefinitions>
10              <RowDefinition Height="24" />
11              <RowDefinition />
12          </Grid.RowDefinitions>
13
14          <ComboBox x:Name="ScreenSelector" SelectedIndex="0">
15              <ComboBox.ItemsSource>
16                  <x:Array Type="{x:Type UserControl}">
17                      <vw:Page1View>
18                          <vw:Page1View.DataContext>
19                              <vm:Page1ViewModel />
20                          </vw:Page1View.DataContext>
21                      </vw:Page1View>
22                      <vw:Page2View>
23                          <vw:Page2View.DataContext>
24                              <vm:Page2ViewModel />
25                          </vw:Page2View.DataContext>
26                      </vw:Page2View>
27                      <vw:Page3View>
28                          <vw:Page3View.DataContext>
29                              <vm:Page3ViewModel />
30                          </vw:Page3View.DataContext>
31                      </vw:Page3View>
32                  </x:Array>
33              </ComboBox.ItemsSource>
34              <ComboBox.ItemTemplate>
35                  <DataTemplate>
36                      <TextBlock Text="{Binding Name}" />
37                  </DataTemplate>
38              </ComboBox.ItemTemplate>
39          </ComboBox>
40
41          <Border Grid.Row="1">
42              <YK:TransitionControl Content="{Binding SelectedItem, ElementName=ScreenSelector}" />
43          </Border>
44      </Grid>
45  </YK:Window>
```

サンプルコードでは、ComboBox で TrnsitionControl コントロールの Content プロパティに対するオブジェクトを指定させています。TransitionControl コントロールでは、Content にインスタンスを指定しないといけないため、ComboBox のアイテムにはオブジェクトインスタンスを並べています。

## 4.14 サンプルプロジェクト

| ソリューション名 | プロジェクト名 | 概　要 |
|---|---|---|
| Section4 | Section4_2 | BarGraph コントロールに関するサンプルプログラム |
| | Section4_3 | BusyIndicator コントロールに関するサンプルプログラム |
| | Section4_4 | ColorPicker コントロールに関するサンプルプログラム |
| | Section4_5 | DirectorySelectDialog コントロールに関するサンプルプログラム |
| | Section4_6 | DropDownButton コントロールに関するサンプルプログラム |
| | Section4_7 | FileTreeView コントロールに関するサンプルプログラム |
| | Section4_8 | LineGraph コントロールに関するサンプルプログラム |
| | Section4_9 | MessageBox コントロールに関するサンプルプログラム |
| | Section4_10 | SpinInput コントロールに関するサンプルプログラム |
| | Section4_11 | TextBox コントロールに関するサンプルプログラム |
| | Section4_12 | Transition コントロールに関するサンプルプログラム |
| | Section4_13 | TransitionControl コントロールに関するサンプルプログラム |

# 5 添付ビヘイビア

この章では、YKToolkit.Controls.dll で公開されている、添付ビヘイビアを紹介します。詳細については YKToolkit 付属のヘルプドキュメントファイルをご参照ください。

## 5.1 概要

YKToolkit.Controls.dll では、よく使う機能を添付ビヘイビアとして YKToolkit.Controls.Behaviors 名前空間で公開しています。下表に添付ビヘイビアの一覧を掲載します。

**表 5.1：添付ビヘイビアー覧表**

| 添付ビヘイビア名 | 概　要 |
|---|---|
| CommonDialogBehavior | ファイルを開くまたは保存するためのコモンダイアログを表示するための添付ビヘイビアです。 |
| DataGridBehavior | DataGrid コントロールの行ヘッダに行番号を付加するための添付ビヘイビアです。 |
| DirectorySelectDialogBehavior | DirectorySelectDialog コントロールを表示するための添付ビヘイビアです。 |
| DragBehavior | 任意のコントロールをドラッグ操作できるようにするための添付ビヘイビアです。FileTreeView コントロールにも使用されています。 |
| FileDropBehavior | ファイルドロップイベントのコールバック関数をデータバインドできるようにするための添付ビヘイビアです。 |
| KeyDownBehavior | ある特定のキーに対してコマンドを割り当てることができる添付ビヘイビアです。 |
| RoutedEventTriggerBehavior | 指定された RoutedEvent に対してコマンドを割り当てることができる添付ビヘイビアです。 |
| SystemMenuBehavior | ウィンドウコントロールのシステムメニューの表示や、Alt+F4 によるアプリケーションの終了をおこなわないようにするための添付ビヘイビアです。 |
| TextBoxGotFocusBehavior | TextBox コントロール選択時に、既に入力されているテキスト全体を選択状態にするための添付ビヘイビアです。 |
| WriteBitmapBehavior | コントロールをビットマップ画像として保存するための添付ビヘイビアです。 |

## 5.2 CommonDialogBehavior

コモンダイアログを表示させ、ファイルの読み書きをおこなうための添付ビヘイビアです。ファイルの読み書きのためのコールバックを ViewModel 側に記述できるという利点があります。CommonDialogBehavior 添付ビヘイビアには次のようなプロパティがあります。

**表 5.2：CommonDialogBehavior 添付ビヘイビアのプロパティ**

| プロパティ名 | 概　要 |
|---|---|
| Callback | コモンダイアログの結果を返す先のコールバックを指定します。 |
| DialogType | SaveFile または OpenFile を指定します。 |
| FileFiler | コモンダイアログで表示するファイルに対するフィルタをしていします。 |
| IsMultiSelect | ファイル読込の場合、複数ファイルに対応するかどうかを指定します。true のときに複数ファイルに対応します。DialogType プロパティが SaveFile の場合は無視されます。 |
| Title | コモンダイアログのキャプション |

コード 5.1：CommonDialogBehavior 添付ビヘイビアのサンプルコード

**MainView.xaml**

```
<YK:Window x:Class="Section5_2.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
        xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
        Title="MainView" Height="300" Width="300">
    <StackPanel>
        <Button Content="Read File"
                Command="{Binding ReadFileCommand}"
                YKb:CommonDialogBehavior.Title="ファイルを読み込みます。"
                YKb:CommonDialogBehavior.IsMultiSelect="True"
                YKb:CommonDialogBehavior.FileFilter="CSVファイル(*.csv)|*.csv|すべてのファイル(*.*)|*.*"
                YKb:CommonDialogBehavior.DialogType="OpenFile"
                YKb:CommonDialogBehavior.Callback="{Binding ReadFileCallback}"
                />
        <Button Content="Write File"
                Command="{Binding WriteFileCommand}"
                YKb:CommonDialogBehavior.Title="ファイルを保存します。"
                YKb:CommonDialogBehavior.FileFilter="HTMLファイル(*.html;*.htm)|*.html;*.htm|すべてのファイル(*.*)|*.*"
                YKb:CommonDialogBehavior.DialogType="SaveFile"
                YKb:CommonDialogBehavior.Callback="{Binding WriteFileCallback}"
                />
    </StackPanel>
</YK:Window>
```

コード 5.2：CommonDialogBehavior 添付ビヘイビアのサンプルコード

**MainViewModel.cs**

```
namespace Section5_2.ViewModels
{
    using System;
    using YKToolkit.Bindings;

    public class MainViewModel : NotificationObject
    {
        private DelegateCommand readFileCommand;
        /// <summary>
        /// ファイル読込コマンドを取得します。
        /// </summary>
        public DelegateCommand ReadFileCommand
        {
            get
            {
                return readFileCommand ?? (readFileCommand = new DelegateCommand(_ =>
                {
                    ReadFileCallback = ReadFile;
                }));
            }
        }

        private DelegateCommand writeFileCommand;
        /// <summary>
        /// ファイル書込コマンドを取得します。
        /// </summary>
```

```
27          public DelegateCommand WriteFileCommand
28          {
29              get
30              {
31                  return writeFileCommand ?? (writeFileCommand = new DelegateCommand(_ =>
32                  {
33                      WriteFileCallback = WriteFile;
34                  }));
35              }
36          }
37
38          private Action<object, bool?> readFileCallback;
39          /// <summary>
40          /// ファイル読込コールバックを取得または設定します。
41          /// </summary>
42          public Action<object, bool?> ReadFileCallback
43          {
44              get { return readFileCallback; }
45              set { SetProperty(ref readFileCallback, value); }
46          }
47
48          private void ReadFile(object parameter, bool? result)
49          {
50              if ((result != null) && (result.Value))
51              {
52                  var filenames = parameter as string[];
53                  if (filenames != null)
54                  {
55                      foreach (var filename in filenames)
56                      {
57                          System.Console.WriteLine(filename);
58                      }
59                  }
60              }
61
62              ReadFileCallback = null;
63          }
64
65          private Action<object, bool?> writeFileCallback;
66          /// <summary>
67          /// ファイル書込コールバックを取得または設定します。
68          /// </summary>
69          public Action<object, bool?> WriteFileCallback
70          {
71              get { return writeFileCallback; }
72              set { SetProperty(ref writeFileCallback, value); }
73          }
74
75          private void WriteFile(object parameter, bool? result)
76          {
77              if ((result != null) && (result.Value))
78              {
79                  var filename = parameter as string;
80                  if (!string.IsNullOrEmpty(filename))
81                  {
82                      System.Console.WriteLine(filename);
83                  }
84              }
```

```
85
86                  WriteFileCallback = null;
87              }
88          }
89      }
```

CommonDialogBehavior 添付ビヘイビアは、Callback プロパティが null 以外に変化したときにコモンダイアログを呼び出すようにしています。その後、コモンダイアログにて決定されたファイルパスを Callback プロパティに指定されたコールバック関数に返して終了します。したがって、何度もコモンダイアログを呼び出せるように、コールバック関数内で Callback プロパティを null に戻す必要があります。上記コードでは 62 および 86 行目でこれをおこなっています。

Callback プロパティは Action<object, bool?> 型を指定し、object 型の引数はファイル読込の場合は string[] 型が、ファイル書込の場合は string 型がボックス化されています。それぞれ状況に応じてボックス化解除する必要があります。

## 5.3 DataGridBehavior

DataGridBehavior 添付ビヘイビアは、DataGrid コントロールの行ヘッダに行番号を付加するための添付ビヘイビアです。

DisplayRowNumber プロパティは int? 型で、任意の開始番号を指定することで DataGrid コントロールの行ヘッダに行番号を表示できます。表示したくないときは DisplayRowNumber プロパティに null を指定して下さい。下記のサンプルでは、開始番号に 1 を指定しているため、1 から開始された番号が自動的に行ヘッダに付加されます。

**コード 5.3：DataGridBehavior 添付ビヘイビアのサンプルコード**

**MainView.xaml**

```
1  <YK:Window x:Class="Section5_3.Views.MainView"
2                  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3                  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                  xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                  xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
6                  Title="MainView" Height="300" Width="300">
7      <Grid>
8          <DataGrid ItemsSource="{Binding People}"
9                          YKb:DataGridBehavior.DisplayRowNumber="1"
10                         />
11     </Grid>
12 </YK:Window>
```

**コード 5.4：DataGridBehavior 添付ビヘイビアのサンプルコード**

**Person.cs**

```
1  namespace Section5_3.Models
2  {
3      using YKToolkit.Bindings;
4
5      public class Person : NotificationObject
6      {
7          private string name;
8          /// <summary>
9          /// 名前を取得または設定します。
10         /// </summary>
11         public string Name
12         {
13             get { return name; }
14             set { SetProperty(ref name, value); }
15         }
16
17         private int age;
18         /// <summary>
```

```
19              /// 年齢を取得または設定します。
20              /// </summary>
21              public int Age
22              {
23                  get { return age; }
24                  set { SetProperty(ref age, value); }
25              }
26          }
27      }
```

コード 5.5：DataGridBehavior 添付ビヘイビアのサンプルコード

**MainViewModel.cs**

```
1   namespace Section5_3.ViewModels
2   {
3       using Section5_3.Models;
4       using System.Collections.ObjectModel;
5       using System.Linq;
6       using YKToolkit.Bindings;
7
8       public class MainViewModel : NotificationObject
9       {
10          /// <summary>
11          /// 新しいインスタンスを生成します。
12          /// </summary>
13          public MainViewModel()
14          {
15              People = new ObservableCollection<Person>(Enumerable.Range(0, 100).Select(i => new Person
16              {
17                  Name = "田中" + i.ToString() + "郎",
18                  Age = 20 + i,
19              }));
20              }
21
22              private ObservableCollection<Person> people;
23              /// <summary>
24              /// 個人情報リストを取得または設定します。
25              /// </summary>
26              public ObservableCollection<Person> People
27              {
28                  get { return people; }
29                  set { SetProperty(ref people, value); }
30              }
31          }
32      }
```

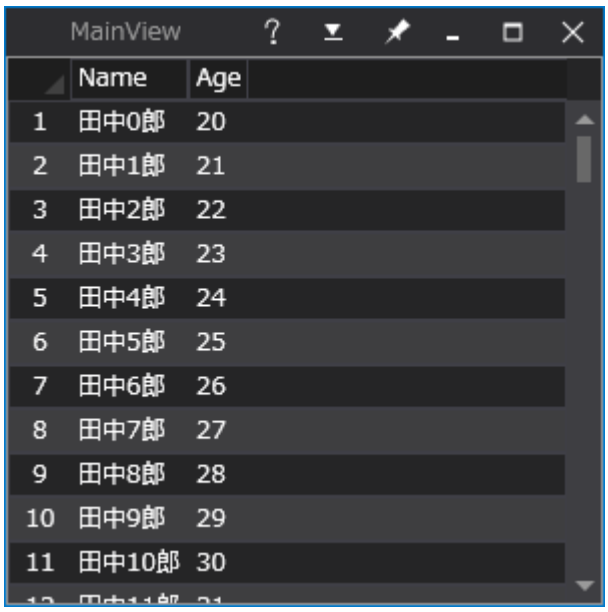



図 5.1：DataGridBehavior 添付ビヘイビアのサンプルアプリケーション

## 5.4 DirectorySelectDialogBehavior

DirectorySelectDialogBehavior 添付ビヘイビアは、ディレクトリ選択ダイアログを呼び出すための添付ビヘイビアです。ディレクトリ選択ダイアログには DirectorySelectDialog コントロールが使われています。使い方については 4.5 節を参照して下さい。

## 5.5 DragBehavior

DragBehavior 添付ビヘイビアは、任意のコントロールをドラッグ操作できるようにするための添付ビヘイビアです。FileTreeView コントロールにも使用されています。

コード 5.6：DragBehavior 添付ビヘイビアのサンプルコード

**MainView.xaml**

```
1  <YK:Window x:Class="Section5_5.Views.MainView"
2              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4              xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5              xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
6              Title="MainView" Height="300" Width="300">
7      <Grid>
8          <Grid.ColumnDefinitions>
9              <ColumnDefinition />
10             <ColumnDefinition />
11         </Grid.ColumnDefinitions>
12
13         <Ellipse Fill="Orange" Width="32" Height="32"
14                  YKb:DragBehavior.IsEnable="True"
15                  />
16
17         <Rectangle Grid.Column="1"
18                    Fill="Cyan" Width="32" Height="32"
19                    AllowDrop="True" Drop="Rectangle_Drop"
20                    />
21
22         <TextBlock Grid.ColumnSpan="2"
23                    Text="左の円を右の四角にドラッグすると出力ウィンドウにメッセージが表示されます。"
24                    TextWrapping="Wrap"
```

```
25                          VerticalAlignment="Top" />
26          </Grid>
27  </YK:Window>
```

コード 5.7：DragBehavior 添付ビヘイビアのサンプルコード

**MainView.xaml.cs**

```
1   namespace Section5_5.Views
2   {
3       using YKToolkit.Controls;
4
5       /// <summary>
6       /// MainView.xaml の相互作用ロジック
7       /// </summary>
8       public partial class MainView : Window
9       {
10          public MainView()
11          {
12              InitializeComponent();
13          }
14
15          private void Rectangle_Drop(object sender, System.Windows.DragEventArgs e)
16          {
17              foreach (var format in e.Data.GetFormats())
18              {
19                  System.Console.WriteLine(format);
20              }
21          }
22      }
23  }
```


図 5.2：DragBehavior 添付ビヘイビアのサンプルアプリケーション

このサンプルコードでは MainView のコードビハインドを使用していますが、実際には Drop イベントに対する添付ビヘイビアを個別に用意するなどして、ViewModel 側で操作をおこないます。

## 5.6 FileDropBehavior

FileDropBehavior 添付ビヘイビアは、ファイルドロップイベントのコールバック関数をデータバインドできるようにするための添付ビヘイビアです。

**コード 5.8：FileDropBehavior 添付ビヘイビアのサンプルコード**

**MainView.xaml**

```
<YK:Window x:Class="Section5_6.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
        xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
        Title="MainView" Height="300" Width="300"
        AllowDrop="True"
        YKb:FileDropBehavior.Callback="{Binding FileDropCallback}">
    <Grid>
        <TextBlock Text="ファイルをドラッグ＆ドロップすると出力ウィンドウにメッセージが表示されます。"
                TextWrapping="Wrap" />
    </Grid>
</YK:Window>
```

**コード 5.9：FileDropBehavior 添付ビヘイビアのサンプルコード**

**MainViewModel.cs**

```
namespace Section5_6.ViewModels
{
    using System;
    using YKToolkit.Bindings;

    public class MainViewModel : NotificationObject
    {
        /// <summary>
        /// ファイルドロップに対するコールバックを取得します。
        /// </summary>
        public Action<string[]> FileDropCallback
        {
            get { return OnFileDrop; }
        }

        /// <summary>
        /// ファイルドロップ時のコールバック
        /// </summary>
        /// <param name="filenames">ファイル名配列</param>
        private void OnFileDrop(string[] filenames)
        {
            foreach (var filename in filenames)
            {
                System.Console.WriteLine(filename);
            }
        }
    }
}
```

サンプルコードにあるように、FileDropBehavior 添付ビヘイビアには Callback 添付プロパティがあります。このプロパティは Action<string[]> 型で、string[] 型を入力引数に持つメソッドを指定することで、ファイルドロップ時にこのメソッドがコールされるようになります。

使用するときは必ずコントロールに対する AllowDrop プロパティを true にする必要があります。

## 5.7 KeyDownBehavior

KeyDownBehavior 添付ビヘイビアは、ある特定のキーに対してコマンドを割り当てることができる添付ビヘイビアです。ICommand インターフェースを使用するため、CanExecute() メソッドによる実行可能判定も有効となりま

す。したがって、キーに対するコマンドを有効/無効にするにはコマンドと同様に記述することができます。また、ShiftCommand プロパティに割り当てられたコマンドは、Shift を押しながら指定されたキーを押したときに実行されるコマンドとなります。

**コード 5.10：KeyDownBehavior 添付ビヘイビアのサンプルコード**

**MainView.xaml**

```
<YK:Window x:Class="Section5_7.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
        xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
        Title="MainView" Height="300" Width="300"
        YKb:KeyDownBehavior.Key="Escape"
        YKb:KeyDownBehavior.Command="{Binding MinimizeCommand}"
        YKb:KeyDownBehavior.ShiftCommand="{Binding MaximizeCommand}">
    <Grid>
        <StackPanel>
            <CheckBox Content="IsEnabled" IsChecked="{Binding IsEnabled}" />
            <TextBlock Text="チェックを入れると、ESC キーで最小化、Shift+ESC キーで最大化/元に戻す動作を実行できます。"
                       TextWrapping="Wrap"
                       />
        </StackPanel>
    </Grid>
</YK:Window>
```

**コード 5.11：KeyDownBehavior 添付ビヘイビアのサンプルコード**

**MainViewModel.cs**

```
namespace Section5_7.ViewModels
{
    using YKToolkit.Bindings;

    public class MainViewModel : NotificationObject
    {
        private bool isEnabled;
        /// <summary>
        /// 有効性を取得または設定します。
        /// </summary>
        public bool IsEnabled
        {
            get { return isEnabled; }
            set { SetProperty(ref isEnabled, value); }
        }

        private DelegateCommand minimizeCommand;
        /// <summary>
        /// 最小化コマンドを取得します。
        /// </summary>
        public DelegateCommand MinimizeCommand
        {
            get
            {
                return minimizeCommand ?? (minimizeCommand = new DelegateCommand(
                _ =>
                {
                    App.Instance.MinimizeMainView();
                },
                _ => IsEnabled));
```

```
31                  }
32              }
33
34              private DelegateCommand maximizeCommand;
35              /// <summary>
36              /// 最大化コマンドを取得します。
37              /// </summary>
38              public DelegateCommand MaximizeCommand
39              {
40                  get
41                  {
42                      return maximizeCommand ?? (maximizeCommand = new DelegateCommand(
43                      _ =>
44                      {
45                          App.Instance.MaximizeMainView();
46                      },
47                      _ => IsEnabled));
48                  }
49              }
50          }
51      }
```

コード 5.12：KeyDownBehavior 添付ビヘイビアのサンプルコード

**App.xaml.cs**

```
1   namespace Section5_7
2   {
3       using Section5_7.ViewModels;
4       using Section5_7.Views;
5       using System.Windows;
6
7       /// <summary>
8       /// App.xaml の相互作用ロジック
9       /// </summary>
10      public partial class App : Application
11      {
12          /// <summary>
13          /// 現在の WPF Application のインスタンスを取得します。
14          /// </summary>
15          public static App Instance { get { return Application.Current as App; } }
16
17          private MainView _mainView;
18
19          protected override void OnStartup(StartupEventArgs e)
20          {
21              base.OnStartup(e);
22
23              var vm = new MainViewModel();
24
25              _mainView = new MainView();
26              _mainView.DataContext = vm;
27              _mainView.Show();
28          }
29
30          /// <summary>
31          /// メインウィンドウを最小化します。
32          /// </summary>
33          public void MinimizeMainView()
34          {
```

```
35                  if (_mainView != null)
36                      _mainView.WindowState = WindowState.Minimized;
37              }
38
39              /// <summary>
40              /// メインウィンドウを最大化または元に戻します。
41              /// </summary>
42              public void MaximizeMainView()
43              {
44                  if (_mainView != null)
45                  {
46                      if (_mainView.WindowState == WindowState.Normal)
47                          _mainView.WindowState = WindowState.Maximized;
48                      else
49                          _mainView.WindowState = WindowState.Normal;
50                  }
51              }
52          }
53      }
```

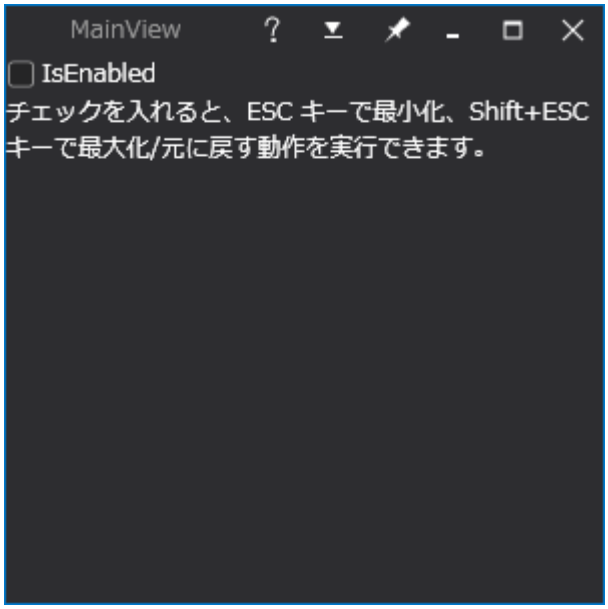



**図 5.3：KeyDownBehavior 添付ビヘイビアのサンプルアプリケーション**

## 5.8 RoutedEventTriggerBehavior

RoutedEventTriggerBehavior 添付ビヘイビアは、指定された RoutedEvent に対してコマンドを割り当てることができる添付ビヘイビアです。

**コード 5.13：RoutedEventTriggerBehavior 添付ビヘイビアのサンプルコード**

**MainView.xaml**

```
1   <YK:Window x:Class="Section5_8.Views.MainView"
2                  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3                  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                  xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                  xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
6                  Title="MainView" Height="200" Width="400">
7       <Grid>
8           <StackPanel>
9               <YK:TextBox Watermark="フォーカスするとコマンドが実行されます。"
```

```
                                YKb:RoutedEventTriggerBehavior.RoutedEvent="GotFocus"
                                YKb:RoutedEventTriggerBehavior.Command="{Binding GotFocusCommand}"
                                />
            <YK:TextBox Watermark="フォーカスしてもコマンドは実行されません。" />
            <YK:TextBox Watermark="フォーカスしてもコマンドは実行されません。" />
            <YK:TextBox Watermark="フォーカスしてもコマンドは実行されません。" />
            <TextBlock Text="{Binding Message}" />
        </StackPanel>
    </Grid>
</YK:Window>
```

コード 5.14：RoutedEventTriggerBehavior 添付ビヘイビアのサンプルコード

**MainView.xaml**

```
namespace Section5_8.ViewModels
{
    using YKToolkit.Bindings;

    public class MainViewModel : NotificationObject
    {
        private int count;

        private string message;
        /// <summary>
        /// メッセージを取得または設定します。
        /// </summary>
        public string Message
        {
            get { return message; }
            set { SetProperty(ref message, value); }
        }

        private DelegateCommand gotFocusCommand;
        /// <summary>
        /// フォーカス時のコマンドを取得します。
        /// </summary>
        public DelegateCommand GotFocusCommand
        {
            get
            {
                return gotFocusCommand ?? (gotFocusCommand = new DelegateCommand(_ =>
                {
                    Message = (count++).ToString();
                }));
            }
        }
    }
}
```

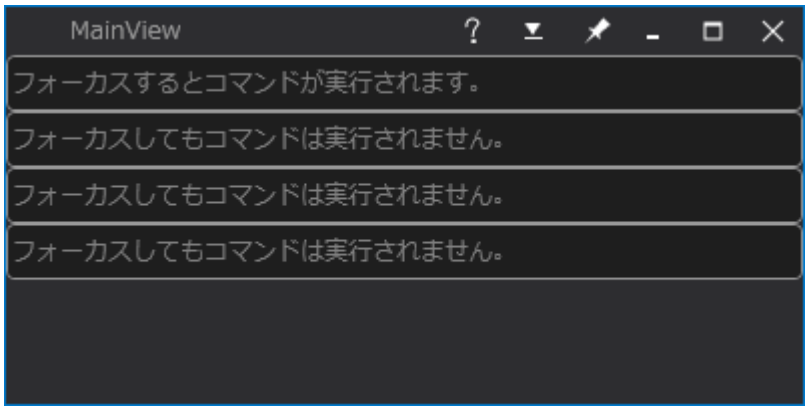



図 5.4：RoutedEventTriggerBehavior 添付ビヘイビアのサンプルアプリケーション

上記のサンプルでは 1 つめの TextBox コントロールに対して GotFocus イベントにコマンドを割り付けています。したがって、Tab キーでキーボードフォーカスを移動させていくと、1 つめの TextBox コントロールにフォーカスが移動するたびに GotFocusCommand コマンドが実行されることになります。

## 5.9 SystemMenuBehavior

SystemMenuBehavior 添付ビヘイビアは、ウィンドウコントロールのシステムメニューの表示や、Alt+F4 によるアプリケーションの終了をおこなわないようにするための添付ビヘイビアです。

コード 5.15：SystemMenuBehavior 添付ビヘイビアのサンプルコード

**MainView.xaml**

```
1  <YK:Window x:Class="Section5_9.Views.MainView"
2                  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3                  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                  xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                  xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
6                  Title="MainView" Height="100" Width="400"
7                  YKb:SystemMenuBehavior.IsSystemMenuEnabled="True">
8      <Grid>
9          <TextBlock Text="Alt+Space キーでシステムメニューが表示されます。" />
10     </Grid>
11 </YK:Window>
```

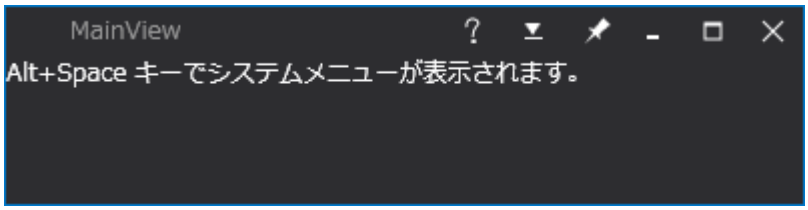



図 5.5：SystemMenuBehavior 添付ビヘイビアのサンプルアプリケーション

YKToolkit.Controls.Window コントロールは、標準でシステムメニューを無効にしています。上記のようなコードを追加することで、システムメニューを有効にできます。

逆に、System.Windows.Window コントロールでシステムメニューを無効にしたい場合は、SystemMenuBehavior 添付ビヘイビアを使って、IsSystemMenuEnabled プロパティに false を指定します。

## 5.10 TextBoxGotFocusBehavior

TextBoxGotFocusBehavior 添付ビヘイビアは、TextBox コントロール選択時に、既に入力されているテキスト全体を選択状態にするための添付ビヘイビアです。YKToolkit.Controls が提供する WPF テーマでは、TextBox コントロールはこの添付ビヘイビアによって標準でテキストが全選択されるようになっています。

コード 5.16：TextBoxGotFocusBehavior 添付ビヘイビアのサンプルコード

MainView.xaml

```
<YK:Window x:Class="Section5_10.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
        xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
        Title="MainView" Height="300" Width="300">
    <Grid>
        <StackPanel>
            <TextBox Text="フォーカス時に全選択されます。" />
            <TextBox Text="フォーカス時に全選択されます。"
                     YKb:TextBoxGotFocusBehavior.SelectAllOnGotFocus="True" />
            <TextBox Text="フォーカス時に全選択されません。"
                     YKb:TextBoxGotFocusBehavior.SelectAllOnGotFocus="False" />
            <TextBox Text="フォーカス時に全選択されません。" Style="{x:Null}" />
        </StackPanel>
    </Grid>
</YK:Window>
```

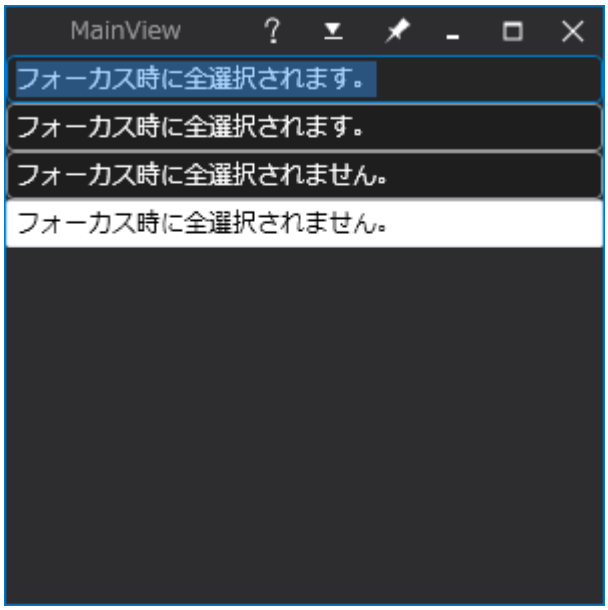



図 5.6：TextBoxGotFocusBehavior 添付ビヘイビアのサンプルアプリケーション

上記のサンプルでは、TextBox コントロールを 4 つ並べていますが、上から順に、YKToolkit.Controls の提供する WPF テーマが適用された TextBox コントロール、TextBoxGotFocusBehavior 添付ビヘイビアの動作を有効にしているもの、無効にしているもの、YKToolkit.Controls が提供する WPF テーマを解除して WPF 標準の TextBox コントロールにしたものとなっています。上 2 つの TextBox コントロールはフォーカス時にテキストが全選択状態になります。

## 5.11 WriteBitmapBehavior

WriteBitmapBehavior 添付ビヘイビアは、コントロールをビットマップ画像として保存するための添付ビヘイビアです。

サンプルとして、紫色の背景色を持った領域をビットマップ画像として保存することを考えます。

コード 5.17：WriteBitmapBehavior 添付ビヘイビアのサンプルコード

MainView.xaml

```
<YK:Window x:Class="Section5_11.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
4                 xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                 xmlns:YKb="clr-namespace:YKToolkit.Controls.Behaviors;assembly=YKToolkit.Controls"
6                 Title="MainView" Height="300" Width="300">
7         <Grid>
8             <Grid.RowDefinitions>
9                 <RowDefinition />
10                <RowDefinition Height="2*" />
11            </Grid.RowDefinitions>
12
13            <StackPanel>
14                <Button Content="Write Bitmap" Command="{Binding WriteBitmapCommand}" />
15            </StackPanel>
16
17            <Border Grid.Row="1" Background="Purple">
18                <Canvas YKb:WriteBitmapBehavior.Callback="{Binding WriteBitmapCallback}">
19                    <Ellipse Fill="Orange" Width="32" Height="32"
20                             Canvas.Left="20" Canvas.Top="40"
21                             />
22                    <Rectangle Fill="Green" Width="32" Height="32"
23                               Canvas.Right="10" Canvas.Bottom="10"
24                               />
25                </Canvas>
26            </Border>
27        </Grid>
28    </YK:Window>
```

**コード 5.18：WriteBitmapBehavior 添付ビヘイビアのサンプルコード**

**MainView.xaml**

```
1   namespace Section5_11.ViewModels
2   {
3       using System;
4       using YKToolkit.Bindings;
5
6       public class MainViewModel : NotificationObject
7       {
8           private DelegateCommand writeBitmapCommand;
9           /// <summary>
10          /// ビットマップ保存コマンドを取得します。
11          /// </summary>
12          public DelegateCommand WriteBitmapCommand
13          {
14              get
15              {
16                  return writeBitmapCommand ?? (writeBitmapCommand = new DelegateCommand(_ =>
17                  {
18                      WriteBitmapCallback = GetBitmapFileName;
19                  }));
20              }
21          }
22
23          private Func<string> writeBitmapCallback;
24          /// <summary>
25          /// ビットマップ保存先フルパスを返すコールバックを取得します。
26          /// </summary>
27          public Func<string> WriteBitmapCallback
28          {
29              get { return writeBitmapCallback; }
30              set { SetProperty(ref writeBitmapCallback, value); }
```

```
31              }
32
33              /// <summary>
34              /// ビットマップ保存先フルパスを取得します。
35              /// </summary>
36              /// <returns>ファイルのフルパス</returns>
37              private string GetBitmapFileName()
38              {
39                  WriteBitmapCallback = null;
40                  return "D:\\test.bmp";
41              }
42          }
43      }
```

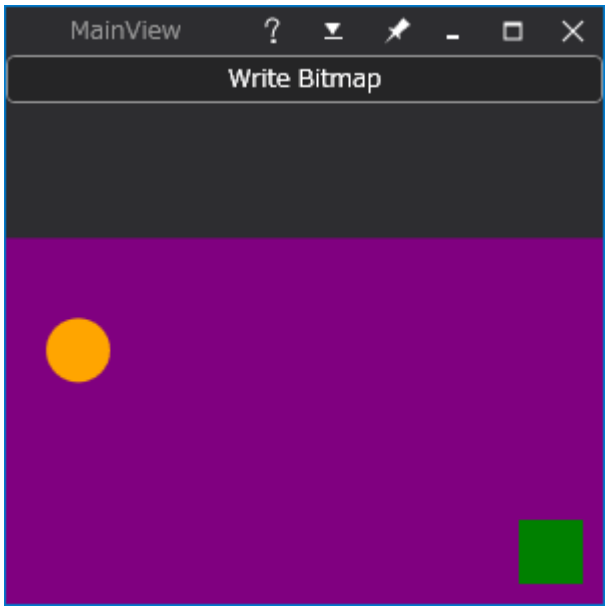



図 5.7：SystemMenuBehavior 添付ビヘイビアのサンプルアプリケーション

ここでは D ドライブ直下に "test.bmp" という名前で保存していますが、ドライブへのアクセス権等の問題もありますので、各自の環境に合わせて適宜変更して下さい。CommonDialogBehavior 添付ビヘイビアと組み合わせてコモンダイアログによるファイル指定としても構いません。

さて、上記の XAML コードでは、Canvas コントロールに対して WriteBitmapBehavior 添付ビヘイビアを設定しています。このまま実行し、「Write Bitmap」ボタンを押すと、次のような画像が保存されます。

図 5.8：ビットマップ画像として保存された Canvas コントロール

Canvas コントロールには Background プロパティによる背景色の指定がないため、Canvas コントロールをビットマップ画像保存対象としてしまうと、背景色が黒い画像となってしまいます。背景色は親要素である Border コントロールが持っているため、Border コントロールを保存対象とするようにコードを変更します。

**コード 5.19：Border コントロールを保存対象とするように変更したコードの一部**

**MainView.xaml**

```
1  <Border Grid.Row="1" Background="Purple"
2          YKb:WriteBitmapBehavior.Callback="{Binding WriteBitmapCallback}">
3      <Canvas>
4          <Ellipse Fill="Orange" Width="32" Height="32"
5                   Canvas.Left="20" Canvas.Top="40"
6                   />
7          <Rectangle Fill="Green" Width="32" Height="32"
8                   Canvas.Right="10" Canvas.Bottom="10"
9                   />
10     </Canvas>
11 </Border>
```

Canvas コントロールに添付していた WriteBitmapBehavior 添付ビヘイビアを Border コントロールに添付し直しただけです。しかし、この状態で「Write Bitmap」ボタンを押して画像を保存すると、次のような画像が保存されます。

**図 5.9：ビットマップ画像として保存された Border コントロール**

ビットマップ画像として保存されるコントロールの親要素が Grid コントロールの場合、Grid.Row="0"、Grid.Column="0" の左上座標を基準座標としてコントロールのサイズ分の画像を保存するため、上図のように保存したい Border コントロールがずれた画像が保存されてしまいます。この問題は、保存対象コントロールの親要素が Grid コントロールにならないように、例えば Border コントロールでラップすることで回避できます。例えば次のように変更することで回避できます。

**コード 5.20：Border コントロールを保存対象とするように変更したコードの一部**

**MainView.xaml**

```
1  <Border Grid.Row="1">
2      <Border Background="Purple"
3              YKb:WriteBitmapBehavior.Callback="{Binding WriteBitmapCallback}">
4          <Canvas>
5              <Ellipse Fill="Orange" Width="32" Height="32"
6                       Canvas.Left="20" Canvas.Top="40"
7                       />
8              <Rectangle Fill="Green" Width="32" Height="32"
9                       Canvas.Right="10" Canvas.Bottom="10"
10                      />
11         </Canvas>
12     </Border>
13 </Border>
```

図 5.10：ビットマップ画像として保存された Border コントロール

## 5.12 サンプルプロジェクト

| ソリューション名 | プロジェクト名 | 概　要 |
|---|---|---|
| Section5 | Section5_2 | CommonDialogBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_3 | DataGridBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_5 | DragBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_6 | FileDropBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_7 | KeyDownBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_8 | RoutedEventTriggerBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_9 | SystemMenuBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_10 | TextBoxGotFocusBehavior 添付ビヘイビアに関するサンプルプログラム |
| | Section5_11 | WriteBitmapBehavior 添付ビヘイビアに関するサンプルプログラム |

# 6 ComboBox のためのアイテムリスト

この章では、YKToolkit.Controls.dll で公開されている、ComboBox で使用するためのアイテムリストを紹介します。詳細については YKToolkit 付属のヘルプドキュメントファイルをご参照ください。

## 6.1 概要

列挙体を ComboBox のアイテムリストとして扱う場合、以下に紹介するアイテムリストのクラスを利用すると便利です。

**表 6.1：ComboBox のためのアイテムリスト一覧表**

| アイテムリスト名 | 概 要 |
|---|---|
| ComboBoxDashStyles | DashStyles 列挙体を ComboBox で扱いやすくするためのクラスです。 |
| ComboBoxLineGraphLegendLocations | LegendLocations 列挙体を ComboBox で扱いやすくするためのクラスです。 |
| ComboBoxLineGraphMarkerTypes | LineGraphMarkerTypes 列挙体を ComboBox で扱いやすくするためのクラスです。 |
| ComboBoxLineGraphMoveOperationModes | OperationModes 列挙体の移動モードのみを ComboBox で扱いやすくするためのクラスです。 |
| ComboBoxLineGraphZoomOperationModes | OperationModes 列挙体の拡大モードのみを ComboBox で扱いやすくするためのクラスです。 |
| ComboBoxTickPlacements | TickPlacement 列挙体を ComboBox で扱いやすくするためのクラスです。 |
| ComboBoxTransitionDirections | TransitionDirections 列挙体を ComboBox で扱いやすくするためのクラスです。 |
| ComboBoxTransitionModes | TransitionModes 列挙体を ComboBox で扱いやすくするためのクラスです。 |

## 6.2 内部構造

どのアイテムリストも下表に示すプロパティを持っています。

**表 6.2：ComboBox のためのアイテムリスト一覧表**

| アイテムリスト名 | 概 要 |
|---|---|
| Items | 自分自身のクラスをリストとして持っている静的プロパティ |
| Item | ある列挙体のひとつの値 |
| Name | ある列挙体のひとつの値に対する名前 |

ひとつひとつのインスタンスは Item プロパティと Name プロパティを持っていて、ある列挙体の値に対して名前を保持しています。このクラスをリストとして保持しているのが Items 静的プロパティで、これを使って ComboBox コントロールにリスト表示します。

例えば ComboBoxLineGraphMoveOperationModes クラスの内部実装は次のようになっています。

**コード 6.1：ComboBoxLineGraphMoveOperationModes クラスの実装**

**ComboBoxLineGraphMoveOperationModes.cs**

```
1  namespace YKToolkit.Controls
2  {
3      using System.Collections.Generic;
4      using System.Windows.Media;
5
```

```
6       /// <summary>
7       /// OperationModes (移動のみ)選択用列挙体表示用クラス
8       /// </summary>
9       public class ComboBoxLineGraphMoveOperationModes
10      {
11          /// <summary>
12          /// アイテムを取得します。
13          /// </summary>
14          public OperationModes Item { get; protected set; }
15          /// <summary>
16          /// 名前を取得します。
17          /// </summary>
18          public string Name { get; protected set; }
19
20          /// <summary>
21          /// リストを取得します。
22          /// </summary>
23          public static List<ComboBoxLineGraphMoveOperationModes> Items { get; private set; }
24
25          /// <summary>
26          /// 新しいインスタンスを生成します。
27          /// </summary>
28          /// <param name="item">アイテムを指定します。</param>
29          /// <param name="name">アイテムに対する名前を指定します。</param>
30          public ComboBoxLineGraphMoveOperationModes(OperationModes item, string name)
31          {
32              Item = item;
33              Name = name;
34          }
35
36          static ComboBoxLineGraphMoveOperationModes()
37          {
38              Items = new List<ComboBoxLineGraphMoveOperationModes>()
39              {
40                  new ComboBoxLineGraphMoveOperationModes(OperationModes.None, "None"),
41                  new ComboBoxLineGraphMoveOperationModes(OperationModes.MoveX, "X-Axis"),
42                  new ComboBoxLineGraphMoveOperationModes(OperationModes.MoveY, "Y-Axis"),
43                  new ComboBoxLineGraphMoveOperationModes(OperationModes.MoveXY, "XY-Axis"),
44                  new ComboBoxLineGraphMoveOperationModes(OperationModes.MoveY2, "Y2-Axis"),
45                  new ComboBoxLineGraphMoveOperationModes(OperationModes.MoveXY2, "XY2-Axis"),
46              };
47          }
48      }
49  }
```

静的コンストラクタの中で Items 静的プロパティの初期化をおこなっており、ここで列挙体の各値に対して名前を決定しています。名前を変更したい場合はこの部分を変更したクラスを用意します。

## 6.3 使用方法

ここでは ComboBoxLineGraphMoveOperationModes クラスを例に具体的な使用方法を紹介します。

まず、View 側に ComboBox コントロールを配置し、その ItemsSource プロパティに ComboBoxLineGraphMoveOperationModes クラスの Items 静的プロパティを割り当てます。

**コード 6.2：ComboBoxLineGraphMoveOperationModes クラスの使用例**

**MainView.xaml**

```
1   <YK:Window x:Class="Section6_3.Views.MainView"
2              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
3                 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                 xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                 Title="MainView" Height="300" Width="300">
6         <Grid>
7             <StackPanel>
8                 <TextBlock Text="移動モード :" />
9                 <ComboBox ItemsSource="{Binding Source={x:Static
  YK:ComboBoxLineGraphMoveOperationModes.Items}}"
10                              SelectedItem="{Binding MoveOperationModeItem}">
11                    <ComboBox.ItemTemplate>
12                        <DataTemplate>
13                            <TextBlock Text="{Binding Name}" />
14                        </DataTemplate>
15                    </ComboBox.ItemTemplate>
16                </ComboBox>
17            </StackPanel>
18        </Grid>
19    </YK:Window>
```

このように、x:Static を用いることで静的プロパティを割り当てることができます。また、ComboBox のリストとして表示する場合、Name プロパティを参照するように DataTemplate を指定しています。

ViewModel 側では選択されたアイテムから列挙体の値を取り出せるようにします。

**コード 6.3：ComboBoxLineGraphMoveOperationModes クラスの使用例**

**MainViewModel.cs**

```
1   namespace Section6_3.ViewModels
2   {
3       using System.Linq;
4       using YKToolkit.Bindings;
5       using YKToolkit.Controls;
6
7       public class MainViewModel : NotificationObject
8       {
9           public MainViewModel()
10          {
11              // MoveOperationMode プロパティを変更しても ComboBox の選択アイテムが変更される
12              MoveOperationMode = OperationModes.None;
13          }
14
15          private ComboBoxLineGraphMoveOperationModes moveOperationModeItem =
  ComboBoxLineGraphMoveOperationModes.Items[0];
16          /// <summary>
17          /// 移動操作モードリストによる選択項目を取得または設定します。
18          /// </summary>
19          public ComboBoxLineGraphMoveOperationModes MoveOperationModeItem
20          {
21              get { return moveOperationModeItem; }
22              set
23              {
24                  if (SetProperty(ref moveOperationModeItem, value))
25                  {
26                      MoveOperationMode = moveOperationModeItem.Item;
27                  }
28              }
29          }
30
31          private OperationModes moveOperationMode;
```

```
32          /// <summary>
33          /// 移動操作モードを取得または設定します。
34          /// </summary>
35          public OperationModes MoveOperationMode
36          {
37              get { return moveOperationMode; }
38              set
39              {
40                  if (SetProperty(ref moveOperationMode, value))
41                  {
42                      MoveOperationModeItem =
    ComboBoxLineGraphMoveOperationModes.Items.FirstOrDefault(item => item.Item == moveOperationMode);
43                  }
44              }
45          }
46      }
47  }
```

ComboBox コントロールからリストを選択すると、MoveOperationModeItem プロパティを通じて MoveOperationMode プロパティが変更されるようになっています。また、MoveOperationModeItem プロパティが変更されれば ComboBox コントロールの選択アイテムが変更されることはもちろんですが、MoveOperationMode プロパティが変更されても同じ挙動になるように、ここでは System.Linq を用いて 1 行で簡潔に表記しています。

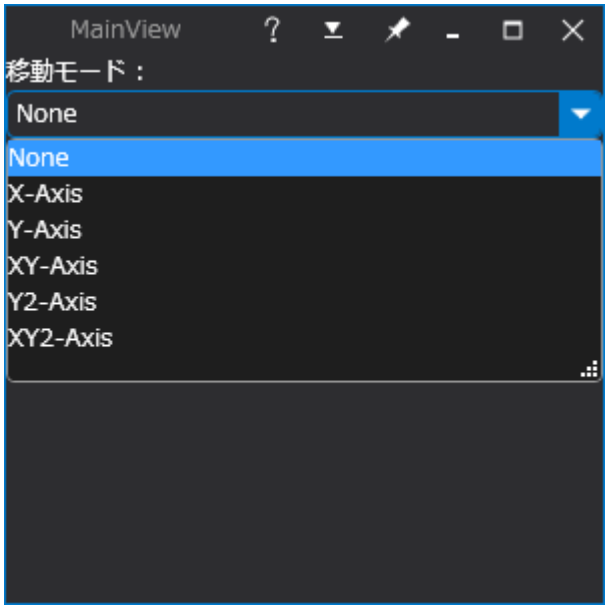



**図 6.1：ComboBoxLineGraphMoveOperationModes クラスで指定された名前がリストになっている**

## 6.4 サンプルプロジェクト

| ソリューション名 | プロジェクト名 | 概 要 |
|---|---|---|
| Section6 | Section6_3 | ComboBoxLineGraphMoveOperationModes クラスに関するサンプルプログラム |

# 7 Converter

## 7.1 概要

View 側でデータバインディングをおこなったデータに対して変換をおこなうためのコンバータの一覧を下表に掲載します。

**表 7.1：IValueConverter を実装したクラスの一覧表**

| コンバータ名 | 概 要 |
|---|---|
| InverseBooleanConverter | bool 型のデータを対象として、そのデータを反転するコンバータです。 |
| InverseDockConverter | Dock 列挙体のデータを対象として、そのデータの反対を示す値に変換するコンバータです。 |
| NegativeDoubleConverter | double 型のデータを対象として、その値の符号を反転させるコンバータです。 |

## 7.2 使用方法

ここでは InverseBooleanConverter クラスを例に具体的な使用方法を紹介します。

コンバータは一般的に XAML コードで使用するもので、Window または UserControl などのルートタグに対する Resources で StaticResource として定義して使用します。

**コード 7.1：ComboBoxLineGraphMoveOperationModes クラスの使用例**

**MainView.xaml**

```
1  <YK:Window x:Class="Section7_2.Views.MainView"
2                  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3                  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4                  xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5                  Title="MainView" Height="300" Width="300">
6      <YK:Window.Resources>
7          <YK:InverseBooleanConverter x:Key="InverseBooleanConverter" />
8      </YK:Window.Resources>
9      <StackPanel>
10         <CheckBox x:Name="checkbox1" Content="Check me!" />
11         <CheckBox Content="Don't touch me!"
12                   IsEnabled="False"
13                   IsChecked="{Binding IsChecked, ElementName=checkbox1, Converter={StaticResource
   InverseBooleanConverter}}" />
14     </StackPanel>
15 </YK:Window>
```

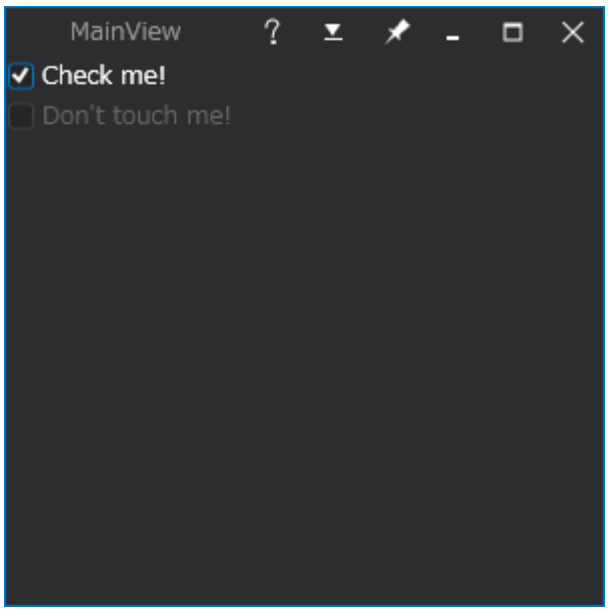



図 7.1：InverseBooleanConverter コンバータによって true/false が反転している

## 7.1 サンプルプロジェクト

| ソリューション名 | プロジェクト名 | 概　要 |
|---|---|---|
| Section7 | Section7_2 | InverseBooleanConverter クラスに関するサンプルプログラム |

# 8 マークアップ拡張

## 8.1 概要

View 側でデータバインドを記述する際に、{Binding vairablename} という記法を用いますが、これをマークアップ拡張と呼びます。このマークアップ拡張は独自に追加することができます。YKToolkit.Controls.dll で公開しているマークアップ拡張を下表に掲載します。

**表 8.1：マークアップ拡張一覧表**

| マークアップ拡張名 | 概　要 |
|---|---|
| ComparisonBinding | バインディングされたデータと指定された値を比較することができるマークアップ拡張です。主に DataTrigger クラスに対して使用します。 |

## 8.2 ComparisonBinding マークアップ拡張の使用方法

コントロールにはその外観を定義するために Style を指定でき、Style.Trigger によって動的に外観を変更することができます。中でも DataTrigger はデータバインドによってより詳細な条件を与えることができます。DataTrigger は例えば次のように使います。

**コード 8.1：DataTrigger の使用例**

**MainView.xaml**

```
1  <YK:Window x:Class="Section8_2.Views.MainView"
2              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4              xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5              xmlns:YKbind="clr-namespace:YKToolkit.Bindings;assembly=YKToolkit.Controls"
6              Title="MainView" Height="300" Width="300">
7      <StackPanel>
8          <YK:SpinInput Value="{Binding Value}" Min="0" Max="10" Tick="1" />
9          <Ellipse>
10             <Ellipse.Style>
11                 <Style TargetType="{x:Type Ellipse}">
12                     <Setter Property="Width" Value="32" />
13                     <Setter Property="Height" Value="32" />
14                     <Setter Property="Fill" Value="Orange" />
15                     <Style.Triggers>
16                         <DataTrigger Binding="{Binding Value}" Value="2">
17                             <Setter Property="Fill" Value="Cyan" />
18                         </DataTrigger>
19                     </Style.Triggers>
20                 </Style>
21             </Ellipse.Style>
22         </Ellipse>
23     </StackPanel>
24 </YK:Window>
```

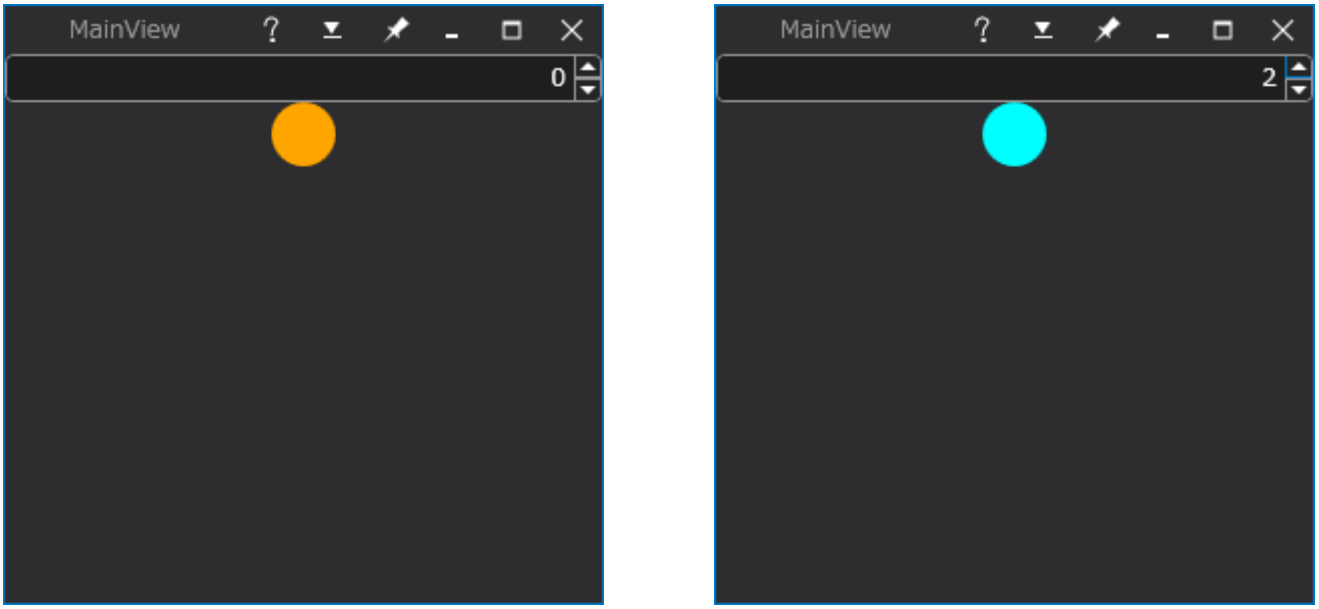
**コード 8.2：DataTrigger の使用例**

**MainViewModel.cs**

```
1  namespace Section8_2.ViewModels
2  {
3      using YKToolkit.Bindings;
4
```

```
5       public class MainViewModel : NotificationObject
6       {
7           private int _value;
8           /// <summary>
9           /// 値を取得または設定します。
10          /// </summary>
11          public int Value
12          {
13              get { return _value; }
14              set { SetProperty(ref _value, value); }
15          }
16      }
17  }
```

上記のコードでは、Value プロパティが 2 になったらシアン、それ以外はオレンジ色の円が表示されます。




(a) 値が 2 以外の場合はオレンジ　　(b) 値が 2 の場合はシアン

**図 8.1：DataTrigger によって動的に外観が変化する**

しかし、このままでは "値が x のときかそれ以外" という場合分けしかできず、例えば "値が x より大きいとき" などのように比較することができません。このような場合は ComparisonBinding を使用します。

上記の XAML を ComparisonBinding に書き換えたコードを以下に示します。

**コード 8.3：DataTrigger の使用例**

**MainView.xaml**

```
1   <YK:Window x:Class="Section8_2.Views.MainView"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4               xmlns:YK="clr-namespace:YKToolkit.Controls;assembly=YKToolkit.Controls"
5               xmlns:YKbind="clr-namespace:YKToolkit.Bindings;assembly=YKToolkit.Controls"
6               Title="MainView" Height="300" Width="300">
7       <StackPanel>
8           <YK:SpinInput Value="{Binding Value}" Min="0" Max="10" Tick="1" />
9           <Ellipse>
10              <Ellipse.Style>
11                  <Style TargetType="{x:Type Ellipse}">
12                      <Setter Property="Width" Value="32" />
```

```
13                          <Setter Property="Height" Value="32" />
14                          <Setter Property="Fill" Value="Orange" />
15                          <Style.Triggers>
16                              <DataTrigger Binding="{YKbind:ComparisonBinding DataContext.Value, GT, 2}"
    Value="{x:Null}">
17                                  <Setter Property="Fill" Value="Cyan" />
18                              </DataTrigger>
19                          </Style.Triggers>
20                      </Style>
21                  </Ellipse.Style>
22              </Ellipse>
23          </StackPanel>
24  </YK:Window>
```

ComparisonBinding では、Binding パス、比較演算子、比較する値の順にコンマ区切りで表記します。上記のコードでは Binding パスは DataContext.Value、比較演算子は GT、比較する値は 2 となります。

ここで、Binding パスは Style のターゲットとなっているコントロールから見たパスにする必要があります。上記の例では Ellipse コントロールの Value プロパティ（存在しない）ではなく、MainViewModel の Value プロパティを参照したいため、Ellipse コントロールの DataContext を明記しています。

比較演算子には GT（Greater Then）を指定しており、不等号 ">" を表しています。この他には下表のような比較演算子があります。

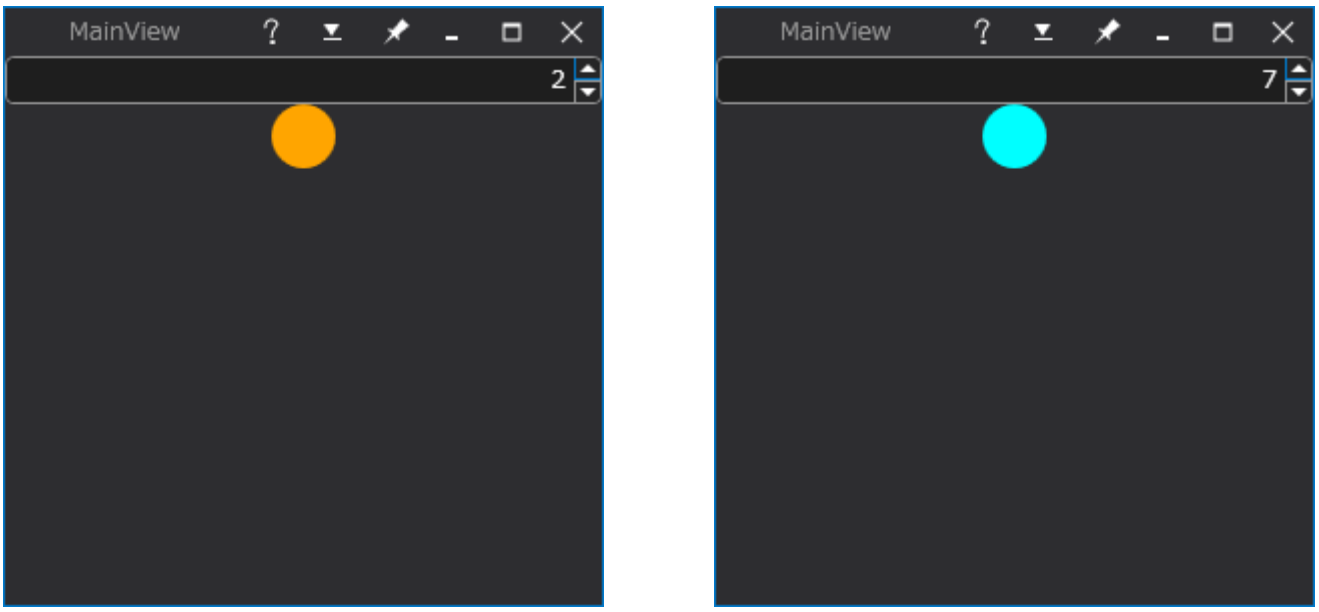
**表 8.2：マークアップ拡張一覧表**

| 比較演算子 | 記　号 | 概　要 |
|---|---|---|
| EQ | = | 等しいかどうかを調べます。 |
| LT | < | 小さいかどうかを調べます。 |
| LTE | <= | 以下かどうかを調べます。 |
| GT | > | 大きいかどうかを調べます。 |
| GTE | >= | 以上かどうかを調べます。 |
| NOT | != | 等しくないかどうかを調べます。 |

比較する値は即値を与えます。

上記の例では Value プロパティの値が 2 より大きいときに条件が成立することになります。条件が成立すると、null が返ってくるようになっています。したがって、上記のコードにもあるように、DataTrigger の Value プロパティには {x:Null} を指定します。

実行結果を以下に示します。




(a) 値が 2 より大きくないのでオレンジ (b) 値が 2 より大きいのでシアン

**図 8.2：ComparisonBinding によって値を比較しながら外観を変更する**

## 8.1 サンプルプロジェクト

| ソリューション名 | プロジェクト名 | 概　要 |
|---|---|---|
| Section8 | Section8_2 | ComparisonBinding クラスに関するサンプルプログラム |

# 改訂履歴

| 改訂年月日 | バージョン | 概　要 |
|---|---|---|
| 2015.02.05 | 1.0.0 | 初版リリース。 |
| 2015.03.13 | 1.1.0 | 章の構成を変更。<br>第 2 章の内容を YKToolkit.Controls Ver.1.9.0.0 以降に対応。<br>Extended Controls のサンプルを掲載。 |
| 2015.03.31 | 1.2.0 | 添付ビヘイビアのサンプルを掲載。 |
| 2015.04.02 | 1.3.0 | ComboBox のためのアイテムリスト、Converter、マークアップ拡張に関するサンプルを掲載。 |
| 2015.04.13 | 1.3.1 | WriteBitmapBehavior の節の開始位置を修正。 |
| | | |

# YKToolkit.Controls 取扱説明書

2015.02.05 初版
2015.04.13 第 4 版 改訂 1