Windows Presentation Foundation
逆引き集

Ver.1.0.5

目次

1 はじめに 10

1.1 目的 10
1.2 注意 10
1.3 開発環境 10

2 共通する設定およびクラス 11

2.1 MVVM パターンを意識した内部構造 11
2.1.1 MainWindow.xaml および MainWindow.xaml.cs を削除する 11
2.1.2 "Views"、"ViewModels"、"Models" フォルダを追加する 12
2.1.3 "Views" フォルダに MainView ウィンドウクラスを追加する 12
2.1.4 "ViewModels" フォルダに MainViewModel クラスを追加する 12
2.1.5 App.xaml に記述されている StartupUri プロパティを削除する 12
2.1.6 App.xaml.cs で OnStartup() メソッドをオーバーライドする 12
2.2 MVVM パターンのための基本的なクラス 13
2.2.1 NotificationObject クラス 13
2.2.2 DelegateCommand クラス 16
2.3 Person クラス 19
2.3.1 Gender 列挙体 19
2.3.2 Person クラス 19

3 C# によるあれこれ 21

3.1 byte[] と int などの基本データ型を相互に変換する (Tips_BitConverter) 21
3.1.1 int などのデータ型から byte[] に変換する 21
3.1.2 byte[] から int などのデータ型に変換する 22
3.2 string 文字列を文字コードに変換する (Tips_Encoding) 24
3.3 特殊ディレクトリのフルパスを取得する (Tips_SpecialFolderPath) 25
3.4 ログファイルを出力する (Tips_DebugTrace) 27
3.4.1 Debug クラスによるログファイル出力 27
3.4.2 Trace クラスによるログファイル出力 27
3.5 独自イベントの作成方法 (Tips_CustomEvent) 31
3.6 ソースにリンクしたデバッグ出力をおこなう (Tips_DebugTrace) 38
3.7 元に戻す/やり直し機能を実装するには (Tips_Undo) 40
3.7.1 データではなく操作を保持する 40
3.7.2 Action デリゲートで操作を保持する 40
3.7.3 元に戻す/やり直し機能を実装する 42
3.8 #if～#endif の代わりに Conditional 属性を使う (Tips_ConditionalAttribute) 48
3.8.1 #if～#endif が向かない例 48
3.8.2 Conditional 属性の使い方 48
3.8.3 複数シンボルの条件 49
3.9 シリアル通信をおこなうには (Tips_Serial) 51
3.9.1 ポートの解放と閉鎖 51
3.9.2 データを受信する 52
3.9.3 データを送信する 53

4 WPF によるあれこれ 55

4.1 XAML デザイナーで ViewModel のメンバを Intellisense 候補とする 55
4.2 デザイン実行かどうかを判定する (Tips_XamlDesigner) 57
4.3 StringFormat によってフォーマットを指定する (Tips_StringFormat) 59
4.4 コントロールを変形させたい（Tips_Transform） 62
4.4.1 RenderTransform プロパティ 62
4.4.2 LayoutTransform プロパティ 63
4.5 コントロールを変形したときのマウス位置を特定したい（Tips_Transform） 64
4.5.1 原点を中心に座標を回転する 64
4.5.2 ある点を中心に座標を回転する 64
4.5.3 Thumb コントロールの DragDelta イベントと DragCompleted イベント 65
4.6 ListBox コントロールなどを物理スクロールにする (Tips_PixelScroll) 67
4.7 ListBox のアイテム追加/削除のときにアニメーションする（Tips_AnimatedListBoxItem） 69
4.8 添付ビヘイビアを作成する (Tips_Behavior) 82
4.8.1 添付プロパティとは 82
4.8.2 添付プロパティの作成 82
4.8.3 ビヘイビアの作成 84
4.9 DataGrid コントロールの行ヘッダに行番号を表示する (Tips_DataGrid1) 86
4.10 ドラッグ操作でゴーストを表示する (Tips_Adorner) 91
4.10.1 Adorner クラス 91
4.10.2 ゴーストを表示するための Adorner 派生クラス 91
4.10.3 ゴーストを表示するための添付ビヘイビア 94
4.11 装飾のコントロールレイアウトを XAML で指定する（Tips_AdornerCore） 98
4.11.1 Adorner クラスからの派生クラスを用意する 98
4.11.2 添付プロパティの定義 100
4.11.3 使用例 103
4.11.4 DataTemplateSelector クラスによる装飾の切り替え 105
4.12 TabControl コントロールのあれこれ (Tips_TabControl) 112
4.12.1 ViewModel のコレクションを ItemsSource プロパティに指定する 112
4.12.2 タブを追加/削除する 115
4.13 多言語対応にする (Tips_MultiLanguage) 122
4.13.1 多言語化されたリソース 122
4.13.2 XAML からリソースを参照する 123
4.13.3 動的な言語切り替え 124
4.14 二重起動させないようにする (Tips_DisabledMultiInstance) 128
4.15 Alt+Tab メニューのウィンドウ一覧に表示させないようにする（Tips_AltTabMenuDisable） 130
4.16 「デスクトップの表示」ボタンを押しても最小化しないようにしたい（Tips_ShowDesktopDisable） 131
4.16.1 ShowDesktopBehavior 添付ビヘイビア 131
4.16.2 複数ウィンドウへの適用 134
4.17 ホットキーを登録する (Tips_HotKey) 135
4.17.1 P/Invoke によるネイティブコード呼び出し 135
4.17.2 添付ビヘイビアとして機能させる 137
4.18 ハンドルされていない例外を処理する (Tips_UnhandledException) 142
4.18.1 UI スレッドにおける未処理例外の捕捉 142
4.18.2 UI スレッド以外における未処理例外の捕捉 145
4.18.3 Task クラスを用いた非同期処理における未処理例外の捕捉 148
4.19 独自のマークアップ拡張を作成するには（Tips_MarkupExtension） 150
4.19.1 基本的な使い方 150

4.19.2 Binding マークアップ拡張を自作してみよう 153

5 アンチパターン 157

5.1 "やってはいけない" コードを検出するコード分析機能を使おう 157
5.2 SQL クエリ構文を string.Format で生成してはいけない（CA2100） 158
5.3 StreamReader/Writer クラスを using で入れ子にしてはいけない（CA2202） 160
5.4 コンストラクタ内でオーバーライド可能なメソッドを呼び出してはいけない（CA2214） 162

6 C/C++ による汎用 DLL あれこれ 164

6.1 C/C++ による汎用 DLL 作成用プロジェクト作成手順 164
6.2 DLL として関数を公開するには 167
6.3 C# から C/C++ DLL で公開されている関数を使用するには 169
6.4 文字列を渡す 170
6.5 文字列を返してもらう 172
6.6 構造体を渡す 174
6.7 構造体を返してもらう 177
6.8 ポインタを含む構造体を受け渡すには 180

7 Visual Studio に関するあれこれ 183

7.1 設定のインポートとエクスポート 183
7.2 ネットワーク共有におけるアクセス許可 184

8 おわりに 185

図 目次

図 2.1：MVVM パターンを意識した内部構造 11
図 3.1：byte[] に変換されている 22
図 3.2：int に変換されている 22
図 3.3：それぞれの文字コードに変換されている 24
図 3.4：特殊フォルダのフルパスが表示される 26
図 3.5：イベント発生回数がカウントされる 33
図 3.6：文字列の長さがカウントされる 37
図 3.7：Person プロパティが候補として表示されている 39
図 3.8：拡張機能 VSColorOutput を使用するとキーワードによって色付けされる 39
図 3.9：外部から指定された処理内容が実行される 41
図 3.10：スタックされた処理が逆順で実行される 42
図 3.11：受信した文字列が表示される 53
図 4.1：Person プロパティが候補として表示されている 56
図 4.2：デザイン時の DataContext 設定方法 56
図 4.3：実行時とデザイナー上の表示が異なる 58
図 4.4：StringFormat サンプルの実行結果 61
図 4.5：RenderTransform によるコントロールの変形はレイアウトに影響しない 62
図 4.6：LayoutTransform によるコントロールの変形はレイアウトに影響する 63
図 4.7：座標系の回転 64
図 4.8：ある点を中心とした座標の回転 65
図 4.9：変形したコントロールで取得できるマウスの移動量 66
図 4.10：物理スクロールサンプルの実行結果 68
図 4.11：サンプルアプリケーションの外観 81
図 4.12：Views フォルダに Behaviors フォルダを追加 82
図 4.13：添付ビヘイビアサンプルの実行結果 85
図 4.14：Person クラスが自動的に表形式で表示される 87
図 4.15：DisplayRowNumber 添付プロパティによる行番号の表示 90
図 4.16：ドラッグ操作でゴーストが表示される 91
図 4.17：ToggleButton をクリックするとそれぞれの装飾が表示される 104
図 4.18：TabControl でコンテンツを切り替えられる 112
図 4.19：表示方法をカスタマイズする必要がある 114
図 4.20：表示方法がカスタマイズされている 115
図 4.21：タブを追加するサンプル 117
図 4.22：タブを削除するサンプル 121
図 4.23：アセンブリリソースファイルを使用する（ない場合は自分で同名ファイルを追加する） 122
図 4.24：リソースを定義する 122
図 4.25：英語用のアセンブリリソースファイルを追加する 123
図 4.26：英語用のリソースを定義する 123
図 4.27：リソースで定義した文字列が表示されている 124
図 4.28：多言語サンプルの実行結果 126
図 4.29：ビルド結果 127
図 4.30：ホットキーサンプルの実行結果 141
図 4.31：未処理例外のサンプル 144
図 4.32：UI スレッド以外の未処理例外を捕捉するサンプル 147
図 4.33：ProvideValue() メソッドの戻り値が表示されている 151
図 4.34：Text プロパティの値がセットされている 152
図 4.35：Text プロパティの値がセットされている 153
図 4.36：Text プロパティが変化すると表示が更新される 155
図 5.1：プロジェクト設定でコード分析を有効化する 157
図 5.2：Initialize() メソッドが DerivedClass クラスのコンストラクタが実行される前に処理されている 163
図 6.1：新しいプロジェクトテンプレートを選択するダイアログ 164
図 6.2：Win32 アプリケーションウィザードの設定例 165
図 6.3：ファイル追加後のソリューションエクスプローラ 165
図 6.4：リンカーの設定でモジュール定義ファイルを指定する 166
図 6.5：プラットフォームツールセットの設定 166

図 6.6：C# から DLL の関数を呼び出した結果 169
図 6.7：C# から文字列を渡す DLL の関数を呼び出した結果 171
図 6.8：C# から文字列を渡して書き換えてもらう DLL の関数を呼び出した結果 173
図 6.9：C# から構造体を渡す DLL の関数を呼び出した結果 176
図 6.10：C# から構造体を書き換えてもらう DLL の関数を呼び出した結果 179
図 6.11：C# からメンバにポインタを含む構造体を書き換えてもらう DLL の関数を呼び出した結果 182
図 7.1：簡単に設定を保存/読込できる 183

表 目次

表 4.1：数値書式指定文字列 61
表 4.2：コントロールを変形させるためのクラス 62
表 6.1：呼び出し規約 168

コード 目次

コード 2.1：StartupUri プロパティの記述を削除した App.xaml 12
コード 2.2：App クラスのコードビハインド 12
コード 2.3：NotificatioinObject クラスの定義 13
コード 2.4：NotificationObject クラスが提供するメソッドの使用例 14
コード 2.5：DelegateCommand クラスの定義 16
コード 2.6：DelegateCommand クラスの使用例 17
コード 2.7：Gender 列挙体の定義 19
コード 2.8：Person クラスの定義 19
コード 3.1：System.BitConverter クラスで byte[] に変換する 21
コード 3.2：System.BitConverter クラスで基本データ型に変換する 22
コード 3.3：System.Text.Encoding クラスで文字コードに変換する 24
コード 3.4：System.Environments クラスを用いた MainViewModel 25
コード 3.5：System.Environment.SpecialFolder 列挙体を ComboBox で選択できる MainView 25
コード 3.6：デバッグ出力をファイルへ出力するようにするためにリスナーを追加する 27
コード 3.7：トレースレベルを考慮したログ出力のためのサンプルコード 27
コード 3.8：独自イベントを持つ Model 31
コード 3.9：イベントを購読する MainViewModel 31
コード 3.10：System.Environment.SpecialFolder 列挙体を ComboBox で選択できる MainView 33
コード 3.11：独自イベント引数 33
コード 3.12：独自イベント引数を使った独自イベントを持つ Model 34
コード 3.13：イベントを購読する MainViewModel 35
コード 3.14：ソースにリンクしたデバッグ出力をするサンプルコード 38
コード 3.15：Action デリゲートの使い方 40
コード 3.16：Action デリゲートをスタックする 41
コード 3.17：History クラスの定義 42
コード 3.18：ViewModel で元に戻す/やり直し機能を実装する一例 43
コード 3.19：#if～#endif でコードを囲む 48
コード 3.20：Conditional 属性の使用例 48
コード 3.21：複数の Conditional 属性は OR 条件となる 49
コード 3.22：Conditional 属性を使って AND 条件にする例 49
コード 3.23：SerialPort クラスでポートをオープンする 51
コード 3.24：SerialPort クラスでデータを受信する 52
コード 3.25：SerialPort クラスでデータを送信する 53
コード 4.1：Person プロパティを持つ ViewModel 55
コード 4.2：DesignInstance に対する ViewModel の指定 55
コード 4.3：TextBlock コントロールを持つ UserControl の例 57
コード 4.4：コードビハインド 57
コード 4.5：TextBlock コントロールを持つ UserControl の例 58
コード 4.6：UI 以外のコードでデザイン実行かどうかを判定する 58
コード 4.7：データの表示形式変更のための ViewModel 59
コード 4.8：StringFormat によってデータの表示形式を変更する 59
コード 4.9：人物データコレクションをプロパティに持つ ViewModel 67
コード 4.10：物理スクロールのサンプルコード 67
コード 4.11：AnimatedContainer コントロールの外観定義 69
コード 4.12：AnimatedContainer コントロールの内部実装定義 69
コード 4.13：コンストラクタでイベントを購読する 71
コード 4.14：ボタンのイベントハンドラで削除用のアニメーションを開始する 71
コード 4.15：アニメーション終了時に実行する DeletedCommand 依存関係プロパティの定義 72
コード 4.16：AnimatedContainer コントロールのサンプルコード 72
コード 4.17：サンプルアプリケーションの MainViewModel 78
コード 4.18：サンプルアプリケーションの MainView 79
コード 4.19：Grid.Row 添付プロパティの使用例 82
コード 4.20：IsEnabled 添付プロパティの作成例 83
コード 4.21：IsEnabled 添付プロパティを添付する 84
コード 4.22：SampleBehavior クラスのビヘイビア 84

コード 4.23：人物データコレクションを持つ MainViewModel 86
コード 4.24：人物データコレクションを DataGrid で表示する MainView 86
コード 4.25：DataGridBehavior 添付ビヘイビアの定義 87
コード 4.26：DataGridBehavior 添付ビヘイビアの使用例 90
コード 4.27：ゴースト表示するためのクラスの定義 91
コード 4.28：ゴースト表示するための添付ビヘイビアの定義 94
コード 4.29：ゴースト表示するための添付ビヘイビアの使用例 96
コード 4.30：任意のコントロールを装飾として表示するための AdornerCore クラスの定義 98
コード 4.31：AdornerTemplate 添付プロパティなどの定義 100
コード 4.32：添付プロパティ変更イベントハンドラの実装 101
コード 4.33：AdornerBehavior クラスのサンプル用 UI 103
コード 4.34：DataTemplateSelector クラスに対応した AdornerBehavior クラス 105
コード 4.35：TabControl の使用例 112
コード 4.36：コンテンツに対する ViewModel の定義 112
コード 4.37：ContentViewModel をコレクションとして保持する MainViewModel 113
コード 4.38： ViewModel のコレクションをデータバインディングする 114
コード 4.39：ItemTemplate プロパティと ContentTemplate プロパティを指定する 114
コード 4.40：AddContentCommand でコレクションに追加する 115
コード 4.41：AddContentCommand をデータバインディングする 116
コード 4.42：閉じるためのコマンドとイベントを定義する 117
コード 4.43：Closed イベントを購読してコレクション操作をおこなう 119
コード 4.44：AddContentCommand をデータバインディングする 120
コード 4.45：リソースを参照してテキストを表示する 123
コード 4.46：リソースを扱うクラスの定義 124
コード 4.47：リソースの参照先を変更する 125
コード 4.48：言語切り替えコマンドをデータバインドする 126
コード 4.49：言語切り替えコマンドを実装する 126
コード 4.50：二重起動させないためのサンプルコード 128
コード 4.51：Alt+Tab メニューのウィンドウ一覧に表示させないようにするコード例 130
コード 4.52：「デスクトップの表示」ボタンで最小化しないようにするための添付ビヘイビア 131
コード 4.53：「デスクトップの表示」ボタンで最小化しないようにするための添付ビヘイビアの使用例 134
コード 4.54：RegisterHotKey 関数と UnregisterHotKey 関数などを組み込む 135
コード 4.55：添付プロパティの定義 137
コード 4.56：RegisterHotKey 関数と UnregisterHotKey 関数などを組み込む 140
コード 4.57：ホットキーのサンプルに対する ViewModel 140
コード 4.58：HotKeyBehavior を組み込んだ View 141
コード 4.59：例外を発生させるコマンドを実装した MainViewModel 142
コード 4.60：例外を発生させるボタンを配置した MainView 143
コード 4.61：未処理例外発生イベントにメソッドを登録しておく 143
コード 4.62：UI スレッド以外の未処理例外発生イベントにメソッドを登録しておく 145
コード 4.63：UI スレッド以外で例外を発生させるコマンドを追加 146
コード 4.64：例外を発生させるボタンを配置した MainView 147
コード 4.65：TAP による非同期処理中の例外処理 148
コード 4.66：非同期処理中の例外発生をイベント通知する 148
コード 4.67：独自のマークアップ拡張 150
コード 4.68：独自のマークアップ拡張使用例 150
コード 4.69：独自のマークアップ拡張にプロパティを追加する 151
コード 4.70：独自のマークアップ拡張使用例 152
コード 4.71：独自のマークアップ拡張にプロパティを追加する 152
コード 4.72：独自のマークアップ拡張でプロパティ名を省略できる 153
コード 4.73：独自のマークアップ拡張にプロパティを追加する 153
コード 4.74：動作確認のために適当なプロパティを定義しておく 155
コード 4.75：TextBlock コントロールで表示する 155
コード 5.1：string.Format() メソッドによる SQL 構文の構築 158
コード 5.2：パラメータ付きコマンド文字列による CA2100 の回避 159
コード 5.3：using が入れ子になったコード 160
コード 5.4：Dispose() メソッドが複数回呼ばれないように stream 変数に null を代入する 160
コード 5.5：コンストラクタ内でオーバーライド可能なメソッドを呼び出している派生クラス 162

コード 6.1：ヘッダファイルで関数定義を宣言 167
コード 6.2：ソースファイルで関数を実装 167
コード 6.3：モジュール定義ファイルに公開する関数名をリストアップする 168
コード 6.4：C# から C/C++ DLL をインポートする 169
コード 6.5：モジュール定義ファイルに公開する関数名をリストアップする 170
コード 6.6：文字列を受け取る関数 170
コード 6.7：C# から C/C++ DLL をインポートする 170
コード 6.8：文字コードを明示的に指定する 171
コード 6.9：モジュール定義ファイルに公開する関数名をリストアップする 172
コード 6.10：文字列を受け渡す関数 172
コード 6.11：C# から C/C++ DLL をインポートする 172
コード 6.12：モジュール定義ファイルに公開する関数名をリストアップする 174
コード 6.13：構造体の定義 174
コード 6.14：構造体を受け取る関数 174
コード 6.15：C# から C/C++ DLL をインポートする 174
コード 6.16：モジュール定義ファイルに公開する関数名をリストアップする 177
コード 6.17：構造体の定義 177
コード 6.18：構造体を受け渡す関数 177
コード 6.19：C# から C/C++ DLL をインポートする 177
コード 6.20：モジュール定義ファイルに公開する関数名をリストアップする 180
コード 6.21：メンバにポインタを含む構造体の定義 180
コード 6.22：メンバにポインタを含む構造体を受け渡す関数 180
コード 6.23：C# から C/C++ DLL をインポートする 180

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 C# ならびに Windows Presentation Foundation（以降 WPF）の基本的な使い方などを共有し、WPF 開発技術力向上促進を目的としています。

1.2 注意

実際のアプリケーション開発では、MVVM パターンを意識した内部構造にすると思います。したがって、本書の中で示すサンプルコードも MVVM パターンを意識した内部構造を前提としたコードとなっています。また、本書ではこのような内部構造とするために、必ず「2 共通する設定およびクラス」で紹介している作業をおこなっていることを前提としています。その他、特に定義などがないようなクラスなどについても同章にて定義をしているので注意してください。また、本書では C# を使用することを前提としています。VB.NET を使用する方は適宜コードを読み替えてください。

1.3 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 Update5
・.NET Framework 4.6

2 共通する設定およびクラス

本章では、本書で共通して使用する設定やクラスについて紹介します。各章で特に定義されていないクラスや説明のないクラスは共通のクラスとして本章で定義しています。

2.1 MVVM パターンを意識した内部構造

本書では MVVM パターンを基にしたサンプルコードを掲載します。したがって、WPF アプリケーションに関するサンプルコードは必ず次のような作業を始めにおこなっています。

1. MainWindow.xaml および MainWindow.xaml.cs を削除する
2. "Views"、"ViewModels"、"Models" フォルダを追加する
3. "Views" フォルダに MainView ウィンドウクラスを追加する
4. "ViewModels" フォルダに MainViewModel クラスを追加する
5. App.xaml に記述されている StartupUri プロパティを削除する
6. App.xaml.cs で OnStartup() メソッドをオーバーライドする

以上の作業をおこなった後の内部構造は下図のようになります。

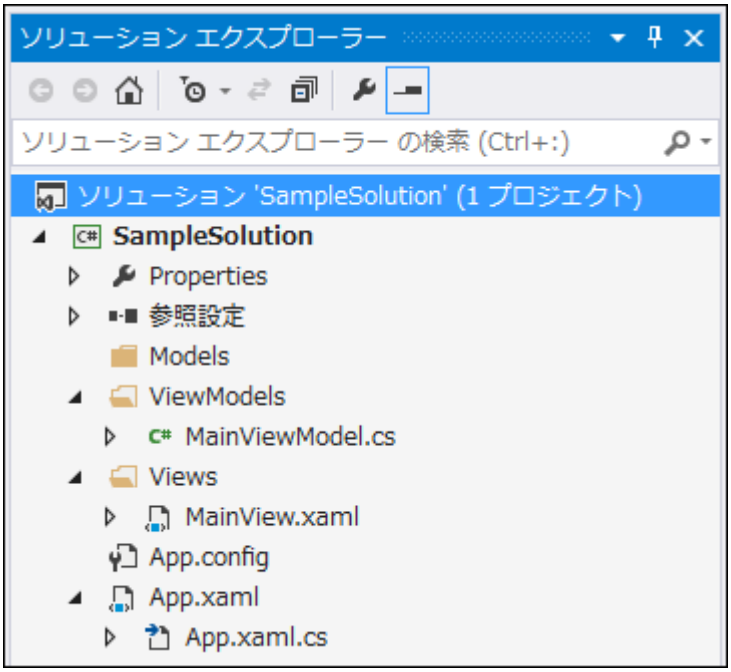


図 2.1：MVVM パターンを意識した内部構造

それぞれの作業について以降で説明します。

2.1.1 MainWindow.xaml および MainWindow.xaml.cs を削除する

MVVM パターンを考えたとき、一般的には一つの View に対して ViewModel は一つ、つまり一対一対応となります。このことから、ウィンドウを表すクラスは ○○View、それに対する "ViewModel" は ○○ViewModel と名付けると、対応付けがはっきりしてわかりやすくなります。

以上のことから、デフォルトで追加されている MainWindow クラスは邪魔になるため、削除します。

2.1.2 "Views"、"ViewModels"、"Models" フォルダを追加する

MVVM パターンに基づくということは、それぞれクラスが View、ViewModel、Model のいずれかに分類できるということになります。そのため、ファイル単位でも分類分けしやすいように、始めからフォルダでわけるようにします。

2.1.3 "Views" フォルダに MainView ウィンドウクラスを追加する

デフォルトで用意されていた MainWindow というウィンドウのクラスを削除してしまったため、このままではウィンドウが表示されません。そのため、MainView という名前のウィンドウのクラスを改めてデフォルトで表示されるウィンドウとして定義します。

ここでは "Views" フォルダに "MainView" という名前のクラスをウィンドウとして追加するだけになります。

2.1.4 "ViewModels" フォルダに MainViewModel クラスを追加する

MVVM パターンに基づいて開発をおこなう場合、View の DataContext として、対応する ViewModel を指定する必要があります。先ほど追加した MainView に対応する ViewModel として、MainViewModel クラスを追加します。この名前は「2.1.1 MainWindow.xaml および MainWindow.xaml.cs を削除する」で述べた命名規則に従っています。

また、すべての ViewModel は、自身のプロパティ値が変更されたとき、その変更を View に伝える必要があります。この変更を通知する機能は INotifyPropertyChanged インターフェースが提供しているため、ViewModel は必然的に INotifyPropertyChanged インターフェースを実装しなければなりません。これを実装するために、ViewModel は必ず NotificationObject クラスを基本クラスとするようにします。NotificationObject クラスについては「2.2.1 NotificationObject クラス」を参照してください。

2.1.5 App.xaml に記述されている StartupUri プロパティを削除する

デフォルトでは MainWindow.xaml で定義されていたウィンドウがアプリケーション起動時に表示されるように、App.xaml に StartupUri プロパティとして "MainWindow.xaml" が指定されていました。ここでは MainWindow は存在しないため、この記述を削除します。

また、起動時にウィンドウを表示するコードはコードビハインドである App.xaml.cs に記述するため、ここでは特に追加しません。

コード 2.1：StartupUri プロパティの記述を削除した App.xaml

App.xaml

```
1 <Application x:Class="Tips_Sample.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>
```

2.1.6 App.xaml.cs で OnStartup() メソッドをオーバーライドする

App クラスの OnStartup() メソッドは、アプリケーション起動時に呼ばれるメソッドです。ここで View に対する DataContext プロパティの設定と、Show() メソッド呼び出しによるウィンドウの表示をおこないます。実際のコードは次のようになります。

コード 2.2：App クラスのコードビハインド

App.xaml.cs

```
1 namespace Tips_Sample
2 {
3     using System.Windows;
4     using Tips_Sample.ViewModels;
```

```
 5      using Tips_Sample.Views;
 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  }
```

2.2 MVVM パターンのための基本的なクラス

ここでは、MVVM パターンを実現するために不可欠な INotifyPropertyChanged インターフェースを実装したクラスと、ICommand インターフェースを実装したクラスを定義します。

2.2.1 NotificationObject クラス

プロパティ変更を View 側に伝達するために、ViewModel または Model は INotifyPropertyChanged インターフェースを実装しなければなりません。しかし、このインターフェースを実装するために毎回同じコードを書くべきではないので、あらかじめこのインターフェースを実装したクラスとして NotificationObject クラスを次のように定義します。

コード 2.3：NotificatioinObject クラスの定義

NotificationObject.cs

```
 1  using System.ComponentModel;
 2  using System.Runtime.CompilerServices;
 3
 4  /// <summary>
 5  /// INotifyPropertyChanged インターフェースを実装した抽象クラスを表します。
 6  /// </summary>
 7  public abstract class NotificationObject : INotifyPropertyChanged
 8  {
 9      #region INotifyPropertyChanged のメンバ
10      /// <summary>
11      /// プロパティ変更時に発生します。
12      /// </summary>
13      public event PropertyChangedEventHandler PropertyChanged;
14      #endregion INotifyPropertyChanged のメンバ
15
16      /// <summary>
17      /// PropertyChanged イベントを発行します。
18      /// </summary>
19      /// <param name="propertyName">プロパティ名を指定します。</param>
20      protected void RaisePropertyChanged([CallerMemberName]string propertyName = null)
21      {
22          var h = this.PropertyChanged;
23          if (h != null) h(this, new PropertyChangedEventArgs(propertyName));
```

```
24      }
25
26      /// <summary>
27      /// プロパティ値変更ヘルパです。
28      /// </summary>
29      /// <typeparam name="T">プロパティの型を表します。</typeparam>
30      /// <param name="target">変更するプロパティの実体を指定します。</param>
31      /// <param name="value">変更後の値を指定します。</param>
32      /// <param name="propertyName">プロパティ名を指定します。</param>
33      /// <returns>プロパティ値に変更があった場合に true を返します。</returns>
34      protected bool SetProperty<T>(ref T target, T value, [CallerMemberName]string
    propertyName = null)
35      {
36          if (Equals(target, value)) { return false; }
37          target = value;
38          RaisePropertyChanged(propertyName);
39          return true;
40      }
41  }
```

INotifyPropertyChanged インターフェースのメンバは PropertyChanged イベントだけですが、このイベントを発生させるためのメソッドを 2 つ用意しています。

RaisePropertyChanged() メソッドは、変更があったプロパティ名を指定してコールすることで、そのプロパティが変更したことを View に伝えます。プロパティ名を省略した場合は、すべてのプロパティが変更されたとして伝えられます。ただし、CallberMemberName 属性をサポートするコンパイラによってコンパイルした場合、プロパティ名を省略してもそのプロパティが変更されたとして通知されます。

SetProperty() メソッドは、プロパティ値を変更するためのヘルパです。上記の RaisePropertyChanged() メソッドは、プロパティ値をセットするときは必ずコールされるメソッドになります。したがって、プロパティ値をセットするコードと一緒にしてしまえば、プロパティをセットするコードと RaisePropertyChanged() メソッドをコールするコードが一つにまとまります。

これらのメソッドの使用例は次のようになります。

コード 2.4：NotificationObject クラスが提供するメソッドの使用例

SampleViewModel.cs

```
1   /// <summary>
2   /// プロパティ変更の使用例を示すためのクラスを表します。
3   /// </summary>
4   public class SampleViewModel : NotificationObject
5   {
6       private int _id;
7       /// <summary>
8       /// 識別子を取得または設定します。
9       /// </summary>
10      public int ID
11      {
12          get { return this._id; }
13          set
14          {
15              // プロパティ値に変更があるかどうかを確認します。
16              if (this._id != value)
17              {
18                  this._id = value;
19                  // 変更があった場合に変更通知をおこないます。
20                  RaisePropertyChanged("ID");
21              }
```

```
22          }
23      }
24
25      private int _age;
26      /// <summary>
27      /// 年齢を取得または設定します。
28      /// </summary>
29      public int Age
30      {
31          get { return this._age; }
32          set { SetProperty(ref this._age, value); }
33      }
34
35      private string _firstName;
36      /// <summary>
37      /// 名を取得または設定します。
38      /// </summary>
39      public string FirstName
40      {
41          get { return this._firstName; }
42          set
43          {
44              // プロパティ値に変更があった場合に true が返ります。
45              if (SetProperty(ref this._firstName, value))
46              {
47                  // 同時に FullName プロパティの変更通知をおこないます。
48                  RaisePropertyChanged("FullName");
49              }
50          }
51      }
52
53      private string _lastName;
54      /// <summary>
55      /// 姓を取得または設定します。
56      /// </summary>
57      public string LastName
58      {
59          get { return this._lastName; }
60          set
61          {
62              // プロパティ値に変更があった場合に true が返ります。
63              if (SetProperty(ref this._lastName, value))
64              {
65                  // 同時に FullName プロパティの変更通知をおこないます。
66                  RaisePropertyChanged("FullName");
67              }
68          }
69      }
70
71      /// <summary>
72      /// 氏名を取得します。
73      /// </summary>
74      public string FullName
75      {
76          get { return string.Format("{0} {1}", this.FirstName, this.LastName); }
77      }
78  }
```

SetProperty() メソッドを使わずにプロパティ値の変更をおこなう場合、ID プロパティの set アクセサのように、現在の値と比較して、異なる場合に新しい値を代入し、その変更通知を RaisePropertyChanged() メソッドでおこなわなければいけません。

これに対し、SetProperty() メソッドは、Age プロパティの set アクセサのように、プロパティ値の変更とその変更通知をたったの一行で書くことができるようになります。通常のプロパティはこの Age プロパティのような書き方をお勧めします。

実際には、FullName プロパティのように、他のプロパティ値に依存して変更されるプロパティもあります。この場合、他のプロパティ値が変更されたときに、FullName プロパティの変更通知をおこなわなければなりません。このような場合でも、SetProperty() メソッドが有用となります。FirstName プロパティや LastName プロパティの set アクセサのように、SetProperty() メソッドの戻り値によって、自分のプロパティ値が変更された場合は、その変更が反映されるように RaisePropertyChanged() メソッドの引数に "FullName" を与え、自分の変更通知の後に FullName プロパティの変更通知もおこなうようにできます。

2.2.2 DelegateCommand クラス

ボタンの Command プロパティなどとデータバインディングするために、ICommand インターフェースを実装した DelegateCommand クラスを次のように定義します。

コード 2.5：DelegateCommand クラスの定義

DelegateCommand.cs

```
 1  using System;
 2  using System.Windows.Input;
 3
 4  /// <summary>
 5  /// ICommand インターフェースを実装したコマンド用のクラスを表します。
 6  /// </summary>
 7  public class DelegateCommand : ICommand
 8  {
 9      #region private フィールド
10      /// <summary>
11      /// コマンドを実行するためのメソッドを保持します。
12      /// </summary>
13      private Action<object> _execute;
14
15      /// <summary>
16      /// コマンドの実行可能性を判別するためのメソッドを保持します。
17      /// </summary>
18      private Func<object, bool> _canExecute;
19      #endregion private フィールド
20
21      #region コンストラクタ
22      /// <summary>
23      /// 新しいインスタンスを生成します。
24      /// </summary>
25      /// <param name="execute">コマンドを実行するためのメソッドを指定します。</param>
26      public DelegateCommand(Action<object> execute)
27          : this(execute, null)
28      {
29      }
30
31      /// <summary>
32      /// 新しいインスタンスを生成します。
33      /// </summary>
34      /// <param name="execute">コマンドを実行するためのメソッドを指定します。</param>
35      /// <param name="canExecute">コマンドの実行可能性を判別するためのメソッドを指定します。
36  </param>
37      public DelegateCommand(Action<object> execute, Func<object, bool> canExecute)
38      {
```

```
39          this._execute = execute;
40          this._canExecute = canExecute;
41      }
42      #endregion コンストラクタ
43
44      #region ICommand のメンバ
45      /// <summary>
46      /// コマンドの実行可能性を判別します。
47      /// </summary>
48      /// <param name="parameter">この処理に対するパラメータを指定します。</param>
49      /// <returns>コマンドが実行可能である場合に true を返します。</returns>
50      public bool CanExecute(object parameter)
51      {
52          return this._canExecute != null ? this._canExecute(parameter) : true;
53      }
54
55      /// <summary>
56      /// コマンドの実行可能性が変更されたときに発生します。
57      /// </summary>
58      public event System.EventHandler CanExecuteChanged
59      {
60          add { CommandManager.RequerySuggested += value; }
61          remove { CommandManager.RequerySuggested -= value; }
62      }
63
64      /// <summary>
65      /// コマンドを実行します。
66      /// </summary>
67      /// <param name="parameter">この処理に対するパラメータを指定します。</param>
68      public void Execute(object parameter)
69      {
70          if (this._execute != null)
71              this._execute(parameter);
72      }
73      #endregion ICommand のメンバ
74  }
```

実際に処理する内容や、実行可能かどうかを判別するための処理はコマンドによって異なるため、これらの処理を直接このクラス内で記述するのではなく、private なフィールドである _execute と _canExecute 変数で保持します。

DelegateCommand クラスの使用例は次のようになります。

コード 2.6：DelegateCommand クラスの使用例

SampleViewModel.cs

```
1   /// <summary>
2   /// DelegaeCommand クラスの使用例を示すためのクラスを表します。
3   /// </summary>
4   public class SampleViewModel : NotificationObject
5   {
6       private double _lhs;
7       /// <summary>
8       /// 割られる数を取得または設定します。
9       /// </summary>
10      public double Lhs
11      {
12          get { return this._lhs; }
13          set { SetProperty(ref this._lhs, value); }
```

```
14      }
15
16      private double _rhs;
17      /// <summary>
18      /// 割る数を取得または設定します。
19      /// </summary>
20      public double Rhs
21      {
22          get { return this._rhs; }
23          set { SetProperty(ref this._rhs, value); }
24      }
25
26      private double _answer;
27      /// <summary>
28      /// 計算結果を取得します。
29      /// </summary>
30      public double Answer
31      {
32          get { return this._answer; }
33          private set { SetProperty(ref this._answer, value); }
34      }
35
36      private DelegateCommand _divideCommand;
37      /// <summary>
38      /// 割り算コマンドを取得します。
39      /// </summary>
40      public DelegateCommand DivideCommand
41      {
42          get
43          {
44              return this._divideCommand ?? (this._divideCommand = new DelegateCommand(
45              _ =>
46              {
47                  // ここにはコマンドとして実行する処理を記述します。
48
49                  // 割り算をおこないます。
50                  this.Answer = this.Lhs / this.Rhs;
51              },
52              _ =>
53              {
54                  // ここには実行可能性を判別する処理を記述します。
55
56                  // 割る数がゼロ以外のときに割り算が実行できます。
57                  return this.Rhs != 0.0;
58              }));
59          }
60      }
61  }
```

DelegateCommand クラスのコンストラクタでは、第一引数にコマンドとして実行する処理を、第二引数としてコマンドの実行可能判別をおこなう処理を指定します。使用例のように、ラムダ式による表現を用いる方法が一般的に使われます。常に実行可能であるコマンドの場合は、コンストラクタの第二引数を省略することができます。

上記のラムダ式では入力引数を使用しないので "_" という変数名としていますが、CommandParaneter を受け取る場合は、適宜変数名を変更したほうが良いでしょう。

2.3 Person クラス

ここでは、サンプルクラスとして良く扱う人物データを表す Person クラスを定義します。

2.3.1 Gender 列挙体

人物の性別を表すための列挙体として、Gender 列挙体を次のように定義します。

コード 2.7：Gender 列挙体の定義

ViewModelBase.cs

```
1  /// <summary>
2  /// 性別を表します。
3  /// </summary>
4  public enum Gender
5  {
6      /// <summary>
7      /// 性別不明を表します。
8      /// </summary>
9      Unknown = 0,
10
11     /// <summary>
12     /// 男性を表します。
13     /// </summary>
14     Male,
15
16     /// <summary>
17     /// 女性を表します。
18     /// </summary>
19     Female,
20 }
```

2.3.2 Person クラス

人物データを表す Person クラスを次のように定義します。Person クラスはコレクションデータとしても良く使われるため、NotificationObject クラスから派生することで、自身のプロパティ変更を通知できるようにしています。

コード 2.8：Person クラスの定義

ViewModelBase.cs

```
1  /// <summary>
2  /// 人物データを表します。
3  /// </summary>
4  public class Person : NotificationObject
5  {
6      private string _name;
7      /// <summary>
8      /// 氏名を取得または設定します。
9      /// </summary>
10     public string Name
11     {
12         get { return this._name; }
13         set { SetProperty(ref this._name, value); }
14     }
15
16     private int _age;
17     /// <summary>
```

```
18      /// 年齢を取得または設定します。
19      /// </summary>
20      public int Age
21      {
22          get { return this._age; }
23          set { SetProperty(ref this._age, value); }
24      }
25
26      private Gender _gender;
27      /// <summary>
28      /// 性別を取得または設定します。
29      /// </summary>
30      public Gender Gender
31      {
32          get { return this._gender; }
33          set { SetProperty(ref this._gender, value); }
34      }
35
36      private bool _isAuthenticated;
37      /// <summary>
38      /// 認証済みかどうかを取得または設定します。
39      /// </summary>
40      public bool IsAuthenticated
41      {
42          get { return this._isAuthenticated; }
43          set { SetProperty(ref this._isAuthenticated, value); }
44      }
45  }
```

3 C# によるあれこれ

本章では、C# で実現するための色々な技術を紹介します。

3.1 byte[] と int などの基本データ型を相互に変換する (Tips_BitConverter)

複数のバイトデータ byte[] から多バイトデータを生成するには、System.BitConverter クラスを使います。このとき、Intel の x86 などはリトルエンディアンを採用しているため、byte[] 配列の要素の順番には気を付ける必要があります。

3.1.1 int などのデータ型から byte[] に変換する

基本データ型から byte[] に変換するときは GetBytes() メソッドを使います。

コード 3.1：System.BitConverter クラスで byte[] に変換する

Program.cs

```
 1 namespace Tips_BitConverter
 2 {
 3     using System;
 4     using System.Linq;
 5
 6     class Program
 7     {
 8         static void Main(string[] args)
 9         {
10             byte[] bytes;
11             int int32;
12
13             // bytes = { 0x78, 0x56, 0x34, 0x12 }
14             int32 = 0x12345678;
15             bytes = BitConverter.GetBytes(int32);
16             PrintByteArray(bytes);
17
18             // bytes = { 0x12, 0x34, 0x56, 0x78 }
19             int32 = 0x78563412;
20             bytes = BitConverter.GetBytes(int32);
21             PrintByteArray(bytes);
22
23             Console.ReadKey();
24         }
25
26         static void PrintByteArray(byte[] bytes)
27         {
28             var str = bytes.Select(x => "0x" + x.ToString("X2"));
29             Console.WriteLine("Bytes = {{ {0} }}", string.Join(", ", str));
30         }
31     }
32 }
```

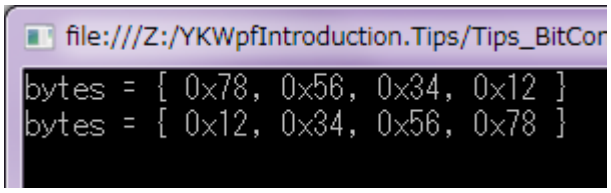


図 3.1：byte[] に変換されている

実行結果を見てもわかるように、リトルエンディアンの場合は byte[] に変換した後の要素の順番が入れ替わります。

GetBytes() メソッドの入力引数は short や long など他のデータ型も指定することができるので、とにかく byte[] に変換したいときはこのメソッドを使うことができます。

3.1.2 byte[] から int などのデータ型に変換する

byte[] から基本データ型に変換するときは、例えば int 型の場合は ToInt32() メソッド、long 型の場合は ToInt64() メソッドを使います。

コード 3.2：System.BitConverter クラスで基本データ型に変換する

Program.cs

```
1  namespace Tips_BitConverter
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              byte[] bytes;
10             int int32;
11
12             // 0x78563412
13             bytes = new byte[] { 0x12, 0x34, 0x56, 0x78 };
14             int32 = BitConverter.ToInt32(bytes, 0);
15             Console.WriteLine("int32 = 0x{0}", int32.ToString("X8"));
16
17             // 0x12345678
18             bytes = new byte[] { 0x78, 0x56, 0x34, 0x12 };
19             int32 = BitConverter.ToInt32(bytes, 0);
20             Console.WriteLine("int32 = 0x{0}", int32.ToString("X8"));
21
22             Console.ReadKey();
23         }
24     }
25 }
```

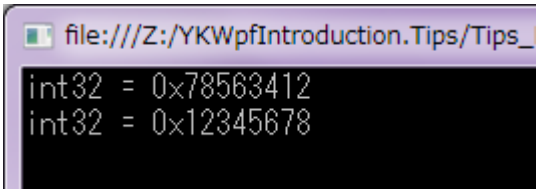


図 3.2：int に変換されている

実行結果から見てもわかるように、リトルエンディアンの場合は元となる byte[] の要素の順番とは逆の順番になります。

他にも ToBoolean()、ToChar() などのメソッドも用意されています。

3.2 string 文字列を文字コードに変換する (Tips_Encoding)

文字列を ASCII コードや Unicode コードに変換するには System.Text.Encoding クラスを使います。

コード 3.3：System.Text.Encoding クラスで文字コードに変換する

Program.cs

```
 1 namespace Tips_Encoding
 2 {
 3     using System;
 4     using System.Linq;
 5     using System.Text;
 6
 7     class Program
 8     {
 9         static void Main(string[] args)
10         {
11             var str = "0123あいう";
12             byte[] bytes;
13
14             Console.WriteLine("str = {0}", str);
15             bytes = Encoding.ASCII.GetBytes(str);
16             PrintByteArray(bytes);
17
18             bytes = Encoding.Unicode.GetBytes(str);
19             PrintByteArray(bytes);
20
21             bytes = Encoding.UTF8.GetBytes(str);
22             PrintByteArray(bytes);
23
24             Console.ReadKey();
25         }
26
27         static void PrintByteArray(byte[] bytes)
28         {
29             var str = bytes.Select(x => "0x" + x.ToString("X2"));
30             Console.WriteLine("Bytes = {{ {0} }}", string.Join(", ", str));
31         }
32     }
33 }
```

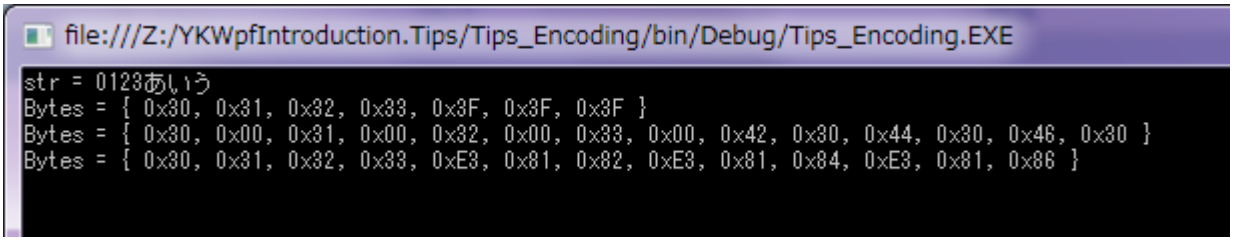


図 3.3：それぞれの文字コードに変換されている

ちなみに ASCII コードには全角文字はないので、0x3F すなわち '?' になるようです。

3.3 特殊ディレクトリのフルパスを取得する (Tips_SpecialFolderPath)

ファイルの読み書きをするとき、実行ファイルが実行されているカレントディレクトリや、現在のユーザーが使用しているデスクトップなどのフルパスを参照することは多々あります。ここではそういった特殊ディレクトリの取得方法を紹介します。

コード 3.4：System.Environments クラスを用いた MainViewModel

MainViewModel.cs

```
1  namespace Tips_SpecialFolderPath.ViewModels
2  {
3      using System;
4
5      public class MainViewModel : NotificationObject
6      {
7          /// <summary>
8          /// 特殊フォルダを示す配列を取得します。
9          /// </summary>
10         public Array SpecialFolders
11         {
12             get { return Enum.GetValues(typeof(Environment.SpecialFolder)); }
13         }
14
15         private Environment.SpecialFolder _specialFolder;
16         /// <summary>
17         /// 選択された特殊フォルダを取得または設定します。
18         /// </summary>
19         public Environment.SpecialFolder SpecialFolder
20         {
21             get { return this._specialFolder; }
22             set
23             {
24                 if (SetProperty(ref this._specialFolder, value))
25                 {
26                     RaisePropertyChanged("FullPath");
27                 }
28             }
29         }
30
31         /// <summary>
32         /// 特殊フォルダのフルパスを取得します。
33         /// </summary>
34         public string FullPath
35         {
36             get { return Environment.GetFolderPath(this.SpecialFolder); }
37         }
38     }
39 }
```

コード 3.5：System.Environment.SpecialFolder 列挙体を ComboBox で選択できる MainView

MainView.xaml

```
1  <Window x:Class="Tips_SpecialFolderPath.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:sys="clr-namespace:System;assembly=mscorlib"
5          Title="MainView" Height="100" Width="400">
6      <StackPanel>
7          <TextBox Text="{Binding FullPath, Mode=OneWay}" IsReadOnly="True" />
8          <ComboBox ItemsSource="{Binding SpecialFolders}"
```

```
 9                 SelectedItem="{Binding SpecialFolder}"
10                 Margin="0,10,0,0" />
11     </StackPanel>
12 </Window>
```

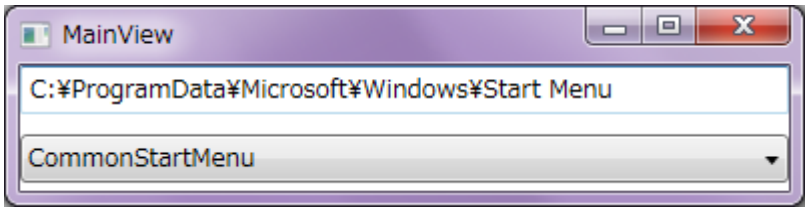


図 3.4：特殊フォルダのフルパスが表示される

System.Environment.GetFolderPath() メソッドを使用すると、デスクトップなどの特殊フォルダへのフルパスを簡単に取得することができます。

3.4 ログファイルを出力する (Tips_DebugTrace)

System.Diagnostics.Debug クラスや System.Diagnostics.Trace クラスでは、メッセージの出力先は出力ウィンドウが既定値となっています。出力先を追加することで、メッセージをファイルとして保存することができます。

3.4.1 Debug クラスによるログファイル出力

Debug クラスはプリプロセスシンボル DEBUG が定義されていない場合はコンパイル対象にはならないため、Release 構成で出力されるアセンブリにはそのコードが含まれないことになります。したがって、あくまでもデバッグ情報として残したいときなどに使用します。

そもそもデバッグ出力をおこなうと出力ウィンドウに表示されるのは、デバッグ出力を監視するリスナーに既定値として設定されているからです。実はこのリスナーはいくつも登録することができます。つまり、このリスナーとしてファイルに保存するリスナーを追加することでログファイルへ出力することができるようになります。登録されているリスナーは Debug. Listeners プロパティで確認することができます。コレクション操作が可能なので、登録追加/解除も簡単におこなえます。ただし、出力ウィンドウへ出力するためのリスナーも含まれているため、Debug.Listeners.Clear() メソッドを実行してしまうと、出力ウィンドウにすら出力されなくなってしまうので注意が必要です。

ファイル出力するためのリスナーは TextWriterTraceListener クラスである必要があります。例えば log.txt というテキストファイルに出力するリスナーを追加するには次のようにします。

コード 3.6：デバッグ出力をファイルへ出力するようにするためにリスナーを追加する

Program.cs

```
1 var fileListener = new TextWriterTraceListener("log.txt", "LogFile");
2 Debug.Listeners.Add(fileListener);
3 Debug.AutoFlush = true;
```

この 3 行を実行すると、Debug.WriteLine() メソッドなどでデバッグ出力をおこなうと、自動的に log.txt テキストファイルにメッセージが追加されていくようになります。3 行目の Debug.AutoFlush プロパティを false にすると、デバッグ出力をおこなってもただちにファイルへ出力せず、Debug.Flush() メソッドを呼び出すとこれまで書き込んでいなかった分のデバッグ出力がファイルへ出力されるようになります。

3.4.2 Trace クラスによるログファイル出力

Trace クラスはプリプロセスシンボル TRACE が定義されていない場合はコンパイル対象にはなりませんが、Release ビルド構成でもシンボル TRACE は定義されるため、リリース後のアプリケーションでもログ情報を残したいときなどに使用します。

Trace クラスでリスナーを追加する方法も前節の Debug クラスによる追加方法と同じで、Trace.Listeners プロパティを用い、TextWriterTraceListener クラスを追加するだけです。さらに、実は Debug.Listeners プロパティも Trace.Listeners プロパティも実体は同じインスタンスを参照していますので、どちらを操作しても同じ結果となります。

Trace クラスには Debug クラスと同様の Write()、WriteLine() メソッドの他に、TraceInformation()、TraceWarning()、TraceError() メソッドというものがあります。それぞれ情報、警告、エラーのためのメッセージ出力をおこなうもので、ただ WriteLine() メソッドでメッセージを出力するよりもトレースレベルなどの情報が付加された形でログが残せます。
ただし、その書式が決まってしまっているので、独自にカスタマイズした書式でログを残したい場合には向いていません。

独自にカスタマイズできるように工夫した DebugTrace クラスをサンプルコードとして掲載します。

コード 3.7：トレースレベルを考慮したログ出力のためのサンプルコード

Program.cs

```
1 namespace YKNlsOfflineCalculation
```

```
 2  {
 3      using System.Diagnostics;
 4      using System.IO;
 5      using System.Runtime.CompilerServices;
 6
 7      /// <summary>
 8      /// デバッグ用トレースをおこなうクラスを表します。
 9      /// </summary>
10      public static class DebugTrace
11      {
12          /// <summary>
13          /// ログファイル名
14          /// </summary>
15          private static readonly string _logFilePath = "log.txt";
16
17          /// <summary>
18          /// ログファイル出力リスナー名
19          /// </summary>
20          private static readonly string _listenerName = "LogFile";
21
22          /// <summary>
23          /// 初期化時にログファイルをクリアする
24          /// </summary>
25          private static readonly bool _clearLogFileAtInitializing = true;
26
27          #region 初期化
28          /// <summary>
29          /// 静的なコンストラクタです。
30          /// </summary>
31          static DebugTrace()
32          {
33              Initialize();
34          }
35
36          /// <summary>
37          /// 静的な初期化をおこないます。
38          /// Debug 構成のみ有効です。
39          /// </summary>
40          [Conditional("DEBUG")]
41          private static void Initialize()
42          {
43              if (!string.IsNullOrWhiteSpace(_logFilePath))
44              {
45                  if (_clearLogFileAtInitializing && File.Exists(_logFilePath))
46                  {
47                      File.Delete(_logFilePath);
48                  }
49
50                  var fileListener = new TextWriterTraceListener(_logFilePath,
    _listenerName);
51                  Trace.Listeners.Add(fileListener);
52                  Trace.AutoFlush = true;
53              }
54          }
55          #endregion 初期化
56
57          #region 公開メソッド
58          /// <summary>
```

```
59          /// 情報メッセージを出力します。
60          /// </summary>
61          /// <param name="message">メッセージを指定します。</param>
62          /// <param name="name">メソッド名を指定します。</param>
63          /// <param name="filePath">ソースファイルのフルパスを指定します。</param>
64          /// <param name="lineNumber">行番号を指定します。</param>
65          [Conditional("DEBUG")]
66          public static void TraceInformation(string message, [CallerMemberName]string name
    = null, [CallerFilePath]string filePath = null, [CallerLineNumber]int lineNumber = 0)
67          {
68              WriteLine(message, "Information", name, filePath, lineNumber);
69          }
70
71          /// <summary>
72          /// 警告メッセージを出力します。
73          /// </summary>
74          /// <param name="message">メッセージを指定します。</param>
75          /// <param name="name">メソッド名を指定します。</param>
76          /// <param name="filePath">ソースファイルのフルパスを指定します。</param>
77          /// <param name="lineNumber">行番号を指定します。</param>
78          [Conditional("DEBUG")]
79          public static void TraceWarning(string message, [CallerMemberName]string name =
    null, [CallerFilePath]string filePath = null, [CallerLineNumber]int lineNumber = 0)
80          {
81              WriteLine(message, "Warning", name, filePath, lineNumber);
82          }
83
84          /// <summary>
85          /// エラーメッセージを出力します。
86          /// </summary>
87          /// <param name="message">メッセージを指定します。</param>
88          /// <param name="name">メソッド名を指定します。</param>
89          /// <param name="filePath">ソースファイルのフルパスを指定します。</param>
90          /// <param name="lineNumber">行番号を指定します。</param>
91          [Conditional("DEBUG")]
92          public static void TraceError(string message, [CallerMemberName]string name = null,
    [CallerFilePath]string filePath = null, [CallerLineNumber]int lineNumber = 0)
93          {
94              WriteLine(message, "Error", name, filePath, lineNumber);
95          }
96          #endregion 公開メソッド
97
98          #region ヘルパ
99          /// <summary>
100         /// メッセージを出力するためのヘルパです。
101         /// </summary>
102         /// <param name="message">メッセージを指定します。</param>
103         /// <param name="type">メッセージの種別を指定します。</param>
104         /// <param name="name">メソッド名を指定します。</param>
105         /// <param name="filePath">ソースファイルのフルパスを指定します。</param>
106         /// <param name="lineNumber">行番号を指定します。</param>
107         [Conditional("DEBUG")]
108         private static void WriteLine(string message, string type, [CallerMemberName]string
    name = null, [CallerFilePath]string filePath = null, [CallerLineNumber]int lineNumber =
    0)
109         {
110             var str = string.Format("{0}({1}): {4} :{2} {3}", filePath, lineNumber, name !=
    null ? " [" + name + "]" : "", message, type);
```

```
111                 Trace.WriteLine(str);
112             }
113             #endregion ヘルパ
114         }
115 }
```

3.5 独自イベントの作成方法 (Tips_CustomEvent)

独自クラスを作成していると、イベントを飛ばしたくなる時があります。C# では event 修飾子を用いてイベントハンドラのデリゲートを定義する必要があります。

コード 3.8：独自イベントを持つ Model

Settings.cs

```
 1 namespace Tips_CustomEvent.Models
 2 {
 3     using System;
 4 
 5     /// <summary>
 6     /// 設定を表します。
 7     /// </summary>
 8     public class Settings
 9     {
10         private string _name;
11         /// <summary>
12         /// 名前を取得または設定します。
13         /// </summary>
14         public string Name
15         {
16             get { return this._name; }
17             set
18             {
19                 if (this._name != value)
20                 {
21                     this._name = value;
22                     RaiseNameChanged();
23                 }
24             }
25         }
26 
27         /// <summary>
28         /// Name プロパティ変更時に発生します。
29         /// </summary>
30         public event EventHandler<EventArgs> NameChanged;
31 
32         /// <summary>
33         /// NameChanged イベントを発行します。
34         /// </summary>
35         protected virtual void RaiseNameChanged()
36         {
37             var h = this.NameChanged;
38             if (h != null) h(this, EventArgs.Empty);
39         }
40     }
41 }
```

このサンプルコードでは、NameChanged という名前のイベントを持っており、Name プロパティの set アクセサ内で、プロパティ値に変更があったときに RaiseNameChanged() メソッドをコールすることで NameChanged イベントを発行しています。

このイベントを購読するように ViewModel を作って見ましょう。

コード 3.9：イベントを購読する MainViewModel

MainViewModel.cs

```
1 namespace Tips_CustomEvent.ViewModels
```

```
 2  {
 3      using System;
 4      using Tips_CustomEvent.Models;
 5
 6      public class MainViewModel : NotificationObject
 7      {
 8          private string _name;
 9          /// <summary>
10          /// 名前を取得または設定します。
11          /// </summary>
12          public string Name
13          {
14              get { return this._name; }
15              set
16              {
17                  if (SetProperty(ref this._name, value))
18                  {
19                      this._settings.Name = this._name;
20                  }
21              }
22          }
23
24          private bool _isSubscribed;
25          /// <summary>
26          /// イベント購読するかどうかを取得または設定します。
27          /// </summary>
28          public bool IsSubscribed
29          {
30              get { return this._isSubscribed; }
31              set
32              {
33                  if (SetProperty(ref this._isSubscribed, value))
34                  {
35                      if (this._isSubscribed)
36                      {
37                          this._settings.NameChanged += Settings_NameChanged;
38                      }
39                      else
40                      {
41                          this._settings.NameChanged -= Settings_NameChanged;
42                      }
43                  }
44              }
45          }
46
47          private int _counter;
48          /// <summary>
49          /// カウンタ値を取得します。
50          /// </summary>
51          public int Counter
52          {
53              get { return this._counter; }
54              private set { SetProperty(ref this._counter, value); }
55          }
56
57          /// <summary>
58          /// 設定を保持します。
59          /// </summary>
```

```
60          private Settings _settings = new Settings();
61
62          /// <summary>
63          /// 設定の Name プロパティ変更イベントハンドラ
64          /// </summary>
65          /// <param name="sender">イベント発行元</param>
66          /// <param name="e">イベント引数</param>
67          private void Settings_NameChanged(object sender, EventArgs e)
68          {
69              this.Counter++;
70          }
71      }
72  }
```

イベントの購読では、37 行目のように "+=" 演算子で対応するイベントハンドラを登録します。逆に購読を解除するときは、41 行目のように "-=" 演算子でおこないます。

イベントハンドラの入力引数は、イベントを定義したデリゲートに依存します。上記のサンプルでは EventArgs クラスを引数としたデリゲートとして宣言しているため、67 行目のように、イベント発行元を示す object 型に続いて EventArgs 型を入力引数としています。

この ViewModel を使用した View のサンプルを次のようにします。

コード 3.10：System.Environment.SpecialFolder 列挙体を ComboBox で選択できる MainView

MainView.xaml

```
1  <Window x:Class="Tips_CustomEvent.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="100" Width="200">
5      <StackPanel>
6          <TextBlock Text="{Binding Counter}" />
7          <TextBox Text="{Binding Name, UpdateSourceTrigger=PropertyChanged}" />
8          <CheckBox IsChecked="{Binding IsSubscribed}" Content="イベント購読する" />
9      </StackPanel>
10 </Window>
```

チェックボックスにチェックを入れるとイベントが購読されるため、文字入力の度にカウンタがインクリメントされていきます。チェックを外すとイベント購読が解除されるため、文字を入力してもカウンタは変化しません。

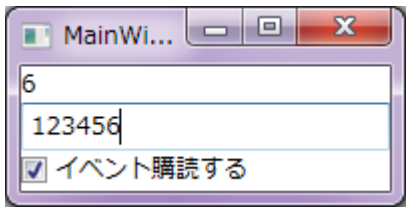


図 3.5：イベント発生回数がカウントされる

上記のサンプルでは Name プロパティが変更されたことがわかっても、Name プロパティがどのように変更されたのかが伝わりません。そこで、入力引数である EventArgs 型をやめて、次のようなクラスを入力引数にすることを考えます。

コード 3.11：独自イベント引数

ValueChangedEventArgs.cs

```
1  namespace Tips_CustomEvent.Models
2  {
```

```
 3      using System;
 4
 5      /// <summary>
 6      /// 値を変更したときのイベント引数を表します。
 7      /// </summary>
 8      /// <typeparam name="T">変更した値の型を表します。</typeparam>
 9      public class ValueChangedEventArgs<T> : EventArgs
10      {
11          /// <summary>
12          /// 新しいインスタンスを生成します。
13          /// </summary>
14          /// <param name="oldValue">変更前の値を指定します。</param>
15          /// <param name="newValue">変更後の値を指定します。</param>
16          public ValueChangedEventArgs(T oldValue, T newValue)
17          {
18              this.OldValue = oldValue;
19              this.NewValue = newValue;
20          }
21
22          /// <summary>
23          /// 変更前の値を取得します。
24          /// </summary>
25          public T OldValue { get; private set; }
26
27          /// <summary>
28          /// 変更後の値を取得します。
29          /// </summary>
30          public T NewValue { get; private set; }
31      }
32  }
```

イベント引数なので、必ず System.EventArgs クラスから派生するようにします。変更前の値と変更後の値をプロパティとして持つイベント引数とすることで、値がどのように変化したかが伝わるようになります。

このイベント引数を入力引数としたデリゲートに変更するために、先ほど定義した Settings クラスを次のように変更します。

コード 3.12：独自イベント引数を使った独自イベントを持つ Model

Settings.cs

```
 1  namespace Tips_CustomEvent.Models
 2  {
 3      using System;
 4
 5      /// <summary>
 6      /// 設定を表します。
 7      /// </summary>
 8      public class Settings
 9      {
10          private string _name;
11          /// <summary>
12          /// 名前を取得または設定します。
13          /// </summary>
14          public string Name
15          {
16              get { return this._name; }
17              set
18              {
19                  if (this._name != value)
```

```
20                 {
21                     var oldName = this._name;
22                     this._name = value;
23                     RaiseNameChanged(oldName, this._name);
24                 }
25             }
26         }
27 
28         /// <summary>
29         /// Name プロパティ変更時に発生します。
30         /// </summary>
31         public event EventHandler<ValueChangedEventArgs<string>> NameChanged;
32 
33         /// <summary>
34         /// NameChanged イベントを発行します。
35         /// </summary>
36         protected virtual void RaiseNameChanged(string oldName, string newName)
37         {
38             var h = this.NameChanged;
39             if (h != null) h(this, new ValueChangedEventArgs<string>(oldName, newName));
40         }
41     }
42 }
```

31 行目のように、EventHandler<EventArgs> の部分を EventHandler<ValueChangedEventArgs<string>> に変更しています。これに伴い、39 行目でイベントを発行していた部分では、EventArgs.Empty を渡していたところを、定義したイベント引数を渡すように変更しています。この変更のため、RaiseNameChanged() メソッドには 2 つの引数を追加し、これを呼び出すところも適宜変更しています。

このような変更をした Model を使用する ViewModel を次のように変更します。

コード 3.13：イベントを購読する MainViewModel

MainViewModel.cs

```
1  namespace Tips_CustomEvent.ViewModels
2  {
3      using System;
4      using Tips_CustomEvent.Models;
5 
6      public class MainViewModel : NotificationObject
7      {
8          private string _name;
9          /// <summary>
10         /// 名前を取得または設定します。
11         /// </summary>
12         public string Name
13         {
14             get { return this._name; }
15             set
16             {
17                 if (SetProperty(ref this._name, value))
18                 {
19                     this._settings.Name = this._name;
20                 }
21             }
22         }
23 
24         private bool _isSubscribed;
25         /// <summary>
```

```
26          /// イベント購読するかどうかを取得または設定します。
27          /// </summary>
28          public bool IsSubscribed
29          {
30              get { return this._isSubscribed; }
31              set
32              {
33                  if (SetProperty(ref this._isSubscribed, value))
34                  {
35                      if (this._isSubscribed)
36                      {
37                          this._settings.NameChanged += Settings_NameChanged;
38                      }
39                      else
40                      {
41                          this._settings.NameChanged -= Settings_NameChanged;
42                      }
43                  }
44              }
45          }
46
47          private int _counter;
48          /// <summary>
49          /// カウンタ値を取得します。
50          /// </summary>
51          public int Counter
52          {
53              get { return this._counter; }
54              private set { SetProperty(ref this._counter, value); }
55          }
56
57          /// <summary>
58          /// 設定を保持します。
59          /// </summary>
60          private Settings _settings = new Settings();
61
62          /// <summary>
63          /// 設定の Name プロパティ変更イベントハンドラ
64          /// </summary>
65          /// <param name="sender">イベント発行元</param>
66          /// <param name="e">イベント引数</param>
67          private void Settings_NameChanged(object sender, ValueChangedEventArgs<string> e)
68          {
69              this.Counter = e.NewValue.Length;
70          }
71      }
72  }
```

変更した部分は 67 行目と 69 行目だけです。イベントハンドラのデリゲートが変更されたので、入力引数の型を変更しています。また、イベントが発生した回数を数えていた Counter プロパティには、新しい Name プロパティの値の長さをカウントしてもらうようにしています。

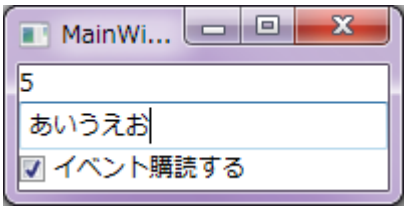


図 3.6：文字列の長さがカウントされる

3.6 ソースにリンクしたデバッグ出力をおこなう (Tips_DebugTrace)

System.Diagnostics.Debug クラスや System.Diagnostics.Trace クラスを使用することで出力ウィンドウに文字列を出力することができます。このとき、次のような書式で出力をおこなうことで、出力されたテキストをダブルクリックすると該当するソースコードにジャンプすることができるようになります。

コード 3.14 : ソースにリンクしたデバッグ出力をするサンプルコード

Program.cs

```
 1 namespace Tips_DebugTrace
 2 {
 3     using System.Diagnostics;
 4     using System.Runtime.CompilerServices;
 5 
 6     class Program
 7     {
 8         static void Main(string[] args)
 9         {
10             WriteLine("アプリケーションが起動しました。");
11 
12             Task();
13 
14             System.Console.ReadKey();
15         }
16 
17         static void Task()
18         {
19             WriteLine("Task メソッドが実行されました。");
20         }
21 
22         /// <summary>
23         /// デバッグ出力をおこないます。
24         /// </summary>
25         /// <param name="message">出力するメッセージを指定します。</param>
26         /// <param name="name">メソッド名を指定します。</param>
27         /// <param name="filePath">ソースファイルのフルパスを指定します。</param>
28         /// <param name="lineNumber">行番号を指定します。</param>
29         [Conditional("DEBUG")]
30         static void WriteLine(string message, [CallerMemberName]string name = null,
    [CallerFilePath]string filePath = null, [CallerLineNumber]int lineNumber = 0)
31         {
32             var str = string.Format("{0}({1}) :{2} {3}", filePath, lineNumber, name != null ?
    " [" + name + "]" : "", message);
33             System.Diagnostics.Debug.WriteLine(str);
34         }
35     }
36 }
```

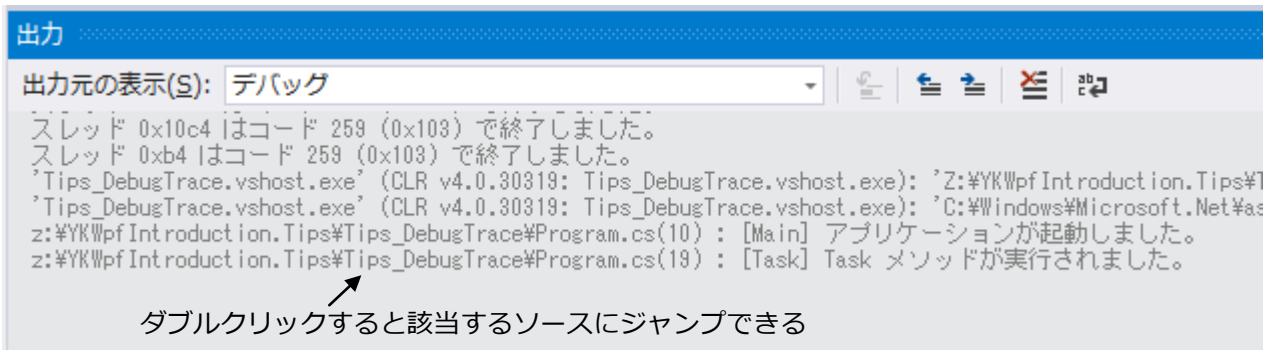


図 3.7：Person プロパティが候補として表示されている

重要なのは始めにファイルのフルパス、続いて "("、")" の括弧で括った行番号となっていることです。このようなメッセージの場合、ダブルクリックすると該当するソースファイルが開き、該当する行番号にカーソルがジャンプするようになります。上記の例では呼び出し元のメソッド名も同時に表示することでさらに分かりやすくしています。

また、VSColorOutput などの拡張機能によって出力ウィンドウに表示されるテキストに色を付けている場合、Information や Error といったキーワードを含むことでさらにわかりやすいメッセージとなります。

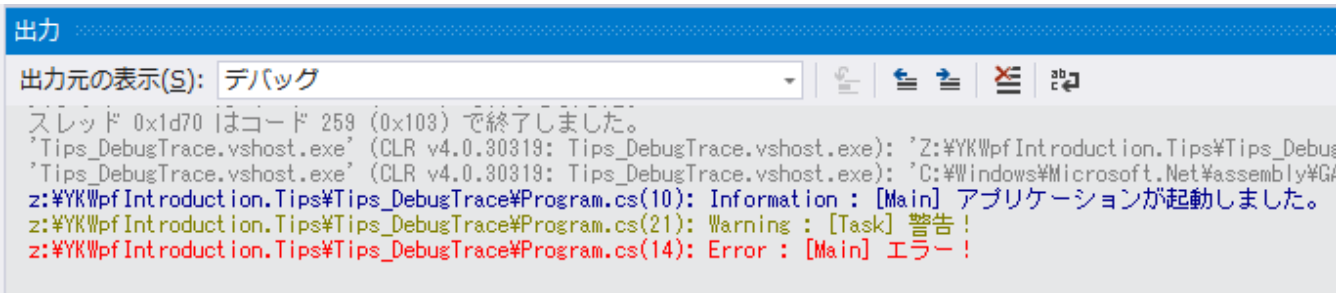


図 3.8：拡張機能 VSColorOutput を使用するとキーワードによって色付けされる

3.7 元に戻す/やり直し機能を実装するには（Tips_Undo）

アプリケーション上の操作をおこなっているとき、操作を取り消して元に戻したい場合や、元に戻したけど、やっぱりやり直したい場合があります。この機能を C# でどのように実装するかを紹介します。

3.7.1 データではなく操作を保持する

「元に戻す」ということは、操作されたデータを元の状態に戻すということです。例えば string 型の Name プロパティを考えると、

1. Name = "田中" とする
2. Name = "佐藤" とする
3. 元に戻す機能を使う → Name == "田中" になる
4. やり直し機能を使う → Name == "佐藤" になる

というイメージです。これだけの機能を実現する場合、プロパティ値の新旧の値、つまりデータを保持しておけば良さそうです。

一方、人物情報を表す Person クラスを用いた、List<Person> クラスの People プロパティではどのようになるでしょうか。

1. 田中さんを表す Person クラスのインスタンスを People プロパティに追加する
2. 佐藤さんを表す Person クラスのインスタンスを People プロパティに追加する
3. 元に戻す機能を使う → People プロパティから佐藤さんのインスタンスを除外する
4. やり直し機能を使う → People プロパティに佐藤さんのインスタンスを追加する

田中さんや佐藤さんを表す Person クラスのインスタンスを保持しておくことも必要ですが、元に戻す場合は List<T>.Remove() メソッド、やり直すとき場合は List<T>.Add() メソッドを使うというように、使用するメソッドも保持しておく必要があります。つまり、データを保持するだけでは不十分で、操作を保持しなくてはいけません。

3.7.2 Action デリゲートで操作を保持する

操作を保持するには、System.Action デリゲートが非常に有用です。例えば次のようなコードを実行してみましょう。

コード 3.15：Action デリゲートの使い方

Program.cs

```
 1  namespace Tips_Undo
 2  {
 3      using System;
 4
 5      class Program
 6      {
 7          static void Main(string[] args)
 8          {
 9              DoWork(() =>
10              {
11                  Console.WriteLine("Action デリゲートで処理します。");
12              });
13
14              Console.ReadKey();
15          }
16
17          private static void DoWork(Action action)
18          {
19              if (action != null)
20                  action();
```

```
        }
    }
}
```

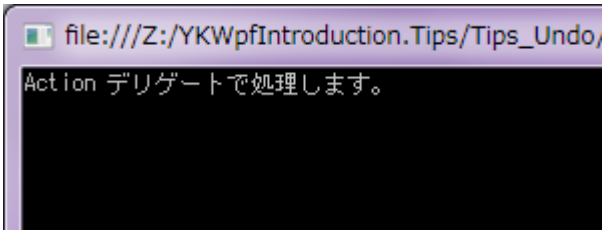


図 3.9：外部から指定された処理内容が実行される

DoWork() メソッドは、入力引数である Action デリゲートを実行するだけとなっています。つまり、DoWork() メソッドを呼び出す側が実際の処理内容を指定できるということです。
これを応用することで、例えば元に戻す機能を想定して次のようなコードが書けると思います。

コード 3.16：Action デリゲートをスタックする

Program.cs

```
namespace Tips_Undo
{
    using System;
    using System.Collections.Generic;

    class Program
    {
        static void Main(string[] args)
        {
            actions = new Stack<Action>();
            actions.Push(() => Console.WriteLine("始めの処理"));
            actions.Push(() => Console.WriteLine("2 番目の処理"));
            actions.Push(() => Console.WriteLine("3 番目の処理"));
            actions.Push(() => Console.WriteLine("最後の処理"));

            Undo();
            Undo();
            Undo();
            Undo();
            Undo();

            Console.ReadKey();
        }

        private static Stack<Action> actions;

        private static void Undo()
        {
            if (actions.Count > 0)
            {
                var action = actions.Pop();
                if (action != null)
                    action();
            }
            else
            {
                Console.WriteLine("もうないよ！");
            }
```

```
39          }
40      }
41  }
```

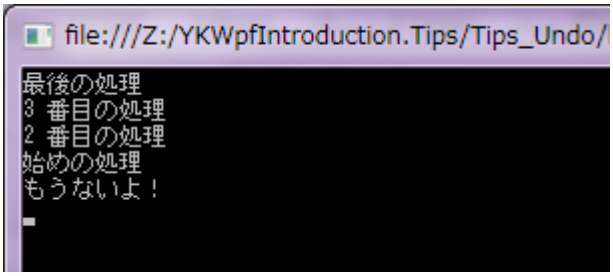


図 3.10：スタックされた処理が逆順で実行される

System.Collections.Generic.Stack<T> クラスはその名の通り型 T のインスタンスをスタックするためのクラスです。ここでは Action デリゲートをスタックするようにしています。
スタックを積み上げるには Push() メソッドを、スタックから取り出すには Pop() メソッドを使用します。Pop() メソッドを使用するとスタックから取り出されたインスタンスは除外されます。

やり直し機能は元に戻す機能の逆操作をおこなうわけですから、同じく System.Collection.Generic.Stack<T> クラスで、元に戻す機能とは逆順に Action デリゲートを積み上げていくことになります。

3.7.3 元に戻す/やり直し機能を実装する

それでは本格的に機能を実装していきましょう。まずは根幹となる History クラスを次のように定義します。元に戻す操作とやり直す操作は表裏一体なので、ひとつのクラスにまとめておくとわかりやすくなります。

コード 3.17：History クラスの定義

History.cs

```
 1  namespace Tips_Undo
 2  {
 3      using System;
 4
 5      /// <summary>
 6      /// 特定のアクションを保持し、任意のタイミングで実行するためのクラスです。
 7      /// </summary>
 8      public class History
 9      {
10          /// <summary>
11          /// 新しいインスタンスを生成します。
12          /// </summary>
13          /// <param name="undoAction">アンドゥアクションを指定します。</param>
14          /// <param name="undoAction">リドゥアクションを指定します。</param>
15          /// <param name="name">操作名を指定します。</param>
16          public History(Action undoAction, Action redoAction, string name)
17          {
18              if (undoAction == null)
19                  throw new ArgumentNullException("必ずアンドゥアクションを指定してください。
    ");
20              if (redoAction == null)
21                  throw new ArgumentNullException("必ずリドゥアクションを指定してください。");
22              this._undoAction = undoAction;
23              this._redoAction = redoAction;
24              this.Name = name;
25          }
```

```
26
27          /// <summary>
28          /// アンドゥアクションを保持します。
29          /// </summary>
30          private Action _undoAction;
31
32          /// <summary>
33          /// リドゥアクションを保持します。
34          /// </summary>
35          private Action _redoAction;
36
37          /// <summary>
38          /// 保持しているアンドゥアクションを実行します。
39          /// </summary>
40          public void UnDo()
41          {
42              this._undoAction();
43          }
44
45          /// <summary>
46          /// 保持しているリドゥアクションを実行します。
47          /// </summary>
48          public void ReDo()
49          {
50              this._redoAction();
51          }
52
53          /// <summary>
54          /// 操作名を取得します。
55          /// </summary>
56          public string Name { get; private set; }
57
58          /// <summary>
59          /// 自身を文字列として表現します。
60          /// </summary>
61          /// <returns>自身を表現する文字列を返します。</returns>
62          public override string ToString()
63          {
64              return this.Name;
65          }
66      }
67  }
```

このクラスを利用して、例えば ViewModel で機能を実現する場合は次のようなコードになります。

何か操作をするときは、コードの最後にある DoAction() メソッドを使用します。このメソッドの入力引数には、元に戻すときの操作と、やり直すときの操作を Action デリゲートとして与えます。また、操作名を指定することで、操作履歴のリストに表示することができます。

コード 3.18：ViewModel で元に戻す/やり直し機能を実装する一例

HistoryViewModel.cs

```
1   namespace Tips_Undo.ViewModels
2   {
3       using System;
4       using System.Collections.Generic;
5       using System.Linq;
6       using ObjectEditorSample.Models;
7       using YKToolkit.Bindings;
```

```
 8
 9      public class HistoryViewModel : NotificationObject
10      {
11          #region シングルトンクラス
12          /// <summary>
13          /// インスタンスを保持します。
14          /// </summary>
15          private static readonly HistoryViewModel _instance;
16
17          /// <summary>
18          /// インスタンスを取得します。
19          /// </summary>
20          public static HistoryViewModel Instance { get { return _instance; } }
21
22          /// <summary>
23          /// 静的なコンストラクタです。
24          /// </summary>
25          static HistoryViewModel()
26          {
27              _instance = new HistoryViewModel();
28          }
29
30          /// <summary>
31          /// private なコンストラクタを定義することで外部からインスタンスを生成されることを抑制
    します。
32          /// </summary>
33          private HistoryViewModel()
34          {
35          }
36          #endregion シングルトンクラス
37
38          private int _stackCapacity = 2;
39          /// <summary>
40          /// 操作履歴のバッファサイズを取得または設定します。
41          /// </summary>
42          public int StackCapacity
43          {
44              get { return this._stackCapacity; }
45              set
46              {
47                  if (SetProperty(ref this._stackCapacity, value))
48                  {
49                      this.UndoStack = new Stack<History>(this._stackCapacity);
50                      this.RedoStack = new Stack<History>(this._stackCapacity);
51                  }
52              }
53          }
54
55          public void ClearHistory()
56          {
57              this.UndoStack.Clear();
58              this.RedoStack.Clear();
59              RaisePropertyChanged("UndoElements");
60              RaisePropertyChanged("RedoElements");
61          }
62
63          private Stack<History> _undoStack;
64          /// <summary>
```

```
65          /// アンドゥする操作を溜めておくスタックを取得します。
66          /// </summary>
67          private Stack<History> UndoStack
68          {
69              get { return _undoStack ?? (_undoStack = new
    Stack<History>(this.StackCapacity)); }
70              set { SetProperty(ref this._undoStack, value); }
71          }
72
73          private Stack<History> _redoStack;
74          /// <summary>
75          /// リドゥする操作を溜めておくスタックを取得します。
76          /// </summary>
77          private Stack<History> RedoStack
78          {
79              get { return _redoStack ?? (_redoStack = new
    Stack<History>(this.StackCapacity)); }
80              set { SetProperty(ref this._redoStack, value); }
81          }
82
83          /// <summary>
84          /// アンドゥ操作リストを取得します。
85          /// </summary>
86          public string[] UndoElements { get { return this.UndoStack.Select(x =>
    x.Name).ToArray(); } }
87
88          private bool _isChagnedFromUI = true;
89
90          private int _selectedUndoElementIndex = -1;
91          public int SelectedUndoElementIndex
92          {
93              get { return this._selectedUndoElementIndex; }
94              set
95              {
96                  if (SetProperty(ref this._selectedUndoElementIndex, value))
97                  {
98                      if (this._isChagnedFromUI)
99                      {
100                         var undoCount = 1 + this._selectedUndoElementIndex;
101
102                         for (var i = 0; i < undoCount; i++)
103                         {
104                             Undo();
105                         }
106
107                         this._isChagnedFromUI = false;
108                         RaisePropertyChanged("UndoElements");
109                         RaisePropertyChanged("RedoElements");
110                         this._isChagnedFromUI = true;
111
112
    System.Windows.Threading.Dispatcher.CurrentDispatcher.BeginInvoke(new Action(() =>
113                         {
114                             this.SelectedUndoElementIndex = -1;
115                         }), System.Windows.Threading.DispatcherPriority.ApplicationIdle);
116                     }
117                 }
118             }
```

```
119         }
120 
121         /// <summary>
122         /// リドゥ操作リストを取得します。
123         /// </summary>
124         public string[] RedoElements { get { return this.RedoStack.Select(x =>
    x.Name).ToArray(); } }
125 
126         private int _selectedRedoElementIndex = -1;
127         public int SelectedRedoElementIndex
128         {
129             get { return this._selectedRedoElementIndex; }
130             set
131             {
132                 if (SetProperty(ref this._selectedRedoElementIndex, value))
133                 {
134                     if (this._isChagnedFromUI)
135                     {
136                         var redoCount = 1 + this._selectedRedoElementIndex;
137 
138                         for (var i = 0; i < redoCount; i++)
139                         {
140                             Redo();
141                         }
142 
143                         this._isChagnedFromUI = false;
144                         RaisePropertyChanged("UndoElements");
145                         RaisePropertyChanged("RedoElements");
146                         this._isChagnedFromUI = true;
147 
148                         this.SelectedRedoElementIndex = -1;
149                     }
150                 }
151             }
152         }
153 
154         private void Undo()
155         {
156             var action = this.UndoStack.Pop();
157             action.UnDo();
158             this.RedoStack.Push(action);
159         }
160 
161         private void Redo()
162         {
163             var action = this.RedoStack.Pop();
164             action.ReDo();
165             this.UndoStack.Push(action);
166         }
167 
168         private DelegateCommand _undoCommand;
169         /// <summary>
170         /// 元に戻すコマンドを取得します。
171         /// </summary>
172         public DelegateCommand UndoCommand
173         {
174             get
175             {
```

```
176                 return _undoCommand ?? (_undoCommand = new DelegateCommand(
177                 _ =>
178                 {
179                     Undo();
180 
181                     this._isChagnedFromUI = false;
182                     RaisePropertyChanged("UndoElements");
183                     RaisePropertyChanged("RedoElements");
184                     this._isChagnedFromUI = true;
185                 },
186                 _ => this.UndoStack.Count > 0));
187             }
188         }
189 
190         private DelegateCommand _redoCommand;
191         /// <summary>
192         /// やり直しコマンドを取得します。
193         /// </summary>
194         public DelegateCommand RedoCommand
195         {
196             get
197             {
198                 return _redoCommand ?? (_redoCommand = new DelegateCommand(
199                 _ =>
200                 {
201                     Redo();
202 
203                     this._isChagnedFromUI = false;
204                     RaisePropertyChanged("UndoElements");
205                     RaisePropertyChanged("RedoElements");
206                     this._isChagnedFromUI = true;
207                 },
208                 _ => this.RedoStack.Count > 0));
209             }
210         }
211 
212         /// <summary>
213         /// 指定された操作をおこない、操作履歴に手順を登録します。
214         /// </summary>
215         /// <param name="undoAction">元に戻す操作を指定します。</param>
216         /// <param name="redoAction">やり直す操作を指定します。</param>
217         /// <param name="name">操作名を指定します。</param>
218         public void DoAction(Action undoAction, Action redoAction, string name)
219         {
220             redoAction();
221 
222             var history = new History(undoAction, redoAction, name);
223             this.UndoStack.Push(history);
224             this.RedoStack.Clear();
225 
226             this._isChagnedFromUI = false;
227             RaisePropertyChanged("UndoElements");
228             RaisePropertyChanged("RedoElements");
229             this._isChagnedFromUI = true;
230         }
231     }
232 }
```

3.8 #if～#endif の代わりに Conditional 属性を使う (Tips_ConditionalAttribute)

3.8.1 #if～#endif が向かない例

#if～#endif で囲むと、#if で指定したプリプロセスのシンボルが存在するかどうかで、囲まれた部分のプログラムをコンパイルするかどうかをスイッチングすることができます。例えば次のようなコードを見てみましょう。

コード 3.19：#if～#endif でコードを囲む

Program.cs

```
 1  namespace Tips_ConditionalAttribute
 2  {
 3      using System;
 4
 5      class Program
 6      {
 7          static void Main(string[] args)
 8          {
 9              var model = new Sample();
10
11  #if DEBUG
12              model.DebugOutput();
13  #endif
14
15              Console.ReadKey();
16          }
17      }
18
19      public class Sample
20      {
21  #if DEBUG
22          public void DebugOutput()
23          {
24              Console.WriteLine("シンボル ¥"DEBUG¥" が定義されているときのみ実行されます。");
25          }
26  #endif
27      }
28  }
```

#if で DEBUG というシンボルを指定しているため、このシンボルが定義されているときだけ DebugOutput() メソッドがコンパイルされることになります。つまり、シンボル DEBUG が定義されていないときは DebugOutput() メソッドが存在しないことになるため、このメソッドを呼び出す部分にも同様に #if～#endif を書かないと、シンボル DEBUG が定義されていない構成でコンパイルしたときにコンパイルエラーとなってしまいます。

このようなケースでは、#if～#endif を両方に書かなくてはならなくなり、コードが煩雑になります。こういった場合は #if～#endif ではなく、Conditional 属性を使用したほうがコードがわかりやすく、ミスが発生する可能性も低くなります。

3.8.2 Conditional 属性の使い方

前節の例を Conditional 属性で書き直したコードは次のようになります。

コード 3.20：Conditional 属性の使用例

Program.cs

```
1  namespace Tips_ConditionalAttribute
2  {
```

```
 3      using System;
 4      using System.Diagnostics;
 5
 6      class Program
 7      {
 8          static void Main(string[] args)
 9          {
10              var model = new Sample();
11
12              model.DebugOutput();
13
14              Console.ReadKey();
15          }
16      }
17
18      public class Sample
19      {
20          [Conditional("DEBUG")]
21          public void DebugOutput()
22          {
23              Console.WriteLine("シンボル ¥"DEBUG¥" が定義されているときのみ実行されます。");
24          }
25      }
26  }
```

Conditional 属性はメソッドの宣言前に添付する形で記述します。引数にプリプロセスシンボルを指定することで、#if と同じ役割を果たします。しかし、このメソッドを呼び出す方には特に追加する記述はありません。このように、メソッドの属性として Conditional 属性を付加することで、コンパイラが自動的に認識し、コンパイル時に呼び出しそのものを省略してくれるようになります。#if～#endif を使用する方法と違い、メソッドの定義側にだけ書けばいいので、使う方も特に気にせずに使うことができます。

ただし、呼び出し自体を省略するようになる性質からか、Conditional 属性は戻り値が void であるメソッドにしか適用できません。void 以外の戻り値を持つメソッドに Conditional 属性を追加した場合、コンパイルエラーが発生します。

3.8.3 複数シンボルの条件

Conditional 属性はひとつのメソッドに対していくつも指定することができます。例えば次のようにした場合、A または B または C のシンボルがひとつでも定義されていた場合、DebugOutput() メソッドが有効となります。

コード 3.21：複数の Conditional 属性は OR 条件となる

Program.cs

```
1      public class Sample
2      {
3          [Conditional("A"), Conditional("B"), Conditional("C")]
4          public void DebugOutput()
5          {
6              Console.WriteLine("A or B or C が定義されているときのみ実行されます。");
7          }
8      }
```

A と B が両方定義されていない限りコンパイルして欲しくないといった場合は、メソッドをわけて定義する必要があります。

コード 3.22：Conditional 属性を使って AND 条件にする例

Program.cs

```
1      public class Sample
```

```
2     {
3         [Conditional("A")]
4         public void DoIfA()
5         {
6             DoIfAandB();
7         }
8 
9         [Conditional("B")]
10        private void DoIfAandB()
11        {
12            Console.WriteLine("A and B が定義されているときのみ実行されます。");
13        }
14    }
```

3.9 シリアル通信をおこなうには（Tips_Serial）

外部機器と通信する方法として昔から RS-232C 通信が良く知られています。ここでは C# による RS-232C 通信の実装方法を紹介します。

3.9.1 ポートの解放と閉鎖

C# で RS-232C 通信をおこなうには、System.IO.Ports.SerialPort クラスを使用します。インスタンス生成時または生成後にポート名やボーレートなどを指定します。その後、Open() メソッドでポートオープン、Close() メソッドでポートクローズをおこないます。ただし、SerialPort クラスは IDisposable インターフェースを実装しているクラスでもあるため、下記コードのように using を使用することで、Close() メソッドを省略することもできます。どちらにしろ、ポートオープン中は他のアプリケーションからアクセスできなくなるため、忘れずにポートを閉じるようにしましょう。

コード 3.23：SerialPort クラスでポートをオープンする

Program.cs

```
 1  namespace Tips_Serial
 2  {
 3      using System;
 4      using System.IO.Ports;
 5      using System.Text;
 6
 7      class Program
 8      {
 9          static void Main(string[] args)
10          {
11              using (var serial = new SerialPort()
12                  {
13                      PortName = "COM11",
14                      BaudRate = 9600,
15                      DataBits = 8,
16                      Parity = Parity.None,
17                      StopBits = StopBits.One,
18                      DtrEnable = false,
19                      RtsEnable = false,
20                      ReadBufferSize = 256,
21                      WriteBufferSize = 256,
22                      Encoding = Encoding.GetEncoding("Shift_JIS"),
23                  })
24              {
25                  try
26                  {
27                      // ポートオープン
28                      serial.Open();
29                  }
30                  catch (Exception ex)
31                  {
32                      Console.WriteLine(ex);
33                  }
34                  finally
35                  {
36                      Console.WriteLine("{0} ポートを" + (serial.IsOpen ? "開きました。" : "
    開けませんでした。"), serial.PortName);
37                  }
38
39                  Console.ReadKey();
40              }
41          }
```

```
42      }
43  }
```

ポートが既に他のアプリケーションによって使用されているとき、UnAuthorizedAccessException 例外が発生するため、try-catch 構文で適切に対処しておいたほうが良いでしょう。

3.9.2 データを受信する

System.IO.Ports.SerialPort クラスは、受信動作自体が .NET Framework の仕組みで動作するため、インスタンス生成後、特にコードを追加する必要がありません。インスタンス生成後、何かデータを受信すると、DataReceived イベントが発生するので、このイベントにイベントハンドラを登録するだけで受信処理をおこなえます。

コード 3.24：SerialPort クラスでデータを受信する

Program.cs

```
 1  namespace Tips_Serial
 2  {
 3      using System;
 4      using System.IO.Ports;
 5      using System.Text;
 6
 7      class Program
 8      {
 9          static void Main(string[] args)
10          {
11              using (var serial = new SerialPort()
12                  {
13                      PortName = "COM11",
14                      BaudRate = 9600,
15                      DataBits = 8,
16                      Parity = Parity.None,
17                      StopBits = StopBits.One,
18                      DtrEnable = false,
19                      RtsEnable = false,
20                      ReadBufferSize = 256,
21                      WriteBufferSize = 256,
22                      Encoding = Encoding.GetEncoding("Shift_JIS"),
23                  })
24              {
25                  try
26                  {
27                      // 受信イベントを購読する
28                      serial.DataReceived += OnReceived;
29
30                      // ポートオープン
31                      serial.Open();
32                  }
33                  catch (Exception ex)
34                  {
35                      Console.WriteLine(ex);
36                  }
37                  finally
38                  {
39                      Console.WriteLine("{0} ポートを" + (serial.IsOpen ? "開きました。" : "
    開けませんでした。"), serial.PortName);
40                  }
41
42                  Console.ReadKey();
```

```
43              }
44          }
45
46          /// <summary>
47          /// データ受信イベントハンドラ
48          /// </summary>
49          /// <param name="sender">イベント発行元</param>
50          /// <param name="e">イベント引数</param>
51          private static void OnReceived(object sender, SerialDataReceivedEventArgs e)
52          {
53              var serial = sender as SerialPort;
54              var data = serial.ReadExisting();
55              Console.WriteLine(data);
56          }
57      }
58  }
```

受信動作自体が非同期的に裏で動くことになるため、上記のプログラムでは、ポートオープン後、42 行目のキー入力待ちで停止することになります。

受信イベントに OnReceived() メソッドを登録しておいたので、受信データを出力できるようになりました。エンコードとして Shift_JIS を指定して SerialPort クラスのインスタンスを生成しているので、日本語を受信すると自動的に変換されている様子がわかります。

ただし、受信イベントは非同期処理されているため、元のスレッドとは異なるスレッドで実行されています。したがって、WPF と連携する場合、UI の操作と絡むようなことをする場合は UI スレッドに切り替える必要があることを注意しなければいけません。スレッドの固有 ID は System.Threading.Thread.CurrentThread.ManagedThreadId プロパティで取得できるので、確認してみてください。

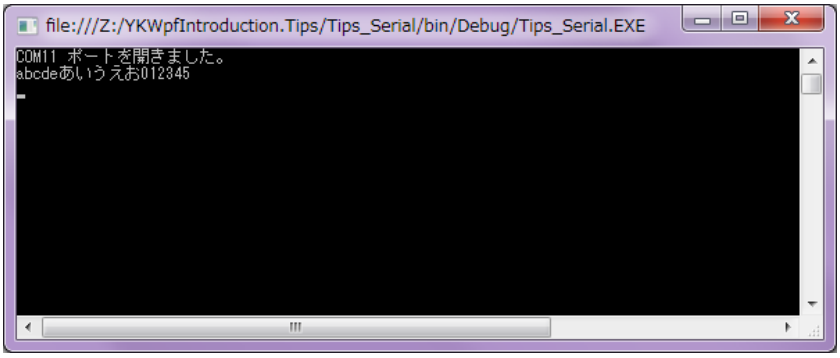


図 3.11：受信した文字列が表示される

3.9.3 データを送信する

System.IO.Ports.SerialPort クラスでデータを送信するときは Write() メソッドなどを使用します。

コード 3.25：SerialPort クラスでデータを送信する

Program.cs

```
1   namespace Tips_Serial
2   {
3       using System;
4       using System.IO.Ports;
5       using System.Text;
6
7       class Program
8       {
9           static void Main(string[] args)
```

```
10          {
11              using (var serial = new SerialPort()
12                  {
13                      PortName = "COM11",
14                      BaudRate = 9600,
15                      DataBits = 8,
16                      Parity = Parity.None,
17                      StopBits = StopBits.One,
18                      DtrEnable = false,
19                      RtsEnable = false,
20                      ReadBufferSize = 256,
21                      WriteBufferSize = 256,
22                      Encoding = Encoding.GetEncoding("Shift_JIS"),
23                  })
24              {
25                  try
26                  {
27                      // 受信イベントを購読する
28                      serial.DataReceived += OnReceived;
29
30                      // ポートオープン
31                      serial.Open();
32                  }
33                  catch (Exception ex)
34                  {
35                      Console.WriteLine(ex);
36                  }
37                  finally
38                  {
39                      Console.WriteLine("{0} ポートを" + (serial.IsOpen ? "開きました。" : "
    開けませんでした。"), serial.PortName);
40                  }
41
42                  Console.ReadKey();
43              }
44          }
45
46          /// <summary>
47          /// データ受信イベントハンドラ
48          /// </summary>
49          /// <param name="sender">イベント発行元</param>
50          /// <param name="e">イベント引数</param>
51          private static void OnReceived(object sender, SerialDataReceivedEventArgs e)
52          {
53              var serial = sender as SerialPort;
54              var data = serial.ReadExisting();
55              Console.WriteLine(data);
56
57              // データ送信
58              serial.Write(data.ToUpper());
59          }
60      }
61  }
```

上記の例では、受信したデータを元に、英字をすべて大文字にしたデータを Write() メソッドで送信しています。

4 WPF によるあれこれ

本章では、WPF で実現するための色々な技術を紹介します。

4.1 XAML デザイナーで ViewModel のメンバを Intellisense 候補とする

Visual Studio の Intellisense 機能によって、コーディングするときに候補が表示されたり、入力を省略できたりするようになり、非常に便利になってきています。しかし、XAML コードを編集する場合、データバインディングなどで ViewModel のクラスのメンバを入力することがありますが、この場合、Intellisense 機能による候補に独自のクラスが含まれないため、Intellisense の恩恵を受けることができません。

そこで、XAML コードに次のようなコードを追加することで、Intellisense 機能に ViewModel のメンバが含まれるようになります。

コード 4.1：Person プロパティを持つ ViewModel

MainViewModel.cs

```
1  namespace Tips_XamlDesigner.ViewModels
2  {
3      using Tips_XamlDesigner.Models;
4
5      public class MainViewModel : NotificationObject
6      {
7          private Person _person = new Person() { Name = "田中太郎" };
8          /// <summary>
9          /// 人物データを取得または設定します。
10         /// </summary>
11         public Person Person
12         {
13             get { return this._person; }
14             set { SetProperty(ref this._person, value); }
15         }
16     }
17 }
```

コード 4.2：DesignInstance に対する ViewModel の指定

MainView.xaml

```
1  <Window x:Class="Tips_XamlDesigner.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6          xmlns:vm="clr-namespace:Tips_XamlDesigner.ViewModels;assembly=Tips_XamlDesigner"
7          d:DataContext="{d:DesignInstance {x:Type vm:MainViewModel}}"
8          mc:Ignorable="d"
9          Title="MainView" Height="300" Width="300">
10     <StackPanel>
11         <TextBlock Text="{Binding Person.Name}" />
12         <TextBlock Text="test" />
13     </StackPanel>
14 </Window>
```

デザインモード時に関する設定を名前空間のエイリアス d で設定できるようにしています。エイリアス d を使用してデザインモード時の DataContext に MainViewModel を指定しています。また、エイリアス d による設定は実際に実行する場合には関係のない設定のため、名前空間のエイリアス mc によって、エイリアス d による設定は無視するように mc:Ignorable に d を設定しています。

こうすることで、XAML コードを編集するとき、MainViewModel クラスのメンバも Intellisense 機能の候補に含まれるようになります。例えば、11 行目で Person プロパティを入力するとき、"Binding " まで入力するといくつか入力候補が表示されますが、その中に "Person" が含まれるようになります。

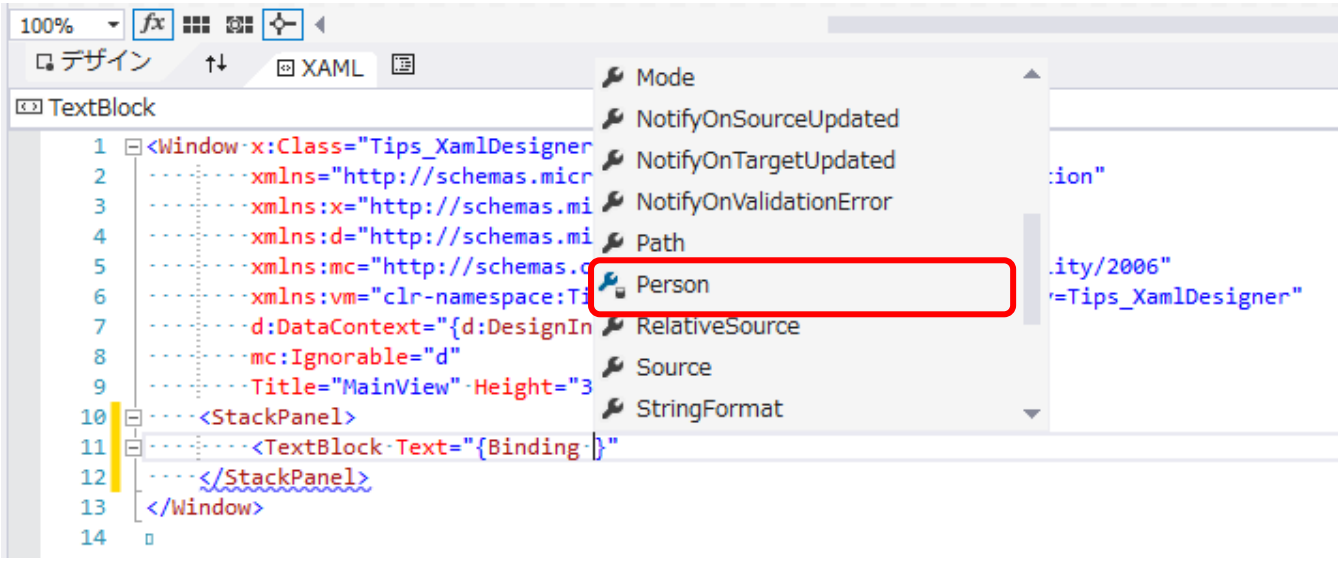
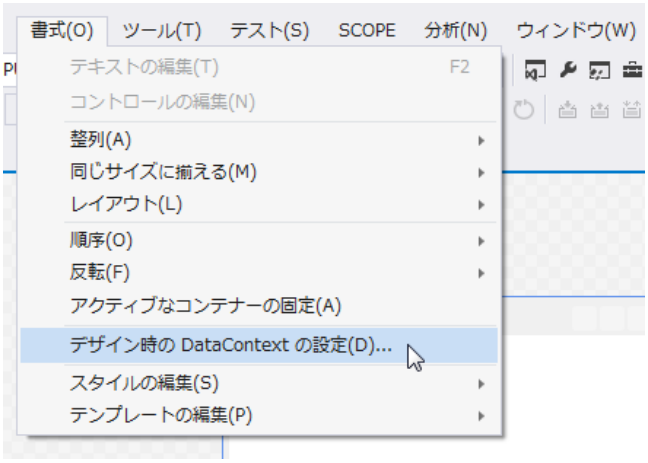


図 4.1：Person プロパティが候補として表示されている

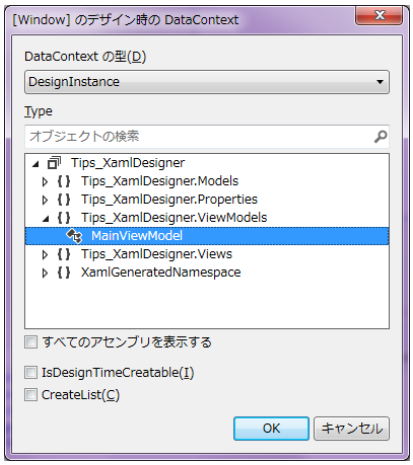
ところで、ここでは XAML コードに mc や d などのエイリアスを手入力しましたが、Visual Studio の GUI 上から自動的に挿入することもできます。

XAML デザイナー上でルート要素のウィンドウを選択した状態で、「書式」メニューの「デザイン時の DataContext の設定(D)...」という項目を選択します。すると、図 4.2 (b) のようなダイアログが表示されるため、DataContext の型として DesignInstance を選択し、設定したい ViewModel のクラスをツリーから選択して「OK」ボタンを押します。

すると、さきほどのサンプルコードのようなエイリアス mc や d などが自動的に追加されます。ただし、スペースや改行などが乱れるため、綺麗にしたい場合は手動で整形する必要があります。



(a) Window を選択してからメニューを表示する



(b) デザイン時の DataContext を選択する

図 4.2：デザイン時の DataContext 設定方法

4.2 デザイン実行かどうかを判定する (Tips_XamlDesigner)

デザイン実行とは、XAML デザイナーがバックグラウンドでコードを解析・実行することです。このおかげで XAML デザイナー上の表示がリアルタイムに変化しています。

ユーザーコントロールやカスタムコントロールを配置するとき、そのコンテンツに動的なデータがある場合、デザイン実行ではまだアプリケーションは動いていないため、デザイン実行中に予期せぬエラーや例外が発生してしまい、XAML デザイナー上で画面の確認ができなくなってしまいます。このようなことが起きないようにするために、デザイン実行かどうかをコードで判断することができます。

例として次のようなユーザーコントロールを作成します。

コード 4.3 : TextBlock コントロールを持つ UserControl の例

Signals.xaml

```
<UserControl x:Class="Tips_XamlDesigner.Views.Signals"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <StackPanel Orientation="Horizontal">
        <Ellipse Width="20" Height="20" Margin="5" Fill="Green" />
        <Ellipse Width="20" Height="20" Margin="5" Fill="Yellow" />
        <Ellipse Width="20" Height="20" Margin="5" Fill="Red" />
        <TextBlock x:Name="textblock1" Text="Signals" VerticalAlignment="Center" />
    </StackPanel>
</UserControl>
```

コード 4.4 : コードビハインド

Signals.xaml.cs

```
namespace Tips_XamlDesigner.Views
{
    using System.Windows.Controls;

    /// <summary>
    /// Signals.xaml の相互作用ロジック
    /// </summary>
    public partial class Signals : UserControl
    {
        public Signals()
        {
            InitializeComponent();

            if (System.ComponentModel.DesignerProperties.GetIsInDesignMode(this))
            {
                this.textblock1.Text = "デザインモード";
            }
        }
    }
}
```

XAML 上で "Signals" というテキストを持つ TextBlock コントロールに対し、コードビハインドでは、ある条件によって "デザインモード" というテキストに変更するコードが組み込まれています。

System.ComponentModel.DesignerProperties クラスは、デザイナーとの通信に使用される添付プロパティを提供しています。そして、IsDesignMode という添付プロパティで、対象とするオブジェクトのインスタンスがデザイナーのコンテキストで実行されているかどうかを確認することができます。

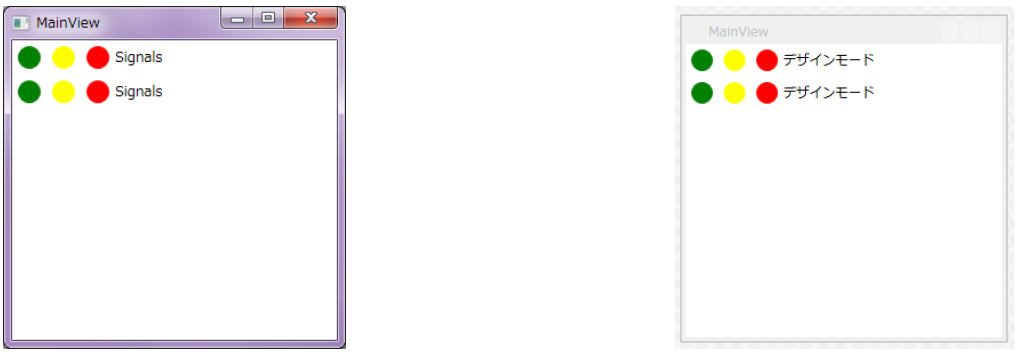
IsDesignMode は添付プロパティなので、プロパティ名の頭に "Get" を付けた GetIsDesignMode() メソッドでプロパティ値を取得することができます。したがって、14 行目のように、引数にユーザーコントロール自身を示す this を指定して IsDesignMode 添付プロパティの値を取得することで、デザインモードかどうかを判定することができます。

このユーザーコントロールをウィンドウに配置してみましょう。XAML コードは次のようになります。

コード 4.5：TextBlock コントロールを持つ UserControl の例

MainView.xaml

```
1  <Window x:Class="Tips_XamlDesigner.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:vw="clr-namespace:Tips_XamlDesigner.Views"
5          Title="MainView" Height="300" Width="300">
6      <StackPanel>
7          <vw:Signals />
8          <vw:Signals />
9      </StackPanel>
10 </Window>
```



(a) 起動時のウィンドウ　　(b) XAML デザイナー上の表示

図 4.3：実行時とデザイナー上の表示が異なる

実行するとユーザーコントロールの XAML で定義したように、"Signals" というテキストが表示されていますが、XAML デザイナー上ではコードビハインドで変更された文字列になっていることがわかります。

このサンプルでは静的なデータを書き換えるのみでしたが、実行しないと取得できない動的なデータなどに対して、デザインモードの場合には仮の値を入れておくようにしておくなどしておけば、XAML デザイナーがエラーを出力することなく画面が確認できるようになります。

また、Window や UserControl など View に関するインスタンスが取得できない場合、GetIsDesignMode() メソッドの引数に DependencyObject を渡すことができません。この場合は次のような方法でデザイン実行かどうかを判定します。

コード 4.6：UI 以外のコードでデザイン実行かどうかを判定する

Sample.cs

```
1  if
   ((bool)(System.ComponentModel.DesignerProperties.IsInDesignModeProperty.GetMetadata(typ
   eof(System.Windows.DependencyObject)).DefaultValue))
2  {
3      // デザイン実行時は処理しない
4      return;
5  }
```

4.3 StringFormat によってフォーマットを指定する (Tips_StringFormat)

TextBlock コントロールなどでテキストを表示するときに、ViewModel から与えられたものを変換して表示したい場合には StringFormat が便利です。

コード 4.7：データの表示形式変更のための ViewModel

MainViewModel.cs

```
1  namespace Tips_StringFormat.ViewModels
2  {
3      using System;
4
5      public class MainViewModel
6      {
7          /// <summary>
8          /// 新しいインスタンスを生成します。
9          /// </summary>
10         public MainViewModel()
11         {
12             this.Value = 1234.56789;
13             this.Integer = 123456789;
14             this.Text = "123456789";
15             this.Name = "田中 太郎";
16             this.Date = DateTime.Now;
17         }
18
19         /// <summary>
20         /// 小数を持つ数値を取得します。
21         /// </summary>
22         public double Value { get; private set; }
23
24         /// <summary>
25         /// 整数値を取得します。
26         /// </summary>
27         public int Integer { get; private set; }
28
29         /// <summary>
30         /// 数値の文字列表現を取得します。
31         /// </summary>
32         public string Text { get; private set; }
33
34         /// <summary>
35         /// 文字列を取得します。
36         /// </summary>
37         public string Name { get; private set; }
38
39         /// <summary>
40         /// 日付を取得します。
41         /// </summary>
42         public DateTime Date { get; private set; }
43     }
44 }
```

コード 4.8：StringFormat によってデータの表示形式を変更する

MainView.xaml

```
1  <Window x:Class="Tips_StringFormat.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="420" Width="525">
```

```
    <StackPanel>
        <GroupBox Header="数値の表示">
            <StackPanel>
                <TextBlock Text="{Binding Value, StringFormat={}{0:N2}}"/>
                <TextBlock Text="{Binding Value, StringFormat={}{0:N3}}"/>
                <TextBlock Text="{Binding Value, StringFormat={}{0:00000}}"/>
            </StackPanel>
        </GroupBox>

        <GroupBox Header="価格の表示">
            <StackPanel>
                <TextBlock Text="{Binding Integer, StringFormat={}{0:C}}"/>
                <TextBlock Text="{Binding Integer, StringFormat={}{0:C},
ConverterCulture=ja-JP}"/>
                <TextBlock Text="{Binding Integer, StringFormat=定価{0:N0}円}"/>
                <TextBlock Text="文字列による数値表現に StringFormat は効果がない" />
                <TextBlock Text="{Binding Text, StringFormat={}{0:N2}円}"/>
            </StackPanel>
        </GroupBox>

        <GroupBox Header="16 進数の表示">
            <StackPanel>
                <TextBlock Text="{Binding Integer, StringFormat={}{0:X}}"/>
                <TextBlock Text="{Binding Integer, StringFormat={}0x{0:X8}}"/>
            </StackPanel>
        </GroupBox>

        <GroupBox Header="日付の表示">
            <StackPanel>
                <TextBlock Text="{Binding Date}"/>
                <TextBlock Text="{Binding Date, ConverterCulture=ja-JP}"/>
                <TextBlock Text="{Binding Date, StringFormat={}{0:yyyy年MM月dd日
HH:mm:ss}}"/>
            </StackPanel>
        </GroupBox>

        <GroupBox Header="マルチバインディング">
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="氏名:{0} 得点:{1:N0}点">
                        <Binding Path="Name"/>
                        <Binding Path="Integer"/>
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </GroupBox>
    </StackPanel>
</Window>
```

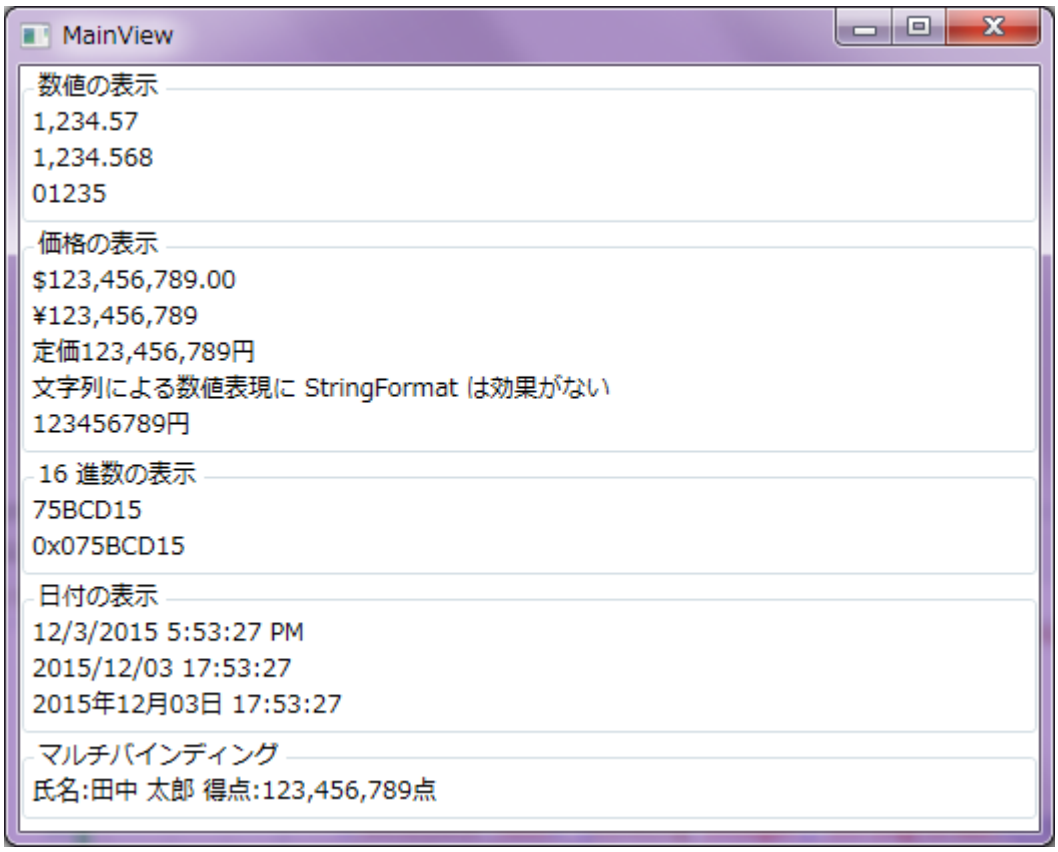


図 4.4：StringFormat サンプルの実行結果

上記の例のように、StringFormat には数値書式指定文字列を指定します。数値書式指定文字列には表 4.1 のようなものがあります。

表 4.1：数値書式指定文字列

書式指定子	サポートするデータ型	説　明
"C" または "c"	すべての数値型	通貨として表示します。
"D" または "d"	整数型のみ	必要に応じて負の符号を付けて表示します。
"E" または "e"	すべての数値型	指数表記で表示します。
"F" または "f"	すべての数値型	必要に応じて負の符号を付けて整数または小数として表示します。
"G" または "g"	すべての数値型	固定小数点表記または指数表記のいずれかの最も簡潔な形式で表示します。
"N" または "n"	すべての数値型	必要に応じて負の符号を付け、桁区切り記号を付けて表示します。
"P" または "p"	すべての数値型	数値に 100 を掛けて、パーセント記号を付けて表示します。
"R" または "r"	Single, Double, BigInteger	同じ数値にラウンドトリップした文字列を表示します。
"X" または "x"	整数型のみ	16 進数文字列に変換して表示します。"0x" のプレフィックスは付きません。
その他	---	実行時に FormatException 例外がスローされます。

4.4　コントロールを変形させたい（Tips_Transform）

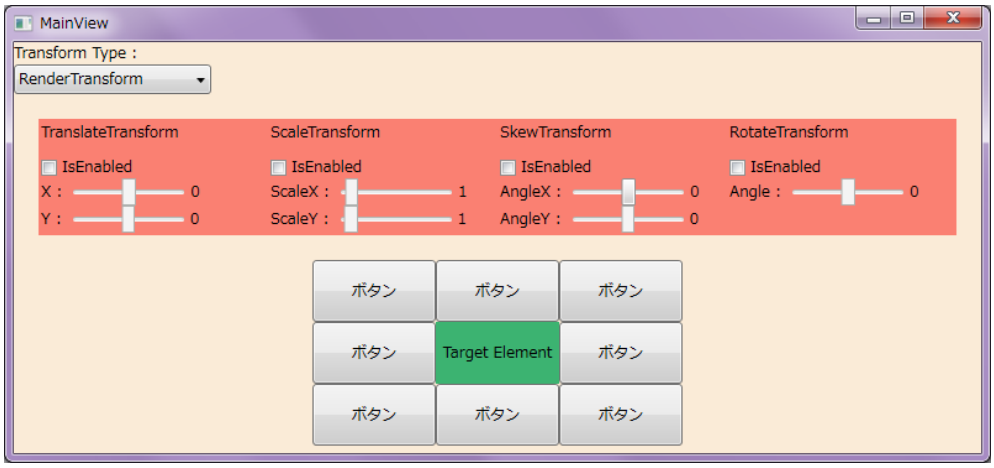
コントロールの変形には Transform クラスから派生した下表のようなクラスを 1 つまたは複数用います。ただし、これを RenderTransform プロパティに指定するか、LayoutTransform プロパティに指定するかによってその挙動が異なります。

表 4.2：コントロールを変形させるためのクラス

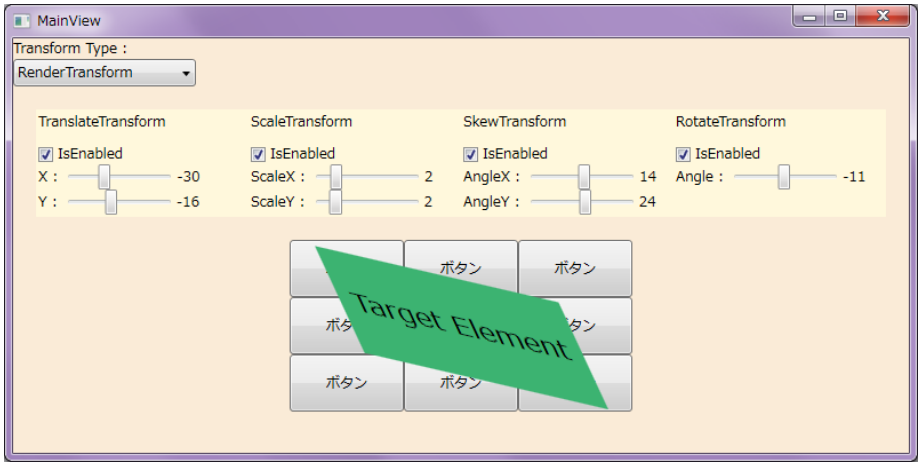
クラス名	効果	説　明
RotateTransform	回転	指定した Angle プロパティ値だけ回転します。
ScaleTransform	スケーリング	指定した ScaleX および ScaleY プロパティ値だけ水平方向および垂直方向にスケーリングします。
SkewTransform	傾斜	指定した AngleX および AngleY プロパティ値だけ水平方向および垂直方向に傾斜させます。
TranslateTransform	平行移動	指定した X および Y プロパティ値だけ水平方向および垂直方向に平行移動します。

4.4.1　RenderTransform プロパティ

RenderTransform プロパティを使用すると、レイアウトシステムに影響しないようにコントロールを変形できます。例えば Grid パネルに配置したコントロールを変形したとき、通常は与えられた領域に収まるように自動的にレイアウトされますが、RenderTransform プロパティに指定された変形によってコントロールの領域が変化しても、このコントロールが収まるようにその他のコントロールが再配置されるようなことはありません。その他のコントロールが再描画されないため処理は比較的高速になります。



(a) 変形していないときのレイアウト

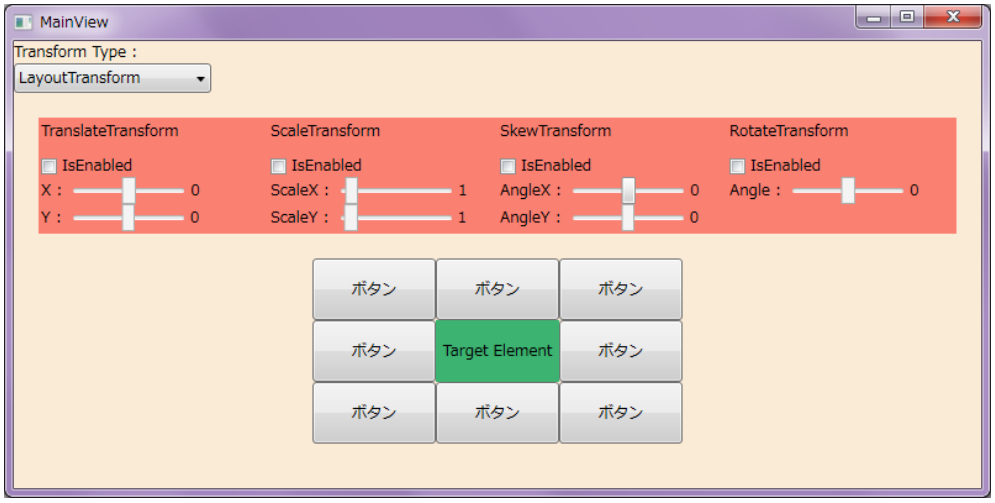


(b) 変形したときのレイアウト

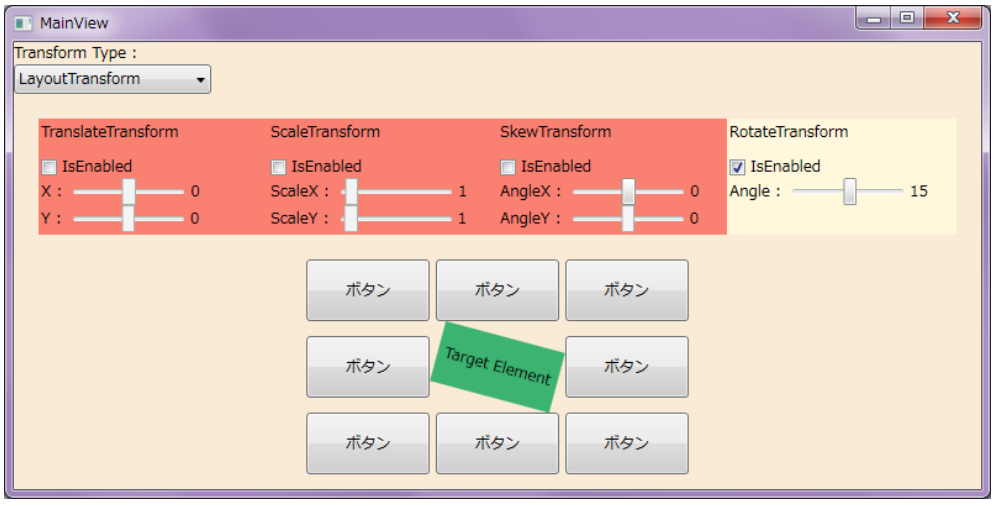
図 4.5：RenderTransform によるコントロールの変形はレイアウトに影響しない

4.4.2 LayoutTransform プロパティ

LayoutTransform プロパティを使ってコントロールを変形させた場合、レイアウトシステムがこれを認識し、関連要素を再配置します。表示結果は常に XAML に表記した通りのレイアウトになりますが、この変形をアニメーションなどで動的におこなった場合、描画処理が都度発生するため処理が比較的遅くなります。



(a) 変形していないときのレイアウト



(b) 変形したときのレイアウト

図 4.6：LayoutTransform によるコントロールの変形はレイアウトに影響する

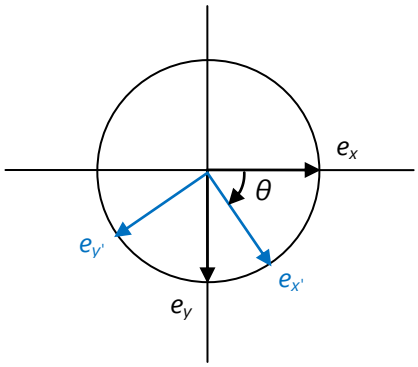
4.5 コントロールを変形したときのマウス位置を特定したい（Tips_Transform）

例えば RotateTransform で回転させたコントロールをドラッグ操作したとき、得られるマウス座標値は回転した座標系の座標値となります。これを元に戻すためには座標変換をおこなう必要があります。

4.5.1 原点を中心に座標を回転する

まず、通常の座標系 (e_x, e_y) を、原点を中心に時計回りに θ[deg] だけ回転させた座標系 $(e_{x'}, e_{y'})$ を考えます。このとき、次式が成り立ちます。

$$\begin{cases} e_{x'} = e_x \cos\theta + e_y \sin\theta \\ e_{y'} = -e_x \sin\theta + e_y \cos\theta \end{cases}$$



e_*：その方向の基底ベクトル

一般的には上方向が縦軸の正だが、アプリケーション上では下方向が正であるため、ここでは下方向を正として考える。

図 4.7：座標系の回転

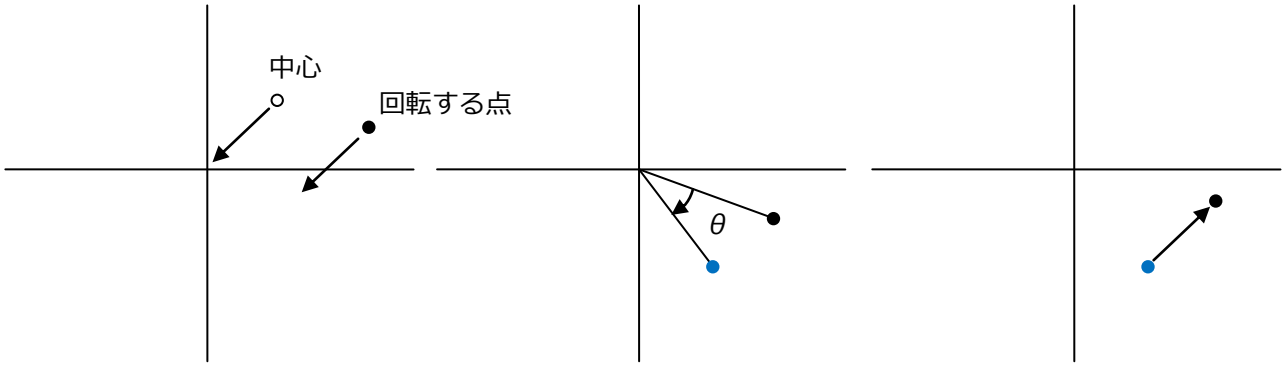
つまり、元の座標系 (e_x, e_y) における点 (x, y) を、原点を中心に時計回りに θ[deg] だけ回転させた点は、$(x\cos\theta + y\sin\theta\,,\, -x\sin\theta + y\cos\theta)$ となります。

アプリケーション上での原点は、例えばウィンドウの左上点であったり、パネルコントロールの左上点であったりします。

4.5.2 ある点を中心に座標を回転する

ある点を中心に座標を回転させる場合は、まず中心が原点になるように全体を平行移動し、回転後に元の位置に戻すようにまた平行移動します。

コントロールが回転した場合、その回転は例えばそのコントロールの左上点が中心座標となります。つまり、この回転座標を算出するときは、コントロールの左上点の座標分だけ平行移動し、回転の中心座標を原点として座標を回転させます。回転後に元に戻すように逆方向に平行移動することでコントロールの回転座標が算出できます。



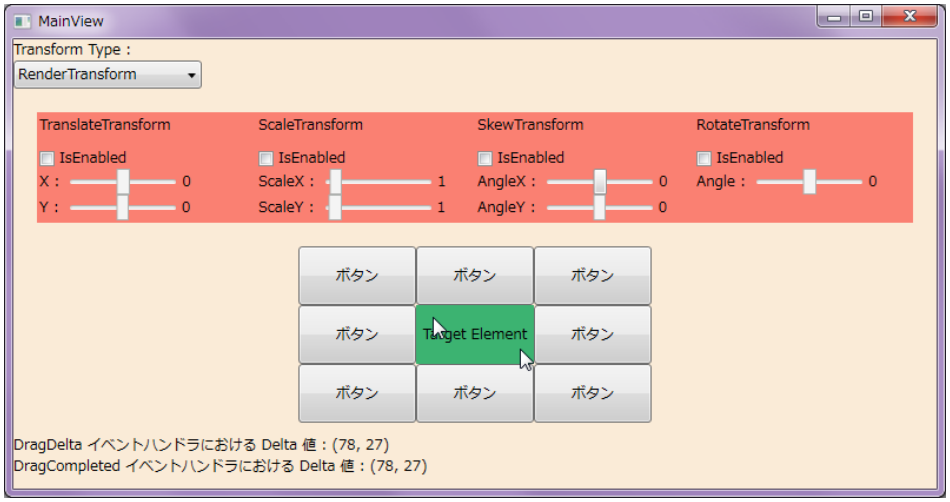
(1) 中心が原点となるように全体を平行移動する　(2) 原点を中心に回転させる　(3) 始めに平行移動した分を元に戻す

図 4.8：ある点を中心とした座標の回転

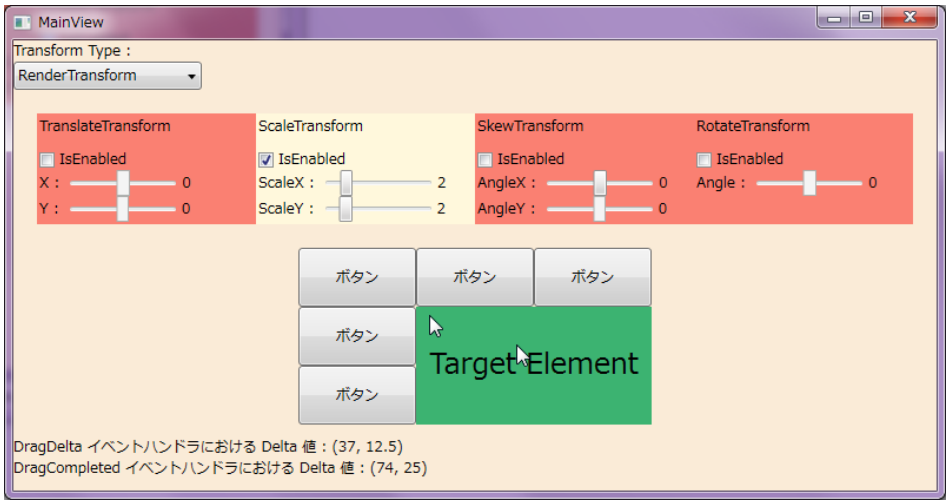
4.5.3 Thumb コントロールの DragDelta イベントと DragCompleted イベント

Thumb コントロールはドラッグ＆ドロップ操作の実装を簡略化できるコントロールで、ドラッグ操作中は DragDelta イベントが、ドロップ操作をおこなうと DragCompleted イベントが発生します。また、それぞれのイベント引数 DragDeltaEventArgs クラスおよび DragCompletedEventArgs クラスには、HorizontalChange プロパティおよび VerticalChange プロパティがあります。これらは水平方向または垂直方向のマウスの移動量を示します。ここで注意しなければならないのは、DragDelta イベントで取得できるマウスの移動量が、スケーリングおよび回転した座標系における移動量を表しているということです。

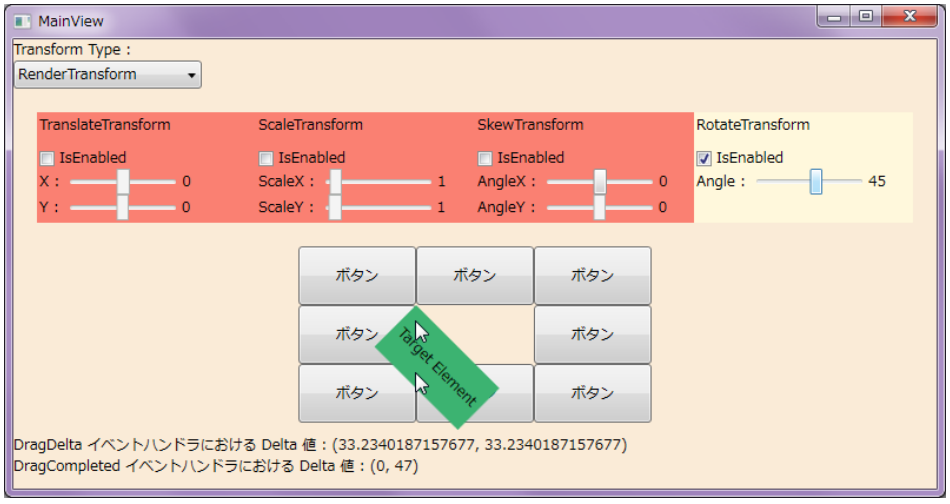
例えば図 4.9 (a) のようにコントロールの左上から右下にかけてドラッグ＆ドロップ操作をおこないます。このとき、DragDelta イベントハンドラにおける移動量と DragCompleted イベントハンドラにおける移動量はどちらも同じ値となります。ところが、同図 (b) のようにスケーリングしたコントロールに対して同じくらいの移動量のドラッグ＆ドロップ操作をおこなうと、DragDelta イベントハンドラで取得したときの移動量がスケーリングされていることがわかります。また、同図 (c) のように回転したコントロールに対してドラッグ＆ドロップ操作をおこなうと、DragDelta イベントハンドラで取得したときの移動量が回転座標系における値であることがわかります。一方、どのような場合でも DragCompleted イベントハンドラで取得できるマウスの移動量はコントロールの変形によらず元の座標系での値となっています。



(a) 通常は同じ値となる



(b) スケーリングしたコントロールでは DragDelta における移動量がスケーリングされる



(c) 回転したコントロールでは DragDelta における移動量が回転座標系における値となる

図 4.9 : 変形したコントロールで取得できるマウスの移動量

4.6 ListBox コントロールなどを物理スクロールにする (Tips_PixelScroll)

ListBox コントロールや DataGrid コントロールはアイテム個々のコンテナを表示していますが、アイテム数が多くてコントロールからはみ出る場合、スクロールバーによってスクロールして表示できるようになっています。このとき、スクロール表示の方法は、既定では論理スクロールといって、アイテムのコンテナ毎の移動量でスクロールする方法になっています。これに対し、ScrollViewer コントロールでの動作に見られるようなピクセル毎の移動量でスクロールする方法を物理スクロールといいます。

言葉で説明されるよりも実際に触って見たほうがわかりやすいので、さっそくサンプルコードを掲載します。

コード 4.9：人物データコレクションをプロパティに持つ ViewModel

MainViewModel.cs

```
namespace Tips_PixelScroll.ViewModels
{
    using System.Collections.Generic;
    using System.Linq;
    using Tips_PixelScroll.Models;

    public class MainViewModel : NotificationObject
    {
        private IEnumerable<Person> _people;
        /// <summary>
        /// 人物データコレクションを取得します。
        /// </summary>
        public IEnumerable<Person> People
        {
            get
            {
                if (this._people == null)
                {
                    this._people = Enumerable.Range(0, 1000)
                                .Select(i => new Person() { Name = "田中太郎 " + i, Age = i });
                }
                return this._people;
            }
        }
    }
}
```

コード 4.10：物理スクロールのサンプルコード

MainView.xaml

```
<Window x:Class="Tips_PixelScroll.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="500">
    <Window.Resources>
        <Style TargetType="ListBoxItem">
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="ContentControl">
                        <Border BorderBrush="Black" BorderThickness="1" Margin="2">
                            <StackPanel>
                                <TextBlock Text="{Binding Name}" />
                                <TextBlock Text="{Binding Age}" />
                            </StackPanel>
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
```

```
18              </Setter>
19          </Style>
20      </Window.Resources>
21
22      <Grid>
23          <Grid.RowDefinitions>
24              <RowDefinition Height="Auto" />
25              <RowDefinition />
26          </Grid.RowDefinitions>
27          <Grid.ColumnDefinitions>
28              <ColumnDefinition />
29              <ColumnDefinition />
30              <ColumnDefinition />
31          </Grid.ColumnDefinitions>
32
33          <TextBlock Grid.Row="0" Grid.Column="0" Text="既定の ListBox" />
34          <ListBox Grid.Row="1" Grid.Column="0" ItemsSource="{Binding People}" />
35
36          <TextBlock Grid.Row="0" Grid.Column="1" Text="仮想化して物理スクロール" />
37          <ListBox Grid.Row="1" Grid.Column="1" ItemsSource="{Binding People}"
38                   VirtualizingPanel.ScrollUnit="Pixel" />
39
40          <TextBlock Grid.Row="0" Grid.Column="2" Text="仮想化なしで物理スクロール" />
41          <ListBox Grid.Row="1" Grid.Column="2" ItemsSource="{Binding People}"
42                   VirtualizingPanel.ScrollUnit="Pixel"
43                   VirtualizingPanel.IsVirtualizing="False" />
44      </Grid>
45  </Window>
```

図 4.10：物理スクロールサンプルの実行結果

論理スクロールでは先頭のアイテムが必ず上からきちんと表示されますが、スクロールさせるとアイテムが動いていないように見えるので、スクロールしているかどうかがわかりにくくなります。これに対して物理スクロールでは、ピクセル単位でのスクロールとなるため、スクロール動作がスムーズに見えます。

どちらがいいかは時と場合によりますが、これらを使い分けられるようにしておくことが大切です。

4.7 ListBox のアイテム追加/削除のときにアニメーションする（Tips_AnimatedListBoxItem）

ListBox コントロールに動的にアイテムを追加または削除したとき、ほわっとアニメーションして表示/非表示されると視覚的に楽しいアプリになります。

ItemsControl クラスから派生している ListBox コントロールなどは、与えられたアイテムを羅列するとき、各アイテムを格納したコンテナをひょうじしています。ここでは、このこんてなにかすたむこんとろーるをしようするようにし、このかすたむこんとロールでアニメーションを実現します。

そういうわけで、カスタムコントロールとして次のような AnimatedContainer コントロールを定義します。

コード 4.11：AnimatedContainer コントロールの外観定義

Generic.AniamtedContainer.xaml

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:local="clr-namespace:Tips_AnimatedListBoxItem">
    <Style TargetType="{x:Type local:AnimatedContainer}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type local:AnimatedContainer}">
                    <Border Background="{TemplateBinding Background}"
                            BorderBrush="{TemplateBinding BorderBrush}"
                            BorderThickness="{TemplateBinding BorderThickness}">
                        <StackPanel>
                            <Button x:Name="PART_DeleteButton" />
                            <ContentPresenter />
                        </StackPanel>
                    </Border>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</ResourceDictionary>
```

コード 4.12：AnimatedContainer コントロールの内部実装定義

Generic.AniamtedContainer.xaml.cs

```
namespace Tips_AnimatedListBoxItem
{
    using System;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Input;
    using System.Windows.Media.Animation;

    /// <summary>
    /// アニメーションを含むコンテナを表します。
    /// </summary>
    [TemplatePart(Name = PART_DeleteButton, Type = typeof(Button))]
    public class AnimatedContainer : ContentControl
    {
        #region TemplatePart
        /// <summary>
        /// 削除ボタンに対する名前
        /// </summary>
        private const string PART_DeleteButton = "PART_DeleteButton";

        private Button _deleteButton;
        /// <summary>
```

```
23          /// 削除ボタンを取得または設定します。
24          /// </summary>
25          private Button DeleteButton
26          {
27              get { return this._deleteButton; }
28              set
29              {
30                  if (this._deleteButton != null)
31                  {
32                      this._deleteButton.Click -= OnDeleteButtonClick;
33                  }
34                  this._deleteButton = value;
35                  if (this._deleteButton != null)
36                  {
37                      this._deleteButton.Click += OnDeleteButtonClick;
38                  }
39              }
40          }
41
42          /// <summary>
43          /// テンプレートを適用します。
44          /// </summary>
45          public override void OnApplyTemplate()
46          {
47              base.OnApplyTemplate();
48
49              this.DeleteButton = this.Template.FindName(PART_DeleteButton, this) as Button;
50          }
51          #endregion TemplatePart
52
53          #region コンストラクタ
54          /// <summary>
55          /// 静的なコンストラクタを表します。
56          /// </summary>
57          static AnimatedContainer()
58          {
59              DefaultStyleKeyProperty.OverrideMetadata(typeof(AnimatedContainer), new
    FrameworkPropertyMetadata(typeof(AnimatedContainer)));
60          }
61          #endregion コンストラクタ
62      }
63  }
```

外観を定義している XAML はカスタムコントロール作成時に自動生成されたコードからほとんど触っておらず、Border コントロールの中に Button コントロールと ContentPresenter コントロールを含む StackPanel コントロールを追加しただけとなっています。追加したコントロールについて特に何も設定していませんが、この AnimatedContainer コントロールを使うときは、ユーザー側が ControlTemplate を与えることで自分のアプリケーションに合ったレイアウトにしてもらうため、ここでは特に外観を定義することはありません。

コードビハインドのほうでは、用意されたボタンに対するクリックイベントを購読するようにしています。イベントハンドラの実装内容については後述します。

やりたいことは、ListBox コントロールにアイテムが追加/削除されるときにアニメーションしたいということでした。まずアイテムが追加されるシーンを考えてみましょう。アイテムが追加されるということは、ListBox コントロールの中でコンテナが追加生成されるということになります。したがって、AnimatedContainer コントロールが生成されたときにアニメーションするようにすればアイテム追加時にアニメーションされるようになるでしょう。

コントロールが生成されるとき、Loaded イベントが発生し、コントロールのサイズが確定すると SizeChanged イベントが発生します。ここでは、コントロールのサイズを使用したアニメーションを使用するため、SizeChanged イベントを購読します。

コード 4.13：コンストラクタでイベントを購読する

Generic.AniamtedContainer.xaml.cs

```
1         /// <summary>
2         /// 新しいインスタンスを生成します。
3         /// </summary>
4         public AnimatedContainer()
5         {
6             this.SizeChanged += OnSizeChanged;
7         }
8
9         /// <summary>
10        /// SizeChanged イベントハンドラ
11        /// </summary>
12        /// <param name="sender">イベント発行元</param>
13        /// <param name="e">イベント引数</param>
14        private void OnSizeChanged(object sender, SizeChangedEventArgs e)
15        {
16            if (this._isAnimated)
17                return;
18
19            BeginInAnimation();
20        }
21
22        /// <summary>
23        /// アニメーション中かどうかを判別します。
24        /// </summary>
25        private bool _isAnimated;
```

コンストラクタの中でイベントを購読し、SizeChanged イベントのイベントハンドラでアニメーションを開始しています。ただし、private フィールドを使ってアニメーション中に再度実行されないようにしています。

アニメーションの詳細についてはここでは説明しません。サンプルコードの全体を最後に掲載しているので、そちらを参考にしてください。

次に、ListBox コントロールからアイテムを削除するときのアニメーションです。削除するときにアニメーションする場合、アイテムが削除されてしまうとそのコンテナが消えてしまうためアニメーションできません。つまり、アニメーションしてからアイテムを削除する、というように順序を逆にする必要があります。

ここでは、冒頭で用意したボタンの Click イベントハンドラで削除用のアニメーションを開始させます。

コード 4.14：ボタンのイベントハンドラで削除用のアニメーションを開始する

Generic.AniamtedContainer.xaml.cs

```
1         /// <summary>
2         /// 削除ボタンクリックイベントハンドラ
3         /// </summary>
4         /// <param name="sender">イベント発行元</param>
5         /// <param name="e">イベント引数</param>
6         private void OnDeleteButtonClick(object sender, RoutedEventArgs e)
7         {
8             BeginOutAnimation();
9         }
```

削除用のアニメーションが終了したら ListBox コントロールから実際にアイテムを削除する処理を実行しなければいけません。ここでは、その処理をこのコントロールに委譲できるように DeletedCommand 依存関係

プロパティを追加し、これをアニメーション終了時に実行するようにします。また、アニメーション終了時に実行させるには、Storyboard クラスの Completed イベントを利用します。

コード 4.15：アニメーション終了時に実行する DeletedCommand 依存関係プロパティの定義

Generic.AniamtedContainer.xaml.cs

```
1        #region DeletedCommand 依存関係プロパティ
2
3        /// <summary>
4        /// DeletedCommand 依存関係プロパティを定義します。
5        /// </summary>
6        public static DependencyProperty DeletedCommandProperty =
   DependencyProperty.Register("DeletedCommand", typeof(ICommand),
   typeof(AnimatedContainer), new PropertyMetadata(null));
7
8        /// <summary>
9        /// 削除ボタンクリック後に実行されるコマンドを取得または設定します。
10       /// </summary>
11       public ICommand DeletedCommand
12       {
13           get { return (ICommand)GetValue(DeletedCommandProperty); }
14           set { SetValue(DeletedCommandProperty, value); }
15       }
16
17       #endregion DeletedCommand 依存関係プロパティ
```

以上のコードを元にしたサンプルコードの全体を次に示します。

コード 4.16：AnimatedContainer コントロールのサンプルコード

Generic.AniamtedContainer.xaml.cs

```
1   namespace Tips_AnimatedListBoxItem
2   {
3       using System;
4       using System.Windows;
5       using System.Windows.Controls;
6       using System.Windows.Input;
7       using System.Windows.Media.Animation;
8
9       /// <summary>
10      /// アニメーションを含むコンテナを表します。
11      /// </summary>
12      [TemplatePart(Name = PART_DeleteButton, Type = typeof(Button))]
13      public class AnimatedContainer : ContentControl
14      {
15          #region TemplatePart
16          /// <summary>
17          /// 削除ボタンに対する名前
18          /// </summary>
19          private const string PART_DeleteButton = "PART_DeleteButton";
20
21          private Button _deleteButton;
22          /// <summary>
23          /// 削除ボタンを取得または設定します。
24          /// </summary>
25          private Button DeleteButton
26          {
27              get { return this._deleteButton; }
28              set
```

```
29              {
30                  if (this._deleteButton != null)
31                  {
32                      this._deleteButton.Click -= OnDeleteButtonClick;
33                  }
34                  this._deleteButton = value;
35                  if (this._deleteButton != null)
36                  {
37                      this._deleteButton.Click += OnDeleteButtonClick;
38                  }
39              }
40          }
41
42          /// <summary>
43          /// テンプレートを適用します。
44          /// </summary>
45          public override void OnApplyTemplate()
46          {
47              base.OnApplyTemplate();
48
49              this.DeleteButton = this.Template.FindName(PART_DeleteButton, this) as Button;
50          }
51          #endregion TemplatePart
52
53          #region コンストラクタ
54          /// <summary>
55          /// 静的なコンストラクタを表します。
56          /// </summary>
57          static AnimatedContainer()
58          {
59              DefaultStyleKeyProperty.OverrideMetadata(typeof(AnimatedContainer), new
    FrameworkPropertyMetadata(typeof(AnimatedContainer)));
60          }
61
62          /// <summary>
63          /// 新しいインスタンスを生成します。
64          /// </summary>
65          public AnimatedContainer()
66          {
67              this.Opacity = 0;
68
69              #region InAnimation 初期化
70
71              this._heightInAnimation = new DoubleAnimation()
72              {
73                  From = 0,
74                  Duration = TimeSpan.FromMilliseconds(200),
75              };
76              Storyboard.SetTargetProperty(this._heightInAnimation, new
    PropertyPath("Height"));
77
78              this._widthInAnimation = new DoubleAnimation()
79              {
80                  From = 0,
81                  Duration = TimeSpan.FromMilliseconds(200),
82              };
83              Storyboard.SetTargetProperty(this._widthInAnimation, new
    PropertyPath("Width"));
```

```
84
85              this._opacityInAnimation = new DoubleAnimation()
86              {
87                  From = 0,
88                  To = 1,
89                  BeginTime = TimeSpan.FromMilliseconds(240),
90                  Duration = TimeSpan.FromMilliseconds(200),
91              };
92              Storyboard.SetTargetProperty(this._opacityInAnimation, new
    PropertyPath("Opacity"));
93
94              this._inStoryboard = new Storyboard();
95              this._inStoryboard.Completed += (_, __) => this._isAnimated = false;
96
97              #endregion InAnimation 初期化
98
99              #region OutAnimation 初期化
100
101             this._heightOutAnimation = new DoubleAnimation()
102             {
103                 To = 0,
104                 BeginTime = TimeSpan.FromMilliseconds(240),
105                 Duration = TimeSpan.FromMilliseconds(200),
106             };
107             Storyboard.SetTargetProperty(this._heightOutAnimation, new
    PropertyPath("Height"));
108
109             this._widthOutAnimation = new DoubleAnimation()
110             {
111                 To = 0,
112                 BeginTime = TimeSpan.FromMilliseconds(240),
113                 Duration = TimeSpan.FromMilliseconds(200),
114             };
115             Storyboard.SetTargetProperty(this._widthOutAnimation, new
    PropertyPath("Width"));
116
117             this._opacityOutAnimation = new DoubleAnimation()
118             {
119                 From = 1,
120                 To = 0,
121                 Duration = TimeSpan.FromMilliseconds(200),
122             };
123             Storyboard.SetTargetProperty(this._opacityOutAnimation, new
    PropertyPath("Opacity"));
124
125             this._outStoryboard = new Storyboard();
126             this._outStoryboard.Completed += (_, __) =>
127             {
128                 this._isAnimated = false;
129                 if (this.DeletedCommand != null)
130                 {
131                     if (DeletedCommand.CanExecute(this.DataContext))
132                     {
133                         DeletedCommand.Execute(this.DataContext);
134                     }
135                 }
136             };
137
```

```
138             #endregion OutAnimation 初期化
139
140             this.SizeChanged += OnSizeChanged;
141         }
142         #endregion コンストラクタ
143
144         #region DeletedCommand 依存関係プロパティ
145
146         /// <summary>
147         /// DeletedCommand 依存関係プロパティを定義します。
148         /// </summary>
149         public static DependencyProperty DeletedCommandProperty =
    DependencyProperty.Register("DeletedCommand", typeof(ICommand),
    typeof(AnimatedContainer), new PropertyMetadata(null));
150
151         /// <summary>
152         /// 削除ボタンクリック後に実行されるコマンドを取得または設定します。
153         /// </summary>
154         public ICommand DeletedCommand
155         {
156             get { return (ICommand)GetValue(DeletedCommandProperty); }
157             set { SetValue(DeletedCommandProperty, value); }
158         }
159
160         #endregion DeletedCommand 依存関係プロパティ
161
162         #region Direction 依存関係プロパティ
163         public static readonly DependencyProperty DirectionProperty =
    DependencyProperty.Register("Direction", typeof(SizeToContent),
    typeof(AnimatedContainer), new PropertyMetadata(SizeToContent.Height));
164
165         public SizeToContent Direction
166         {
167             get { return (SizeToContent)GetValue(DirectionProperty); }
168             set { SetValue(DirectionProperty, value); }
169         }
170         #endregion Direction 依存関係プロパティ
171
172         #region イベントハンドラ
173
174         /// <summary>
175         /// 削除ボタンクリックイベントハンドラ
176         /// </summary>
177         /// <param name="sender">イベント発行元</param>
178         /// <param name="e">イベント引数</param>
179         private void OnDeleteButtonClick(object sender, RoutedEventArgs e)
180         {
181             BeginOutAnimation();
182         }
183
184         /// <summary>
185         /// SizeChanged イベントハンドラ
186         /// </summary>
187         /// <param name="sender">イベント発行元</param>
188         /// <param name="e">イベント引数</param>
189         private void OnSizeChanged(object sender, SizeChangedEventArgs e)
190         {
191             if (this._isAnimated)
```

```
192                 return;
193
194             this._heightInAnimation.To = e.NewSize.Height;
195             this._widthInAnimation.To = e.NewSize.Width;
196             BeginInAnimation();
197         }
198
199         #endregion イベントハンドラ
200
201         #region アニメーション
202         /// <summary>
203         /// 表示開始アニメーションを開始します。
204         /// </summary>
205         private void BeginInAnimation()
206         {
207             this._inStoryboard.Children.Clear();
208             switch (this.Direction)
209             {
210                 case SizeToContent.Height:
211                     this._inStoryboard.Children.Add(this._heightInAnimation);
212                     break;
213
214                 case SizeToContent.Manual:
215                     break;
216
217                 case SizeToContent.Width:
218                     this._inStoryboard.Children.Add(this._widthInAnimation);
219                     break;
220
221                 case SizeToContent.WidthAndHeight:
222                     this._inStoryboard.Children.Add(this._heightInAnimation);
223                     this._inStoryboard.Children.Add(this._widthInAnimation);
224                     break;
225             }
226             this._inStoryboard.Children.Add(this._opacityInAnimation);
227
228             this._isAnimated = true;
229             this.BeginStoryboard(this._inStoryboard);
230         }
231
232         /// <summary>
233         /// 非表示アニメーションを開始します。
234         /// </summary>
235         private void BeginOutAnimation()
236         {
237             this._outStoryboard.Children.Clear();
238             switch (this.Direction)
239             {
240                 case SizeToContent.Height:
241                     this._outStoryboard.Children.Add(this._heightOutAnimation);
242                     break;
243
244                 case SizeToContent.Manual:
245                     break;
246
247                 case SizeToContent.Width:
248                     this._outStoryboard.Children.Add(this._widthOutAnimation);
249                     break;
```

```
250
251                 case SizeToContent.WidthAndHeight:
252                     this._outStoryboard.Children.Add(this._heightOutAnimation);
253                     this._outStoryboard.Children.Add(this._widthOutAnimation);
254                     break;
255             }
256             this._outStoryboard.Children.Add(this._opacityOutAnimation);
257 
258             this._isAnimated = true;
259             this.BeginStoryboard(this._outStoryboard);
260         }
261         #endregion アニメーション
262 
263         #region private フィールド
264 
265         /// <summary>
266         /// アニメーション中かどうかを判別します。
267         /// </summary>
268         private bool _isAnimated;
269 
270         /// <summary>
271         /// 表示されるときのアニメーション用ストーリーボード
272         /// </summary>
273         private Storyboard _inStoryboard;
274 
275         /// <summary>
276         /// 表示されるときの高さアニメーション
277         /// </summary>
278         private DoubleAnimation _heightInAnimation;
279 
280         /// <summary>
281         /// 表示されるときの幅アニメーション
282         /// </summary>
283         private DoubleAnimation _widthInAnimation;
284 
285         /// <summary>
286         /// 表示されるときの透明度アニメーション
287         /// </summary>
288         private DoubleAnimation _opacityInAnimation;
289 
290         /// <summary>
291         /// 非表示になるときのアニメーション用ストーリーボード
292         /// </summary>
293         private Storyboard _outStoryboard;
294 
295         /// <summary>
296         /// 非表示になるときの高さアニメーション
297         /// </summary>
298         private DoubleAnimation _heightOutAnimation;
299 
300         /// <summary>
301         /// 非表示になるときの幅アニメーション
302         /// </summary>
303         private DoubleAnimation _widthOutAnimation;
304 
305         /// <summary>
306         /// 非表示になるときの透明度アニメーション
307         /// </summary>
```

```
308         private DoubleAnimation _opacityOutAnimation;
309
310         #endregion private フィールド
311     }
312 }
```

このカスタムコントロールを使用したサンプルアプリケーションのコードを以下に掲載します。

コード 4.17：サンプルアプリケーションの MainViewModel

MainViewModel.cs

```
1  namespace Tips_AnimatedListBoxItem.ViewModels
2  {
3      using System;
4      using System.Collections.ObjectModel;
5      using System.ComponentModel;
6      using System.Runtime.CompilerServices;
7
8      internal class MainViewModel : NotificationObject
9      {
10         private ObservableCollection<string> _stringCollection = new
       ObservableCollection<string>();
11         /// <summary>
12         /// コレクションデータを取得します。
13         /// </summary>
14         public ObservableCollection<string> StringCollection
15         {
16             get { return this._stringCollection; }
17         }
18
19         /// <summary>
20         /// コレクションデータ数を取得します。
21         /// </summary>
22         public int Count
23         {
24             get { return this.StringCollection.Count; }
25         }
26
27         private DelegateCommand _addCommand;
28         /// <summary>
29         /// コレクション追加コマンドを取得します。
30         /// </summary>
31         public DelegateCommand AddCommand
32         {
33             get
34             {
35                 return this._addCommand ?? (this._addCommand = new DelegateCommand(_ =>
36                 {
37                     this._counter++;
38                     Add(string.Format("私がアイテム No.{0} だ。", this._counter));
39                 }));
40             }
41         }
42
43         private DelegateCommand _deleteCommand;
44         /// <summary>
45         /// コレクション削除コマンドを取得します。
46         /// </summary>
```

```
47          public DelegateCommand DeleteCommand
48          {
49              get
50              {
51                  return this._deleteCommand ?? (this._deleteCommand = new DelegateCommand(p
    =>
52                  {
53                      Delete(p as string);
54                  }));
55              }
56          }
57
58          /// <summary>
59          /// カウンタ
60          /// </summary>
61          private int _counter;
62
63          /// <summary>
64          /// 乱数発生器
65          /// </summary>
66          private Random _random = new Random();
67
68          /// <summary>
69          /// コレクションにアイテムを追加します。
70          /// </summary>
71          /// <param name="item">追加するアイテムを指定します。</param>
72          private void Add(string item)
73          {
74              this.StringCollection.Insert(this._random.Next(0,
    this.StringCollection.Count), item);
75              RaisePropertyChanged("Count");
76          }
77
78          /// <summary>
79          /// コレクションからアイテムを削除します。
80          /// </summary>
81          /// <param name="item"></param>
82          private void Delete(string item)
83          {
84              this.StringCollection.Remove(item);
85              RaisePropertyChanged("Count");
86          }
87      }
88  }
```

コード 4.18：サンプルアプリケーションの MainView

MainView.xaml

```
1   <Window x:Class="Tips_AnimatedListBoxItem.Views.MainView"
2           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4           xmlns:local="clr-namespace:Tips_AnimatedListBoxItem"
5           Title="MainView" Height="300" Width="300">
6       <Grid>
7           <Grid.RowDefinitions>
8               <RowDefinition Height="Auto" />
9               <RowDefinition />
10          </Grid.RowDefinitions>
11
```

```
12          <StackPanel>
13              <Button Content="Add" Command="{Binding AddCommand}" />
14              <TextBlock Text="{Binding Count, StringFormat='{}アイテムが {0} 個登録されています。'}" />
15          </StackPanel>
16
17          <ListBox Grid.Row="1" ItemsSource="{Binding StringCollection}">
18              <ListBox.ItemContainerStyle>
19                  <Style TargetType="{x:Type ListBoxItem}">
20                      <Setter Property="Template">
21                          <Setter.Value>
22                              <ControlTemplate TargetType="{x:Type ContentControl}">
23                                  <Border Background="{TemplateBinding Background}">
24                                      <local:AnimatedContainer DeletedCommand="{Binding DataContext.DeleteCommand, RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type ItemsControl}}}"
25                                                               Direction="Height">
26                                          <local:AnimatedContainer.Template>
27                                              <ControlTemplate TargetType="{x:Type local:AnimatedContainer}">
28                                                  <StackPanel Orientation="Horizontal">
29                                                      <Button x:Name="PART_DeleteButton"
30                                                              Content="Delete" Margin="2"
31                                                              />
32                                                      <TextBlock Text="{Binding .}" VerticalAlignment="Center" />
33                                                  </StackPanel>
34                                              </ControlTemplate>
35                                          </local:AnimatedContainer.Template>
36                                      </local:AnimatedContainer>
37                                  </Border>
38                              </ControlTemplate>
39                          </Setter.Value>
40                      </Setter>
41                      <Style.Triggers>
42                          <Trigger Property="IsMouseOver" Value="True">
43                              <Setter Property="Background" Value="LightGray" />
44                          </Trigger>
45                          <Trigger Property="IsSelected" Value="True">
46                              <Setter Property="Background" Value="Plum" />
47                          </Trigger>
48                      </Style.Triggers>
49                  </Style>
50              </ListBox.ItemContainerStyle>
51          </ListBox>
52      </Grid>
53  </Window>
```

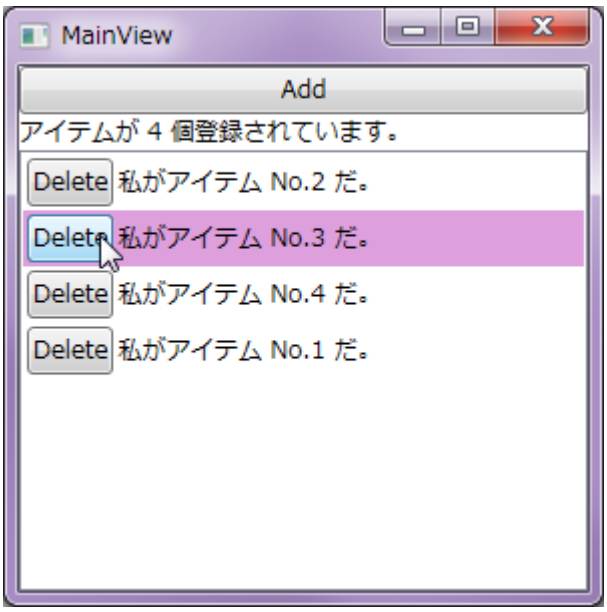


図 4.11：サンプルアプリケーションの外観

Add ボタンを押すと ListBox コントロールにアイテムがランダムの順序で追加され、アイテム中の Delete ボタンを押すと該当するアイテムが削除されます。アイテム追加/削除されうときにはアニメーションが実行されるようになっています。

4.8 添付ビヘイビアを作成する (Tips_Behavior)

添付ビヘイビアとは、主にプロパティ変更時のコントロールの振る舞いを決めるためのものです。この機能を実現するために、添付プロパティと呼ばれるプロパティを活用します。

4.8.1 添付プロパティとは

WPF のプロパティシステムには依存関係プロパティの他に、添付プロパティと呼ばれるものがあります。添付プロパティとは、あるオブジェクトに対してそのプロパティ値を添付するものです。例えば、Grid.Row プロパティは、Grid コントロールのどの行に表示するかを決めるための添付プロパティです。これは Grid コントロールそのものに Row というプロパティがあるわけではなく、Grid コントロールに属する他のコントロールに添付して使います。次のコードを見てみましょう。

コード 4.19：Grid.Row 添付プロパティの使用例

MainView.xaml

```
<Window x:Class="Tips_Behavior.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>

        <Button Grid.Row="1" Content="Click me." />
    </Grid>
</Window>
```

このコードでは、Button コントロールに Grid.Row プロパティを添付しています。こうすることで、親要素である Grid コントロールが 1 行目に Button コントロールを表示するように認識できます。

4.8.2 添付プロパティの作成

添付ビヘイビアを作成するためには、独自の添付プロパティを作成する必要があります。ここでは、SampleBehavior という名前のクラスを定義し、その中に添付プロパティを作成します。
作成した添付ビヘイビアは XAML 上で使用する View の要素となるので、作成するソースは Views フォルダの中にまとめておくとわかりやすくなります。

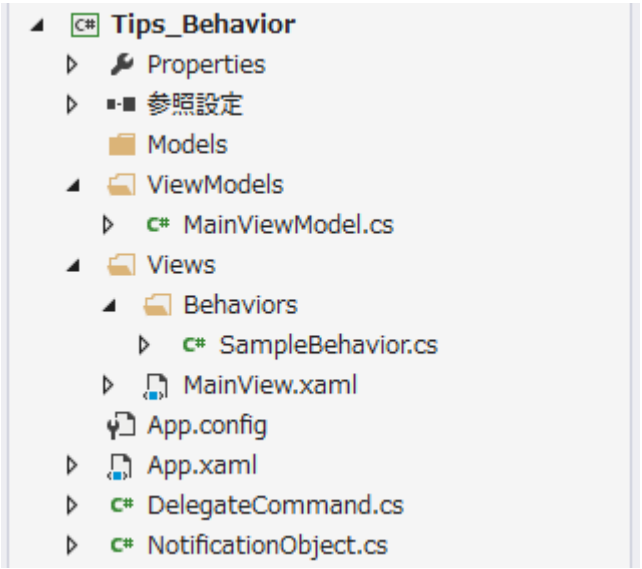


図 4.12：Views フォルダに Behaviors フォルダを追加

添付プロパティを定義するには、DependencyProperty.RegisterAttached() メソッドを使用します。例えば bool 型で IsEnabled という名前の添付プロパティは次のように記述することで定義できます。

コード 4.20：IsEnabled 添付プロパティの作成例

SampleBehavior.cs

```
1  namespace Tips_Behavior.Views.Behaviors
2  {
3      using System.Windows;
4      using System.Windows.Controls;
5
6      /// <summary>
7      /// サンプルのビヘイビアを表します。
8      /// </summary>
9      public class SampleBehavior
10     {
11         /// <summary>
12         /// IsEnabled 添付プロパティの定義
13         /// </summary>
14         public static readonly DependencyProperty IsEnabledProperty =
       DependencyProperty.RegisterAttached("IsEnabled", typeof(bool), typeof(SampleBehavior),
       new FrameworkPropertyMetadata(false, OnIsEnabledPropertyChanged));
15
16         /// <summary>
17         /// IsEnabled 添付プロパティを取得します。
18         /// </summary>
19         /// <param name="target">対象とする DependencyObject を指定します。</param>
20         /// <returns>取得した値を返します。</returns>
21         public static bool GetIsEnabled(DependencyObject target)
22         {
23             return (bool)target.GetValue(IsEnabledProperty);
24         }
25
26         /// <summary>
27         /// IsEnabled 添付プロパティを設定します。
28         /// </summary>
29         /// <param name="target">対象とする DependencyObject を指定します。</param>
30         /// <param name="value">設定する値を指定します。</param>
31         public static void SetIsEnabled(DependencyObject target, bool value)
32         {
33             target.SetValue(IsEnabledProperty, value);
34         }
35
36         /// <summary>
37         /// IsEnabled 添付プロパティ変更イベントハンドラ
38         /// </summary>
39         /// <param name="sender">イベント発行元</param>
40         /// <param name="e">イベント引数</param>
41         private static void OnIsEnabledPropertyChanged(DependencyObject sender,
       DependencyPropertyChangedEventArgs e)
42         {
43         }
44     }
45 }
```

13 行目で IsEnabled 添付プロパティを定義しています。変数名は任意で構いませんが、可読性を考慮して、プロパティ名に "Property" を付加した IsEnabledProeprty などといった名前にすることが一般的です。RegisterAttached() メソッドの第 1 引数にプロパティ名 "IsEnabled" を与えます。

第 2 引数にはプロパティの型を System.Type クラスで指定します。ここでは bool 型のプロパティを定義したいので、typeof(bool) を指定しています。
第 3 引数には、このプロパティがどのクラスに属するのかを System.Type クラスで指定します。ここでは SampleBehavior クラスを指定したので、追加した IsEnabled 添付プロパティは、SampleBehavior.IsEnabled として認識されるようになります。
第 4 引数以降は特に指定しなくてもプロパティ定義としては成立しますが、このプロパティの既定値や、値が変化したときのコールバックなどが指定でき、ビヘイビアを使用するためにはこの部分が重要になるので、ここでは FrameworkPropertyMetadata クラスを第 4 引数として指定しています。
FrameworkPropertyMetadata クラスのコンストラクタに対する第 1 引数に IsEnabled プロパティの既定値、第 2 引数にプロパティ値変更時のコールバックメソッドを指定しています。

添付プロパティを取得したり設定したりするためには、規定された名前のメソッドを用意する必要があります。取得するためのメソッドはプロパティ名の頭に "Get" を付けた名前で、ここでは GetIsEnabled という名前になります。設定するためのメソッドはプロパティ名の頭に "Set" を付けた名前で、ここでは SetIsEnabled という名前になります。それぞれのメソッドを 15～33 行目のように定義します。

以上で添付プロパティの定義は完了です。試しに XAML 上で添付プロパティを設定してみましょう。

コード 4.21：IsEnabled 添付プロパティを添付する

MainView.xaml

```
 1 <Window x:Class="Tips_Behavior.Views.MainView"
 2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4         xmlns:b="clr-namespace:Tips_Behavior.Views.Behaviors"
 5         Title="MainView" Height="120" Width="300">
 6     <StackPanel>
 7         <Button Content="Click me." b:SampleBehavior.IsEnabled="False" />
 8         <Button Content="Click me." b:SampleBehavior.IsEnabled="True" />
 9         <TextBlock Text="Click me." b:SampleBehavior.IsEnabled="True" />
10     </StackPanel>
11 </Window>
```

作成した IsEnabled 添付プロパティは SampleBehavior クラスに属しているので、このクラスが属する名前空間を 4 行目のように定義する必要があります。このエイリアスを用いて、7～9 行目のように IsEnabled 添付プロパティを任意のコントロールに添付することができます。
ここまでのサンプルコードでは、IsEnabled 添付プロパティを作成しただけなので、このプロパティが設定されたコントロールには特に変化はありません。

4.8.3 ビヘイビアの作成

最後に、コントロールに対するビヘイビアの一例をサンプルとして作成してみます。

IsEnabled 添付プロパティの変更イベントハンドラを次のように変更し、他のメソッドも追加します。

コード 4.22：SampleBehavior クラスのビヘイビア

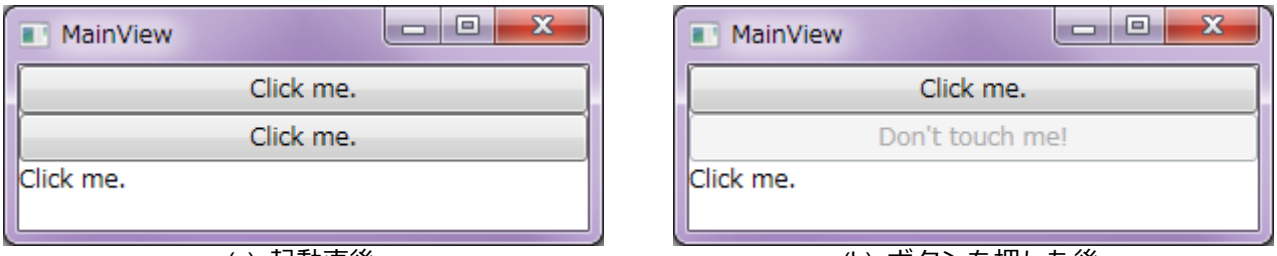
SampleBehavior.cs

```
36         /// <summary>
37         /// IsEnabled 添付プロパティ変更イベントハンドラ
38         /// </summary>
39         /// <param name="sender">イベント発行元</param>
40         /// <param name="e">イベント引数</param>
41         private static void OnIsEnabledPropertyChanged(DependencyObject sender,
   DependencyPropertyChangedEventArgs e)
42         {
43             var control = sender as Button;
44             if (control == null)
45                 return;
46
```

```
47              var isEnabled = GetIsEnabled(control);
48              if (isEnabled)
49              {
50                  control.Click += OnClick;
51              }
52              else
53              {
54                  control.Click -= OnClick;
55              }
56          }
57
58          /// <summary>
59          /// Click イベントハンドラ
60          /// </summary>
61          /// <param name="sender">イベント発行元</param>
62          /// <param name="e">イベント引数</param>
63          private static void OnClick(object sender, RoutedEventArgs e)
64          {
65              var control = sender as Button;
66              if (control == null)
67                  return;
68
69              control.Content = "Don't touch me!";
70              control.IsEnabled = false;
71          }
```

SampleBehavior.IsEnabled 添付プロパティは System.Windows.Button クラスに対して有効に機能するもので、IsEnabled 添付プロパティ値が true のとき、クリックイベントに登録したメソッドが処理されるようになっています。

このようなビヘイビアを定義してアプリケーションを実行すると次のようになります。



(a) 起動直後　　(b) ボタンを押した後

図 4.13：添付ビヘイビアサンプルの実行結果

IsEnabled 添付プロパティが false であるボタンのほうは押しても何も起きませんが、true であるボタンは指定されたイベントハンドラが処理されていることがわかります。また、Button コントロールではないものに true が指定されていても無視されていることがわかります。

4.9 DataGrid コントロールの行ヘッダに行番号を表示する (Tips_DataGrid1)

DataGrid コントロールによって表形式でデータを表示したとき、その行ヘッダに行番号を入れたい場合は多々あります。このような機能は使い回しが効くように、添付ビヘイビアで作成しておくと便利です。

MainViewModel クラスと MainView を次のように定義します。

コード 4.23：人物データコレクションを持つ MainViewModel

MainViewModel.cs

```
1  namespace Tips_DataGrid1.ViewModels
2  {
3      using System.Collections.ObjectModel;
4      using System.Linq;
5      using Tips_DataGrid1.Models;
6
7      public class MainViewModel : NotificationObject
8      {
9          public MainViewModel()
10         {
11             this.People = new ObservableCollection<Person>(Enumerable.Range(0,
       10).Select(x => new Person()
12             {
13                 Name = "田中 " + x + "太郎",
14                 Age = x,
15                 Gender = x % 3 != 0 ? Gender.Male : Gender.Female,
16                 IsAuthenticated = x % 4 != 0,
17             }));
18         }
19
20         private ObservableCollection<Person> _people;
21         /// <summary>
22         /// 人物データコレクションを取得します。
23         /// </summary>
24         public ObservableCollection<Person> People
25         {
26             get { return this._people; }
27             private set { SetProperty(ref this._people, value); }
28         }
29     }
30 }
```

コード 4.24：人物データコレクションを DataGrid で表示する MainView

MainView.xaml

```
1  <Window x:Class="Tips_DataGrid1.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"
5          Height="300" Width="300"
6          WindowStartupLocation="CenterScreen">
7      <DataGrid ItemsSource="{Binding People}" />
8  </Window>
```

DataGrid コントロールによって People プロパティの持つ人物データコレクションが表形式で表示されることになります。ここまでのコードによる実行結果は下図のようになります。

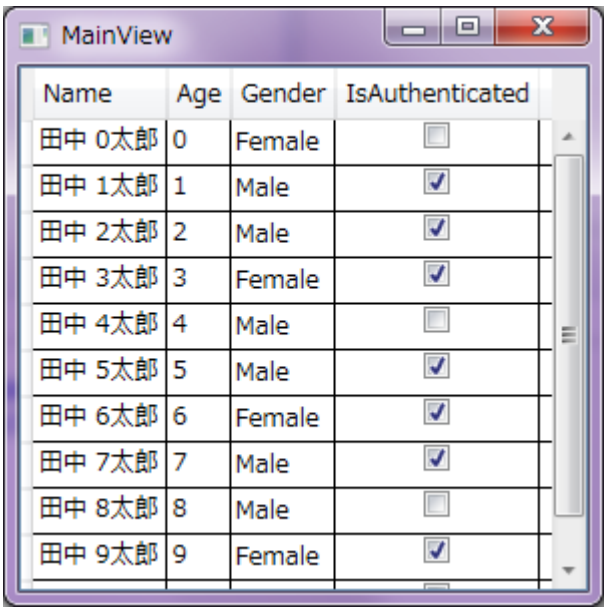


図 4.14：Person クラスが自動的に表形式で表示される

ここで、次のような添付ビヘイビアを定義します。添付ビヘイビアに関しては「4.8 添付ビヘイビアを作成する (Tips_Behavior)」を参照してください。

コード 4.25：DataGridBehavior 添付ビヘイビアの定義

DataGridBehavior.cs

```
namespace Tips_DataGrid1.Views.Behaviors
{
    using System;
    using System.Collections.Generic;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Controls.Primitives;
    using System.Windows.Media;

    /// <summary>
    /// <c>System.Windows.Controls.DataGrid</c> コントロールに関する添付ビヘイビアを表します。
    /// </summary>
    public class DataGridBehavior
    {
        #region DisplayRowNumber 添付プロパティ
        /// <summary>
        /// DisplayRowNumber 添付プロパティの定義
        /// </summary>
        public static DependencyProperty DisplayRowNumberProperty = DependencyProperty.RegisterAttached("DisplayRowNumber", typeof(int?), typeof(DataGridBehavior), new FrameworkPropertyMetadata(null, OnDisplayRowNumberChanged));

        /// <summary>
        /// DisplayRowNumber 添付プロパティを取得します。
        /// </summary>
        /// <param name="target">対象とする DependencyObject を指定します。</param>
        /// <returns>取得した値を返します。</returns>
        public static int? GetDisplayRowNumber(DependencyObject target)
        {
            return (int?)target.GetValue(DisplayRowNumberProperty);
```

```
29          }
30
31          /// <summary>
32          /// DisplayRowNumber 添付プロパティを設定します。
33          /// </summary>
34          /// <param name="target">対象とする DependencyObject を指定します。</param>
35          /// <param name="value">設定値を指定します。</param>
36          public static void SetDisplayRowNumber(DependencyObject target, int? value)
37          {
38              target.SetValue(DisplayRowNumberProperty, value);
39          }
40
41          /// <summary>
42          /// DisplayRowNumber 添付プロパティ変更イベントハンドラ
43          /// </summary>
44          /// <param name="sender">イベント発行元</param>
45          /// <param name="e">イベント引数</param>
46          private static void OnDisplayRowNumberChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
47          {
48              var dataGrid = sender as DataGrid;
49              if (dataGrid == null)
50                  return;
51
52              if (e.NewValue != null)
53              {
54                  var displayNumber = GetDisplayRowNumber(dataGrid);
55
56                  EventHandler<DataGridRowEventArgs> loadedRowHandler = null;
57                  loadedRowHandler = (object _, DataGridRowEventArgs ea) =>
58                  {
59                      if (displayNumber == null)
60                      {
61                          dataGrid.LoadingRow -= loadedRowHandler;
62                          return;
63                      }
64                      ea.Row.Header = ea.Row.GetIndex() + displayNumber;
65                  };
66                  dataGrid.LoadingRow += loadedRowHandler;
67
68                  ItemsChangedEventHandler itemsChangedHandler = null;
69                  itemsChangedHandler = (object _, ItemsChangedEventArgs ea) =>
70                  {
71                      if (displayNumber == null)
72                      {
73                          dataGrid.ItemContainerGenerator.ItemsChanged -=
    itemsChangedHandler;
74                          return;
75                      }
76                      // 子要素の DataGridRow クラスに対してのみヘッダ情報を書き換える
77                      GetVisualChildCollection<DataGridRow>(dataGrid).
78                          ForEach(d => d.Header = d.GetIndex() + displayNumber);
79                  };
80                  dataGrid.ItemContainerGenerator.ItemsChanged += itemsChangedHandler;
81              }
82          }
83          #endregion DisplayRowNumber 添付プロパティ
84
```

```
85          /// <summary>
86          /// 指定された型の子要素をリストとして取得します。
87          /// </summary>
88          /// <typeparam name="T">リストアップする型を指定します。</typeparam>
89          /// <param name="parent">子要素を持つ親を指定します。</param>
90          /// <returns>指定された型の子要素のみを集めたリストを返します。</returns>
91          private static List<T> GetVisualChildCollection<T>(object parent)
92              where T : Visual
93          {
94              var visualCollection = new List<T>();
95              GetVisualChildCollection(parent as DependencyObject, visualCollection);
96              return visualCollection;
97          }
98
99          /// <summary>
100         /// 指定された型の子要素を与えられたリストに追加します。
101         /// </summary>
102         /// <typeparam name="T">リストアップする型を指定します。</typeparam>
103         /// <param name="parent">子要素を持つ親を指定します。</param>
104         /// <param name="visualCollection">リストアップするためのリストを指定します。
    </param>
105         private static void GetVisualChildCollection<T>(DependencyObject parent, List<T>
    visualCollection)
106             where T : Visual
107         {
108             int count = VisualTreeHelper.GetChildrenCount(parent);
109             for (int i = 0; i < count; i++)
110             {
111                 DependencyObject child = VisualTreeHelper.GetChild(parent, i);
112                 if (child is T)
113                 {
114                     visualCollection.Add(child as T);
115                 }
116                 if (child != null)
117                 {
118                     GetVisualChildCollection(child, visualCollection);
119                 }
120             }
121         }
122     }
123 }
```

非常に長いですが、DisplayRowNumber という名前の添付プロパティを持っており、このプロパティに指定された数値を始めとして行ヘッダをナンバリングするようにしています。

DataGrid コントロールは各行を DataGridRow コントロールをコンテナとして表現しています。したがって、上記の添付ビヘイビアの中では、DataGrid が持つ DataGridRow コントロールを抽出し、それぞれの DataGridRow.Header プロパティにアクセスして行ヘッダを変更しています。

どの DataGridRow コントロールが何番目であるかは DataGridRow クラスが持つ GetIndex() メソッドで取得しています。GetIndex() メソッドは DataGrid コントロールの Items プロパティを参照して何番目かを取得しているため、列ヘッダをクリックすることなどによるソーティングがおこなわれたとしても、一行目は必ず 0 となります。

上記のコードでは、DisplayRowNumber 添付プロパティを基準として行番号を振っているため、任意の数値から開始させることができます。

この添付ビヘイビアの使用例は次のようになります。

コード 4.26：DataGridBehavior 添付ビヘイビアの使用例

MainView.xaml

```
1  <Window x:Class="Tips_DataGrid1.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:b="clr-namespace:Tips_DataGrid1.Views.Behaviors"
5          Title="MainView"
6          Height="300" Width="300"
7          WindowStartupLocation="CenterScreen">
8      <DataGrid ItemsSource="{Binding People}" b:DataGridBehavior.DisplayRowNumber="1" />
9  </Window>
```

添付ビヘイビアを使用するために名前空間 "b" の定義を追加しています。後は通常の添付プロパティを設定する要領で DataGrid コントロールに DisplayRowNumber 添付プロパティを記述します。

このコードの実行結果は次のようになります。DisplayRowNumber 添付プロパティに 1 を指定したため、行番号は 1 から始まっています。また、あくまでも行番号であるため、列をソートしたとしても、この番号は変わらず 1 から順に表示されるようになります。

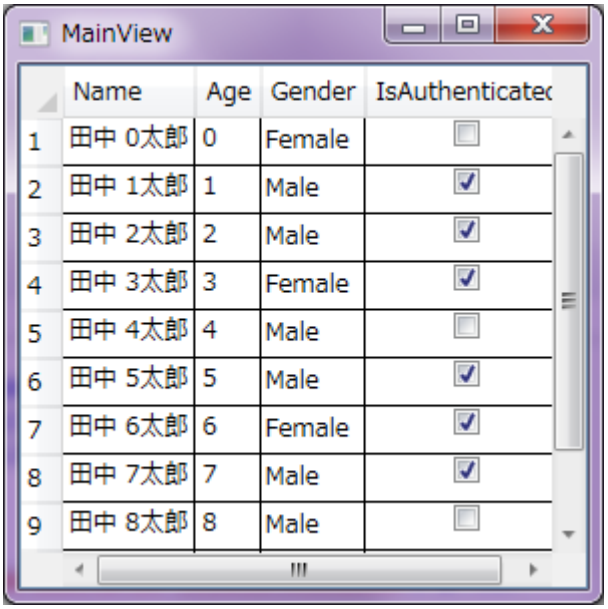
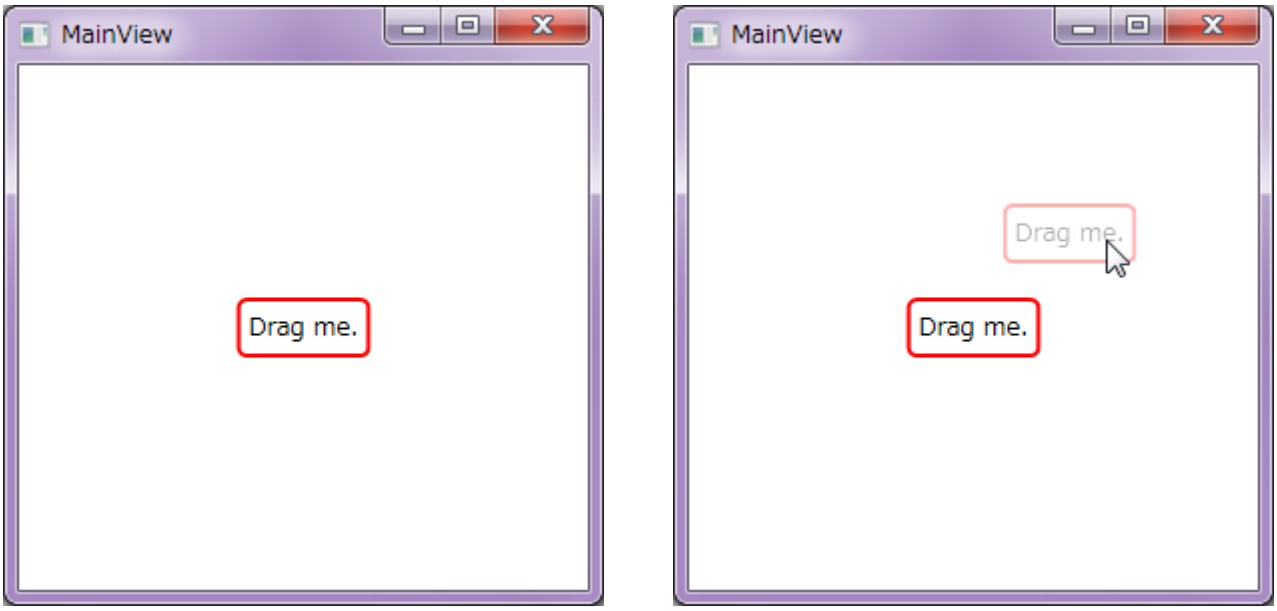


図 4.15：DisplayRowNumber 添付プロパティによる行番号の表示

4.10 ドラッグ操作でゴーストを表示する (Tips_Adorner)

ドラッグ操作を実装したとき、ドラッグ対象のコントロールが半透明になってマウスポインタに追従するようになると、操作感が格段に良くなります。ここでは Adorner クラスを使用して実現する方法を紹介します。

ここでは最終的に図 4.16 のような動作が実現できるようなサンプルを紹介します。



(a) Border 枠で囲まれた TextBlock (b) Border をドラッグするとゴーストが着いてくる

図 4.16：ドラッグ操作でゴーストが表示される

4.10.1 Adorner クラス

Adorner クラスは、UIElement クラスを装飾するための FrameworkElement クラスの派生クラスです。また、Adorner クラスを表示するには、装飾層と呼ばれるレイヤーが必要になり、これは AdornerDecorator クラスによって提供されます。AdornerDecorator クラスは Grid コントロールなどの標準的なパネルコントロールに含まれているため、意図的に AdornerDecorator を XAML に記述することはあまりありません。

4.10.2 ゴーストを表示するための Adorner 派生クラス

まずゴーストを表示するために次のようなクラスを定義します。

コード 4.27：ゴースト表示するためのクラスの定義

GhostAdorner.cs

```
 1  /// <summary>
 2  /// ゴーストを表示する装飾用コントロールを表します。
 3  /// </summary>
 4  internal class GhostAdorner : Adorner
 5  {
 6      /// <summary>
 7      /// 新しいインスタンスを生成します。
 8      /// </summary>
 9      /// <param name="visual">装飾する要素を指定します。</param>
10      /// <param name="adornedElement">装飾に用いる要素を指定します。</param>
11      /// <param name="point">装飾を表示する位置を、装飾する要素に対する相対位置として指定します。</param>
12      /// <param name="offset">装飾を表示する位置に対するオフセットを指定します。</param>
13      public GhostAdorner(Visual visual, UIElement adornedElement, Point point, Point offset)
```

```
14          : base(adornedElement)
15      {
16          this._layer = AdornerLayer.GetAdornerLayer(visual);
17          this.CurrentPoint = point;
18          this.Offset = offset;
19
20          Attach();
21      }
22
23      #region 依存関係プロパティ
24
25      /// <summary>
26      /// CurrentPoint 依存関係プロパティの定義
27      /// </summary>
28      public static readonly DependencyProperty CurrentPointProperty =
    DependencyProperty.Register("CurrentPoint", typeof(Point), typeof(GhostAdorner), new
    FrameworkPropertyMetadata(default(Point),
    FrameworkPropertyMetadataOptions.AffectsRender));
29
30      /// <summary>
31      /// ゴーストの表示位置を取得または設定します。
32      /// </summary>
33      public Point CurrentPoint
34      {
35          get { return (Point)GetValue(CurrentPointProperty); }
36          set { SetValue(CurrentPointProperty, value); }
37      }
38
39      /// <summary>
40      /// Offset 依存関係プロパティの定義
41      /// </summary>
42      public static readonly DependencyProperty OffsetProperty =
    DependencyProperty.Register("Offset", typeof(Point), typeof(GhostAdorner), new
    FrameworkPropertyMetadata(default(Point),
    FrameworkPropertyMetadataOptions.AffectsRender));
43
44      /// <summary>
45      /// ゴーストの表示位置のオフセットを取得または設定します。
46      /// </summary>
47      public Point Offset
48      {
49          get { return (Point)GetValue(OffsetProperty); }
50          set { SetValue(OffsetProperty, value); }
51      }
52
53      #endregion 依存関係プロパティ
54
55      #region 公開メソッド
56
57      /// <summary>
58      /// アタッチします。
59      /// </summary>
60      public void Attach()
61      {
62          if (this._layer != null)
63          {
64              if (!this._isAttached)
65              {
```

```
66                  this._layer.Add(this);
67                  this._isAttached = true;
68              }
69          }
70      }
71
72      /// <summary>
73      /// デタッチします。
74      /// </summary>
75      public void Detach()
76      {
77          if (this._layer != null)
78          {
79              if (this._isAttached)
80              {
81                  this._layer.Remove(this);
82                  this._isAttached = false;
83              }
84          }
85      }
86
87      #endregion 公開メソッド
88
89      #region 描画オーバーライド
90
91      /// <summary>
92      /// 描画処理のオーバーライド
93      /// </summary>
94      /// <param name="drawingContext">描画先のコンテキストを指定します。</param>
95      protected override void OnRender(DrawingContext drawingContext)
96      {
97          var pt = new Point(this.CurrentPoint.X + this.Offset.X, this.CurrentPoint.Y +
    this.Offset.Y);
98          var rect = new Rect(pt, this.AdornedElement.RenderSize);
99          var brush = new VisualBrush(this.AdornedElement);
100         brush.Opacity = 0.3;
101
102         drawingContext.DrawRectangle(brush, null, rect);
103     }
104
105     #endregion 描画オーバーライド
106
107     #region private フィールド
108
109     /// <summary>
110     /// 装飾層
111     /// </summary>
112     private AdornerLayer _layer;
113
114     /// <summary>
115     /// アタッチされているかどうか
116     /// </summary>
117     private bool _isAttached;
118
119     #endregion private フィールド
120 }
```

このクラスは OnRender() メソッドをオーバーライドしており、AdornedElement という UIElement クラスを VisualBrush にし、Opacity を指定して DrawRectangle() メソッドで描画しています。つまり、AdornedElement に指定されたコントロールを半透明にして表示している、まさにゴーストを表示していることになります。

描画処理の中で、ゴーストを表示させる位置を CurrentPoint プロパティと Offset プロパティから決定しています。CurrentPoint プロパティはゴーストの現在位置、Offset プロパティは CurrentPoint プロパティからのオフセットを表しています。

コンストラクタでは、この Adorner クラスを描画するための装飾層を取得しています。AdornerLayer.GetAdornerLayer() メソッドで、指定したコントロールに対する装飾層を取得できます。ここで取得した装飾層をプライベートフィールドである _layer 変数で保持し、各メソッド内で使い回しています。

Attach() メソッドで、このクラスを装飾層に登録することで実際に描画させています。また、この装飾の描画を取りやめるための Detach() メソッドも用意しています。

4.10.3 ゴーストを表示するための添付ビヘイビア

前節で定義したクラスを実際に使ってみます。ここでは添付ビヘイビアに組み込んでみます。

コード 4.28：ゴースト表示するための添付ビヘイビアの定義

AdornerBehavior.cs

```
 1  namespace Tips_Adorner.Views.Behaviors
 2  {
 3      using System.Windows;
 4      using System.Windows.Controls;
 5      using System.Windows.Documents;
 6      using System.Windows.Input;
 7      using System.Windows.Media;
 8
 9      public class AdornerBehavior
10      {
11          /// <summary>
12          /// IsEnabled 添付プロパティの定義
13          /// </summary>
14          public static readonly DependencyProperty IsEnabledProperty =
    DependencyProperty.RegisterAttached("IsEnabled", typeof(bool), typeof(AdornerBehavior),
    new PropertyMetadata(false, OnIsEnabledPropertyChanged));
15
16          /// <summary>
17          /// IsEnabled 添付プロパティを取得します。
18          /// </summary>
19          /// <param name="target">対象とする DependencyObject を指定します。</param>
20          /// <returns>取得した値を返します。</returns>
21          public static bool GetIsEnabled(DependencyObject target)
22          {
23              return (bool)target.GetValue(IsEnabledProperty);
24          }
25
26          /// <summary>
27          /// IsEnabled 添付プロパティを設定します。
28          /// </summary>
29          /// <param name="target">対象とする DependencyObject を指定します。</param>
30          /// <param name="value">設定する値を指定します。</param>
31          public static void SetIsEnabled(DependencyObject target, bool value)
32          {
33              target.SetValue(IsEnabledProperty, value);
34          }
```

```
35
36          /// <summary>
37          /// IsEnabled 添付プロパティ変更イベントハンドラ
38          /// </summary>
39          /// <param name="sender">イベント発行元</param>
40          /// <param name="e">イベント引数</param>
41          private static void OnIsEnabledPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
42          {
43              var element = sender as UIElement;
44              if (element == null)
45                  return;
46              var isEnabled = GetIsEnabled(element);
47              if (isEnabled)
48              {
49                  element.PreviewMouseLeftButtonDown += element_PreviewMouseLeftButtonDown;
50                  element.PreviewMouseMove += element_PreviewMouseMove;
51                  element.PreviewMouseLeftButtonUp += element_PreviewMouseLeftButtonUp;
52              }
53              else
54              {
55              }
56          }
57
58          /// <summary>
59          /// 装飾用コントロール
60          /// </summary>
61          private static GhostAdorner _adorner;
62
63          /// <summary>
64          /// PreviewMouseLeftButtonDown イベントハンドラ
65          /// </summary>
66          /// <param name="sender">イベント発行元</param>
67          /// <param name="e">イベント引数</param>
68          static void element_PreviewMouseLeftButtonDown(object sender,
    System.Windows.Input.MouseButtonEventArgs e)
69          {
70              var originalElement = e.OriginalSource as FrameworkElement;
71              var parent = originalElement != null ? FindAncestor<Panel>(originalElement) :
    null;
72              var adornedElement = sender as FrameworkElement;
73              if ((parent == null) || (adornedElement == null))
74                  return;
75
76              var pt = e.GetPosition(adornedElement);
77              var offset = new Point(-pt.X, -pt.Y);
78              _adorner = new GhostAdorner(parent, adornedElement, pt, offset);
79
80              adornedElement.CaptureMouse();
81          }
82
83          /// <summary>
84          /// PreviewMouseLeftButtonUp イベントハンドラ
85          /// </summary>
86          /// <param name="sender">イベント発行元</param>
87          /// <param name="e">イベント引数</param>
88          static void element_PreviewMouseLeftButtonUp(object sender, MouseButtonEventArgs
    e)
```

```
89          {
90              if (_adorner != null)
91              {
92                  _adorner.AdornedElement.ReleaseMouseCapture();
93                  _adorner.Detach();
94                  _adorner = null;
95              }
96          }
97
98          /// <summary>
99          /// PreviewMouseMove イベントハンドラ
100         /// </summary>
101         /// <param name="sender">イベント発行元</param>
102         /// <param name="e">イベント引数</param>
103         static void element_PreviewMouseMove(object sender, MouseEventArgs e)
104         {
105             if (_adorner != null)
106             {
107                 if (_adorner.AdornedElement.IsMouseCaptured && (e.LeftButton ==
    MouseButtonState.Pressed))
108                 {
109                     var pt = e.GetPosition(_adorner.AdornedElement);
110                     _adorner.CurrentPoint = pt;
111                 }
112             }
113         }
114
115         /// <summary>
116         /// 指定された型の親要素を探します。
117         /// </summary>
118         /// <typeparam name="T">親要素の型を指定します。</typeparam>
119         /// <param name="element">探索を開始する要素を指定します。</param>
120         /// <returns>親要素を返します。</returns>
121         private static T FindAncestor<T>(FrameworkElement element)
122             where T : FrameworkElement
123         {
124             do
125             {
126                 element = VisualTreeHelper.GetParent(element) as FrameworkElement;
127                 if (element is T)
128                     return element as T;
129             } while (element != null);
130             return null;
131         }
132     }
133 }
```

やり方はそれぞれですが、ここで肝心なのは、PreviewMouseLeftButtonDown イベントハンドラで GhostAdorner クラスのインスタンスを生成し、PreviewMouseMove イベントハンドラで CurrentPoint プロパティを更新し、PreviewMouseLeftButtonUp イベントハンドラで GhostAdorner クラスをデタッチしているということです。

これを使用した XAML の例を以下に示します。

コード 4.29：ゴースト表示するための添付ビヘイビアの使用例

MainView.xaml

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

```
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         xmlns:b="clr-namespace:Tips_Adorner.Views.Behaviors"
5         Title="MainView" Height="300" Width="300">
6     <Grid>
7         <Border b:AdornerBehavior.IsEnabled="True">
8             <Border.Style>
9                 <Style TargetType="Border">
10                    <Setter Property="BorderBrush" Value="Red" />
11                    <Setter Property="BorderThickness" Value="2" />
12                    <Setter Property="CornerRadius" Value="4" />
13                    <Setter Property="Padding" Value="4" />
14                    <Setter Property="Background" Value="Transparent" />
15                    <Setter Property="HorizontalAlignment" Value="Center" />
16                    <Setter Property="VerticalAlignment" Value="Center" />
17                </Style>
18            </Border.Style>
19
20            <TextBlock Text="Drag me." />
21        </Border>
22    </Grid>
23 </Window>
```

定義した添付ビヘイビアを Border コントロールに添付しているため、ゴーストは Border コントロール以下のコントロールになります。

4.11 装飾のコントロールレイアウトを XAML で指定する（Tips_AdornerCore）

Adorner クラスを使用した装飾層への描画は、例えば「4.10 ドラッグ操作でゴーストを表示する(Tips_Adorner)」で示しているように C# コードから OnRender() メソッドをオーバーライドして描画する方法があります。ここでは少し工夫をして、C# コードからではなく、XAML 上で定義したレイアウトを描画する方法を紹介します。

装飾に使用するコントロールを XAML 上で定義できるようになることで、ボタンなどユーザーの操作を要求するようなコントロールを配置し、そのコマンドをデータバインディングによって ViewModel に通知するといったことが簡単にできるようになります。

4.11.1 Adorner クラスからの派生クラスを用意する

ここでも Adorner クラスによる描画をおこなうため、Adorner クラスを基本とした派生クラスを AdorenerCore クラスとして次のように定義します。ここでは任意のコントロールを描画させることを目的としているため、OnRender() メソッドはオーバーライドしません。

コード 4.30：任意のコントロールを装飾として表示するための AdornerCore クラスの定義

AdornerBehavior.cs

```
 1  namespace Tips_AdornerCore.Views.Behaviors
 2  {
 3      using System.Windows;
 4      using System.Windows.Documents;
 5      using System.Windows.Media;
 6
 7      public class AdornerBehavior
 8      {
 9          /// <summary>
10          /// 装飾のためのクラスを表します。
11          /// </summary>
12          internal class AdornerCore : Adorner
13          {
14              /// <summary>
15              /// 新しいインスタンスを生成します。
16              /// </summary>
17              /// <param name="adornedElement">装飾対象を指定します。</param>
18              /// <param name="adorner">装飾を指定します。</param>
19              public AdornerCore(UIElement adornedElement, FrameworkElement adorner)
20                  : base(adornedElement)
21              {
22                  this._layer = AdornerLayer.GetAdornerLayer(adornedElement);
23                  this._adorner = adorner;
24                  AddVisualChild(this._adorner);
25                  Attach();
26              }
27
28              /// <summary>
29              /// Adorner 依存関係プロパティの定義を表します。
30              /// </summary>
31              public static readonly DependencyProperty AdornerProperty =
    DependencyProperty.Register("Adorner", typeof(AdornerCore), typeof(AdornerCore), new
    PropertyMetadata(null));
32
33              /// <summary>
34              /// 装飾を表示します。
35              /// </summary>
36              public void Attach()
37              {
```

```
38                 if (this._layer != null)
39                 {
40                     this._layer.Add(this);
41                 }
42             }
43 
44             /// <summary>
45             /// 装飾を非表示にします。
46             /// </summary>
47             public void Detach()
48             {
49                 if (this._layer != null)
50                 {
51                     this._layer.Remove(this);
52                 }
53             }
54 
55             /// <summary>
56             /// 装飾対象の描画サイズを取得します。
57             /// </summary>
58             private Size ActualAdoredElementSize
59             {
60                 get { return new Size((this.AdornedElement as
61 FrameworkElement).ActualWidth, (this.AdornedElement as
   FrameworkElement).ActualHeight); }
62             }
63 
64             /// <summary>
65             /// コントロールのサイズ計測処理のオーバーライド
66             /// </summary>
67             /// <param name="constraint">サイズに対する制約</param>
68             /// <returns>計測結果</returns>
69             protected override Size MeasureOverride(Size constraint)
70             {
71                 return this.ActualAdoredElementSize;
72             }
73 
74             /// <summary>
75             /// 子要素の配置処理のオーバーライド
76             /// </summary>
77             /// <param name="finalSize">親から与えられる領域のサイズ</param>
78             /// <returns>配置に要したサイズ</returns>
79             protected override Size ArrangeOverride(Size finalSize)
80             {
81                 var size = this.ActualAdoredElementSize;
82                 if (this._adorner != null)
83                 {
84                     this._adorner.Arrange(new Rect(size));
85                 }
86                 return size;
87             }
88 
89             /// <summary>
90             /// ビジュアルツリーにおける子要素の数を取得します。
91             /// </summary>
92             protected override int VisualChildrenCount
93             {
94                 get
```

```
94                 {
95                     return 1;
96                 }
97             }
98 
99             /// <summary>
100            /// ビジュアルツリーにおける子要素を取得します。
101            /// </summary>
102            /// <param name="index"></param>
103            /// <returns></returns>
104            protected override Visual GetVisualChild(int index)
105            {
106                return this._adorner;
107            }
108 
109            /// <summary>
110            /// 装飾層を保持します。
111            /// </summary>
112            private AdornerLayer _layer;
113 
114            /// <summary>
115            /// 装飾を保持します。
116            /// </summary>
117            private FrameworkElement _adorner;
118        }
119    }
120 }
```

AdornerCore クラスはそのオブジェクトがインスタンス化されたとき、コンストラクタ内で Attach() メソッドによって装飾が描画されます。この装飾はコンストラクタで指定された FrameworkElement クラスのコントロールであり、UIElement クラスから見た装飾層に描画されます。

4.11.2 添付プロパティの定義

描画する装飾コントロールは XAML 上で定義することを想定しているため、XAML 上で定義した DataTemplate クラスをデータバインディングできるように添付プロパティを用意します。

コード 4.31：AdornerTemplate 添付プロパティなどの定義

AdornerBehavior.cs

```
1  namespace Tips_AdornerCore.Views.Behaviors
2  {
3      using System.Windows;
4      using System.Windows.Documents;
5      using System.Windows.Media;
6 
7      public class AdornerBehavior
8      {
9          #region AdornerTemplate 添付プロパティ
10         /// <summary>
11         /// AdornerTemplate 添付プロパティを表します。
12         /// </summary>
13         public static readonly DependencyProperty AdornerTemplateProperty =
       DependencyProperty.RegisterAttached("AdornerTemplate", typeof(DataTemplate),
       typeof(AdornerBehavior), new UIPropertyMetadata(null, OnAdornerTemplateProperty));
14 
15         /// <summary>
16         /// AdornerTemplate 添付プロパティを取得します。
17         /// </summary>
```

```
18          /// <param name="target">対象とする DependencyObject を指定します。</param>
19          /// <returns>取得した値を返します。</returns>
20          public static DataTemplate GetAdornerTemplate(DependencyObject target)
21          {
22              return (DataTemplate)target.GetValue(AdornerTemplateProperty);
23          }
24
25          /// <summary>
27          /// AdornerTemplate 添付プロパティを設定します。
28          /// </summary>
29          /// <param name="target">対象とする DependencyObject を指定します。</param>
30          /// <param name="value">設定する値を指定します。</param>
31          public static void SetAdornerTemplate(DependencyObject target, DataTemplate value)
32          {
33              target.SetValue(AdornerTemplateProperty, value);
34          }
35          #endregion AdornerTemplate 添付プロパティ
36
37          #region IsEnabled 添付プロパティ
38          /// <summary>
39          /// IsEnabled 添付プロパティを表します。
40          /// </summary>
41          public static readonly DependencyProperty IsEnabledProperty =
    DependencyProperty.RegisterAttached("IsEnabled", typeof(bool), typeof(AdornerBehavior),
    new UIPropertyMetadata(false, OnIsEnabledPropertyChanged));
42
43          /// <summary>
44          /// IsEnabled 添付プロパティを取得します。
45          /// </summary>
46          /// <param name="target">対象とする DependencyObject を指定します。</param>
47          /// <returns>取得した値を返します。</returns>
48          public static bool GetIsEnabled(DependencyObject target)
49          {
50              return (bool)target.GetValue(IsEnabledProperty);
51          }
52
53          /// <summary>
54          /// IsEnabled 添付プロパティを設定します。
55          /// </summary>
56          /// <param name="target">対象とする DependencyObject を指定します。</param>
57          /// <param name="value">設定する値を指定します。</param>
58          public static void SetIsEnabled(DependencyObject target, bool value)
59          {
60              target.SetValue(IsEnabledProperty, value);
61          }
62          #endregion IsEnabled 添付プロパティ
63
64          #region AdornerCore クラス
            ・・・
175         #endregion AdornerCore クラス
176     }
177 }
```

後はそれぞれの添付プロパティの変更イベントハンドラ内で装飾をおこなったりおこなわないようにしたりするだけです。

コード 4.32：添付プロパティ変更イベントハンドラの実装

AdornerBehavior.cs

```
 1  namespace Tips_AdornerCore.Views.Behaviors
 2  {
 3      using System.Windows;
 4      using System.Windows.Documents;
 5      using System.Windows.Media;
 6
 7      public class AdornerBehavior
 8      {
 9          #region AdornerTemplate 添付プロパティ
            ・・・
34          #endregion AdornerTemplate 添付プロパティ
35
36          #region IsEnabled 添付プロパティ
            ・・・
61          #endregion IsEnabled 添付プロパティ
62
63          /// <summary>
64          /// AdornerTemplate 添付プロパティ変更イベントハンドラ
65          /// </summary>
66          /// <param name="sender">イベント発行元</param>
67          /// <param name="e">イベント引数</param>
68          private static void OnAdornerTemplateProperty(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
69          {
70              var element = sender as UIElement;
71              var template = GetAdornerTemplate(element);
72              var isEnabled = GetIsEnabled(element);
73
74              if (isEnabled)
75              {
76                  var adorner = element.GetValue(AdornerCore.AdornerProperty) as
    AdornerCore;
77                  if (adorner != null)
78                  {
79                      adorner.Detach();
80                  }
81                  var frameworkElement = template != null ? template.LoadContent() as
    FrameworkElement : null;
82                  adorner = frameworkElement != null ? new AdornerCore(element,
    frameworkElement) : null;
83                  element.SetValue(AdornerCore.AdornerProperty, adorner);
84              }
85          }
86
87          /// <summary>
88          /// IsEnabled 添付プロパティ変更イベントハンドラ
89          /// </summary>
90          /// <param name="sender">イベント発行元</param>
91          /// <param name="e">イベント引数</param>
92          private static void OnIsEnabledPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
93          {
94              var element = sender as UIElement;
95              var template = GetAdornerTemplate(element);
96              var isEnabled = GetIsEnabled(element);
97
98              var adorner = element.GetValue(AdornerCore.AdornerProperty) as AdornerCore;
99              if (isEnabled)
```

```
100             {
101                 if (adorner == null)
102                 {
103                     if ((element != null) && (template != null))
104                     {
105                         var frameworkElement = template.LoadContent() as FrameworkElement;
106                         adorner = new AdornerCore(element, frameworkElement);
107                         element.SetValue(AdornerCore.AdornerProperty, adorner);
108                     }
109                 }
110                 else
111                 {
112                     adorner.Attach();
113                 }
114             }
115             else
116             {
117                 if (adorner != null)
118                 {
119                     adorner.Detach();
120                 }
121             }
122         }
123 
124         #region AdornerCore クラス
            ・・・
235         #endregion AdornerCore クラス
236     }
237 }
```

4.11.3 使用例

前節で定義した AdornerBehavior クラスを使った例を紹介します。

UI のレイアウトは 2 つの ToggleButton コントロールを並べるだけのシンプルなもので、リソースとして装飾用のコントロールを 2 つ定義しています。

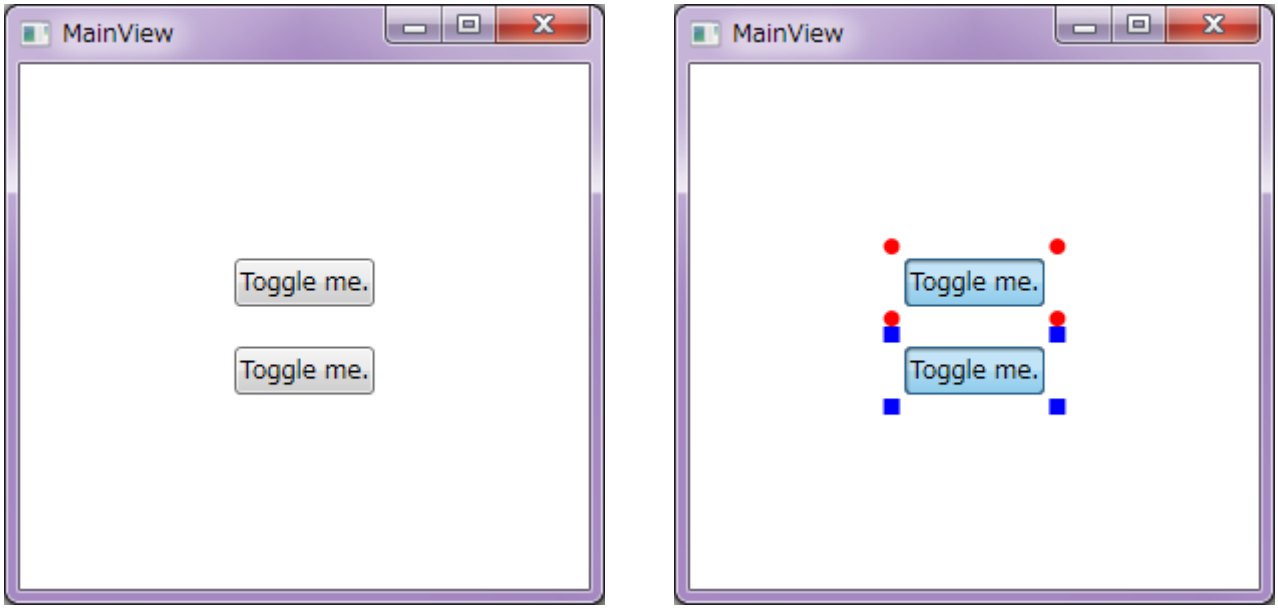
コード 4.33 : AdornerBehavior クラスのサンプル用 UI

MainView.xaml

```
1  <Window x:Class="Tips_AdornerCore.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:b="clr-namespace:Tips_AdornerCore.Views.Behaviors"
5          Title="MainView" Height="300" Width="300">
6      <Window.Resources>
7          <DataTemplate x:Key="template1">
8              <Grid Margin="-10">
9                  <Ellipse Width="8" Height="8" Fill="Red" HorizontalAlignment="Left"
   VerticalAlignment="Top" />
10                 <Ellipse Width="8" Height="8" Fill="Red" HorizontalAlignment="Right"
   VerticalAlignment="Top" />
11                 <Ellipse Width="8" Height="8" Fill="Red" HorizontalAlignment="Right"
   VerticalAlignment="Bottom" />
12                 <Ellipse Width="8" Height="8" Fill="Red" HorizontalAlignment="Left"
   VerticalAlignment="Bottom" />
13             </Grid>
14         </DataTemplate>
15 
```

```
16          <DataTemplate x:Key="template2">
17              <Grid Margin="-10">
18                  <Rectangle Width="8" Height="8" Fill="Blue" HorizontalAlignment="Left"
    VerticalAlignment="Top" />
19                  <Rectangle Width="8" Height="8" Fill="Blue" HorizontalAlignment="Right"
    VerticalAlignment="Top" />
20                  <Rectangle Width="8" Height="8" Fill="Blue" HorizontalAlignment="Right"
    VerticalAlignment="Bottom" />
21                  <Rectangle Width="8" Height="8" Fill="Blue" HorizontalAlignment="Left"
    VerticalAlignment="Bottom" />
22              </Grid>
23          </DataTemplate>
24      </Window.Resources>
25
26      <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
27          <ToggleButton x:Name="button1"
28                        Content="Toggle me."
29                        Margin="10"
30                        b:AdornerBehavior.AdornerTemplate="{StaticResource template1}"
31                        b:AdornerBehavior.IsEnabled="{Binding IsChecked,
    ElementName=button1}" />
32          <ToggleButton x:Name="button2"
33                        Content="Toggle me."
34                        Margin="10"
35                        b:AdornerBehavior.AdornerTemplate="{StaticResource template2}"
36                        b:AdornerBehavior.IsEnabled="{Binding IsChecked,
    ElementName=button2}" />
37      </StackPanel>
38  </Window>
```

AdornerBehavior.IsEnabled 添付プロパティには、それぞれ自身の IsChecked プロパティをデータバインディングさせています。つまり、ToggleButton をクリックする度に装飾の表示/非表示を切り替えられます。



(a) 起動直後　　(b) ToggleButton を有効にした場合

図 4.17：ToggleButton をクリックするとそれぞれの装飾が表示される

4.11.4 DataTemplateSelector クラスによる装飾の切り替え

定義した AdornerBehavior クラスを使用することによって、XAML 上で定義した DataTemplate クラスを装飾として描画できるようになりました。せっかくなので、このビヘイビアに DataTemplateSelector クラスを追加して、ある条件にしたがって表示する装飾を動的に変更できるようにしてみましょう。

変更するところは、AdornerBehavior クラスに DataTemplateSelector クラスの添付プロパティを追加することと、装飾するコントロールとして DataTemplate クラスを取得する際に、DataTemplte クラスの添付プロパティ あるいは DataTemplateSelector クラスの添付プロパティのどちらから取得するかを決めておくだけです。

以下、変更後のソース全文を掲載します。
上記のサンプルに加え、AdornerCore クラスのコンストラクタ内で装飾用のコントロールをビジュアルツリーに追加しています。こうすることによって描画されるコントロールをインタラクティブに操作することができるようになります。逆に言うと、こうしなければ、例えば Button コントロールを装飾として配置しても、そのボタンをクリック操作することができません。
また、AdornerCore クラスの DataContext プロパティを指定するための DataContextElement 添付プロパティを追加しています。DataContextElement 添付プロパティに指定されたコントロールと同じ DataContext を持つようになります。例えば ItemsControl コントロールで並べたアイテムに対する装飾をおこなう場合、親要素である ItemsControl コントロールに対する DataContext 上にあるコマンドを利用したい場合などに使うと便利です。

コード 4.34：DataTemplateSelector クラスに対応した AdornerBehavior クラス

AdornerBehavior.cs

```
 1  namespace Tips_AdornerCore.Views.Behaviors
 2  {
 3      using System.Windows;
 4      using System.Windows.Controls;
 5      using System.Windows.Documents;
 6      using System.Windows.Media;
 7
 8      /// <summary>
 9      /// 装飾に関するビヘイビアを表します。
10      /// </summary>
11      public class AdornerBehavior
12      {
13          #region AdornerTemplate 添付プロパティ
14          /// <summary>
15          /// AdornerTemplate 添付プロパティを表します。
16          /// </summary>
17          public static readonly DependencyProperty AdornerTemplateProperty =
    DependencyProperty.RegisterAttached("AdornerTemplate", typeof(DataTemplate),
    typeof(AdornerBehavior), new UIPropertyMetadata(null, OnAdornerTemplateProperty));
18
19          /// <summary>
20          /// AdornerTemplate 添付プロパティを取得します。
21          /// </summary>
22          /// <param name="target">対象とする DependencyObject を指定します。</param>
23          /// <returns>取得した値を返します。</returns>
24          public static DataTemplate GetAdornerTemplate(DependencyObject target)
25          {
26              return (DataTemplate)target.GetValue(AdornerTemplateProperty);
27          }
28
29          /// <summary>
30          /// AdornerTemplate 添付プロパティを設定します。
31          /// </summary>
32          /// <param name="target">対象とする DependencyObject を指定します。</param>
```

```
33          /// <param name="value">設定する値を指定します。</param>
34          public static void SetAdornerTemplate(DependencyObject target, DataTemplate value)
35          {
36              target.SetValue(AdornerTemplateProperty, value);
37          }
38          #endregion AdornerTemplate 添付プロパティ
39
40          #region AdornerTemplateSelector 添付プロパティ
41          /// <summary>
42          /// AdornerTemplateSelector 添付プロパティを表します。
43          /// </summary>
44          public static readonly DependencyProperty AdornerTemplateSelectorProperty =
    DependencyProperty.RegisterAttached("AdornerTemplateSelector",
    typeof(DataTemplateSelector), typeof(AdornerBehavior), new UIPropertyMetadata(null,
    OnAdornerTemplateSelectorProperty));
45
46          /// <summary>
47          /// AdornerTemplateSelector 添付プロパティを取得します。
48          /// </summary>
49          /// <param name="target">対象とする DependencyObject を指定します。</param>
50          /// <returns>取得した値を返します。</returns>
51          public static DataTemplateSelector GetAdornerTemplateSelector(DependencyObject
    target)
52          {
53              return
    (DataTemplateSelector)target.GetValue(AdornerTemplateSelectorProperty);
54          }
55
56          /// <summary>
57          /// AdornerTemplateSelector 添付プロパティを設定します。
58          /// </summary>
59          /// <param name="target">対象とする DependencyObject を指定します。</param>
60          /// <param name="value">設定する値を指定します。</param>
61          public static void SetAdornerTemplateSelector(DependencyObject target,
    DataTemplateSelector value)
62          {
63              target.SetValue(AdornerTemplateSelectorProperty, value);
64          }
65          #endregion AdornerTemplateSelector 添付プロパティ
66
67          #region IsEnabled 添付プロパティ
68          /// <summary>
69          /// IsEnabled 添付プロパティを表します。
70          /// </summary>
71          public static readonly DependencyProperty IsEnabledProperty =
    DependencyProperty.RegisterAttached("IsEnabled", typeof(bool), typeof(AdornerBehavior),
    new UIPropertyMetadata(false, OnIsEnabledPropertyChanged));
72
73          /// <summary>
74          /// IsEnabled 添付プロパティを取得します。
75          /// </summary>
76          /// <param name="target">対象とする DependencyObject を指定します。</param>
77          /// <returns>取得した値を返します。</returns>
78          public static bool GetIsEnabled(DependencyObject target)
79          {
80              return (bool)target.GetValue(IsEnabledProperty);
81          }
82
```

```
83          /// <summary>
84          /// IsEnabled 添付プロパティを設定します。
85          /// </summary>
86          /// <param name="target">対象とする DependencyObject を指定します。</param>
87          /// <param name="value">設定する値を指定します。</param>
88          public static void SetIsEnabled(DependencyObject target, bool value)
89          {
90              target.SetValue(IsEnabledProperty, value);
91          }
92          #endregion IsEnabled 添付プロパティ
93
94          #region DataContextElement 添付プロパティ
95          /// <summary>
96          /// DataContextElement 添付プロパティを表します。
97          /// </summary>
98          public static readonly DependencyProperty DataContextElementProperty =
    DependencyProperty.RegisterAttached("DataContextElement", typeof(FrameworkElement),
    typeof(AdornerBehavior), new UIPropertyMetadata(null,
    OnDataContextElementPropertyChanged));
99
100         /// <summary>
101         /// DataContextElement 添付プロパティを取得します。
102         /// </summary>
103         /// <param name="target">対象とする DependencyObject を指定します。</param>
104         /// <returns>取得した値を返します。</returns>
105         public static FrameworkElement GetDataContextElement(DependencyObject target)
106         {
107             return (FrameworkElement)target.GetValue(DataContextElementProperty);
108         }
109
110         /// <summary>
111         /// DataContextElement 添付プロパティを設定します。
112         /// </summary>
113         /// <param name="target">対象とする DependencyObject を指定します。</param>
114         /// <param name="value">設定する値を指定します。</param>
115         public static void SetDataContextElement(DependencyObject target,
    FrameworkElement value)
116         {
117             target.SetValue(DataContextElementProperty, value);
118         }
119         #endregion DataContextElement 添付プロパティ
120
121         #region 添付プロパティ変更イベントハンドラ
122
123         /// <summary>
124         /// AdornerTemplate 添付プロパティ変更イベントハンドラ
125         /// </summary>
126         /// <param name="sender">イベント発行元</param>
127         /// <param name="e">イベント引数</param>
128         private static void OnAdornerTemplateProperty(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
129         {
130             OnChange(sender);
131         }
132
133         /// <summary>
134         /// AdornerTemplateSelector 添付プロパティ変更イベントハンドラ
135         /// </summary>
```

```
136         /// <param name="sender">イベント発行元</param>
137         /// <param name="e">イベント引数</param>
138         private static void OnAdornerTemplateSelectorProperty(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
139         {
140             OnChange(sender);
141         }
142 
143         /// <summary>
144         /// IsEnabled 添付プロパティ変更イベントハンドラ
145         /// </summary>
146         /// <param name="sender">イベント発行元</param>
147         /// <param name="e">イベント引数</param>
148         private static void OnIsEnabledPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
149         {
150             OnChange(sender);
151         }
152 
153         /// <summary>
154         /// DataContextElement 添付プロパティ変更イベントハンドラ
155         /// </summary>
156         /// <param name="sender">イベント発行元</param>
157         /// <param name="e">イベント引数</param>
158         private static void OnDataContextElementPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
159         {
160             SetAdornerDataContext(sender);
161         }
162 
163         #endregion 添付プロパティ変更イベントハンドラ
164 
165         #region ヘルパ
166 
167         /// <summary>
168         /// テンプレート変更時のヘルパ
169         /// </summary>
170         /// <param name="target">対象とする DependencyObject を指定します。</param>
171         private static void OnChange(DependencyObject target)
172         {
173             var element = target as FrameworkElement;
174             var template = GetDataTemplate(element);
175             var isEnabled = GetIsEnabled(element);
176 
177             var adorner = element.GetValue(AdornerCore.AdornerProperty) as AdornerCore;
178             if (isEnabled)
179             {
180                 if (adorner == null)
181                 {
182                     if ((element != null) && (template != null))
183                     {
184                         var frameworkElement = template.LoadContent() as FrameworkElement;
185                         adorner = new AdornerCore(element, frameworkElement);
186                         element.SetValue(AdornerCore.AdornerProperty, adorner);
187                         SetAdornerDataContext(target);
188                     }
189                 }
190                 else
```

```
191                 {
192                     adorner.Attach();
193                 }
194             }
195             else
196             {
197                 if (adorner != null)
198                 {
199                     adorner.Detach();
200                 }
201             }
202         }
203 
204         /// <summary>
205         /// 装飾に対する DataContext を設定します。
206         /// </summary>
207         /// <param name="target">添付ビヘイビアの対象となっている DependencyObject を指定します。</param>
208         private static void SetAdornerDataContext(DependencyObject target)
209         {
210             var adorner = (target as DependencyObject).GetValue(AdornerCore.AdornerProperty) as AdornerCore;
211             if (adorner != null)
212             {
213                 var dataContextElement = GetDataContextElement(target);
214                 if (dataContextElement != null)
215                 {
216                     adorner.DataContext = dataContextElement.DataContext;
217                 }
218                 else
219                 {
220                     adorner.DataContext = (target as FrameworkElement).DataContext;
221                 }
222             }
223         }
224 
225         /// <summary>
226         /// AdornerTemplateSelector を優先的に DataTemplate を取得します。
227         /// </summary>
228         /// <param name="target">対象とする DependencyObject を指定します。</param>
229         /// <returns>装飾に対する DataTemplate を返します。</returns>
230         private static DataTemplate GetDataTemplate(DependencyObject target)
231         {
232             var selector = GetAdornerTemplateSelector(target);
233             var templateFromSelector = selector != null ? selector.SelectTemplate((target as FrameworkElement).DataContext, target) : null;
234             return templateFromSelector != null ? templateFromSelector : GetAdornerTemplate(target);
235         }
236 
237         #endregion ヘルパ
238 
239         /// <summary>
240         /// 装飾のためのクラスを表します。
241         /// </summary>
242         internal class AdornerCore : Adorner
243         {
244             /// <summary>
```

```
245             /// 新しいインスタンスを生成します。
246             /// </summary>
247             /// <param name="adornedElement">装飾対象を指定します。</param>
248             /// <param name="adorner">装飾を指定します。</param>
249             public AdornerCore(UIElement adornedElement, FrameworkElement adorner)
250                 : base(adornedElement)
251             {
252                 this._layer = AdornerLayer.GetAdornerLayer(adornedElement);
253                 this._adorner = adorner;
254                 AddVisualChild(this._adorner);
255                 Attach();
256             }
257 
258             /// <summary>
259             /// Adorner 依存関係プロパティの定義を表します。
260             /// </summary>
261             public static readonly DependencyProperty AdornerProperty =
    DependencyProperty.Register("Adorner", typeof(AdornerCore), typeof(AdornerCore), new
    PropertyMetadata(null));
262 
263             /// <summary>
264             /// 装飾を表示します。
265             /// </summary>
266             public void Attach()
267             {
268                 if (this._layer != null)
269                 {
270                     this._layer.Add(this);
271                 }
272             }
273 
274             /// <summary>
275             /// 装飾を非表示にします。
276             /// </summary>
277             public void Detach()
278             {
279                 if (this._layer != null)
280                 {
281                     this._layer.Remove(this);
282                 }
283             }
284 
285             /// <summary>
286             /// 装飾対象の描画サイズを取得します。
287             /// </summary>
288             private Size ActualAdoredElementSize
289             {
290                 get { return new Size((this.AdornedElement as
    FrameworkElement).ActualWidth, (this.AdornedElement as
    FrameworkElement).ActualHeight); }
291             }
292 
293             /// <summary>
294             /// コントロールのサイズ計測処理のオーバーライド
295             /// </summary>
296             /// <param name="constraint">サイズに対する制約</param>
297             /// <returns>計測結果</returns>
298             protected override Size MeasureOverride(Size constraint)
```

```
299             {
300                 return this.ActualAdoredElementSize;
301             }
302 
303             /// <summary>
304             /// 子要素の配置処理のオーバーライド
305             /// </summary>
306             /// <param name="finalSize">親から与えられる領域のサイズ</param>
307             /// <returns>配置に要したサイズ</returns>
308             protected override Size ArrangeOverride(Size finalSize)
309             {
310                 var size = this.ActualAdoredElementSize;
311                 if (this._adorner != null)
312                 {
313                     this._adorner.Arrange(new Rect(size));
314                 }
315                 return size;
316             }
317 
318             /// <summary>
319             /// ビジュアルツリーにおける子要素の数を取得します。
320             /// </summary>
321             protected override int VisualChildrenCount
322             {
323                 get
324                 {
325                     return 1;
326                 }
327             }
328 
329             /// <summary>
330             /// ビジュアルツリーにおける子要素を取得します。
331             /// </summary>
332             /// <param name="index"></param>
333             /// <returns></returns>
334             protected override Visual GetVisualChild(int index)
335             {
336                 return this._adorner;
337             }
338 
339             /// <summary>
340             /// 装飾層を保持します。
341             /// </summary>
342             private AdornerLayer _layer;
343 
344             /// <summary>
345             /// 装飾を保持します。
346             /// </summary>
347             private FrameworkElement _adorner;
348         }
349     }
350 }
```

4.12 TabControl コントロールのあれこれ (Tips_TabControl)

TabControl コントロールは複数のコンテンツを切り替えることができるコントロールで、例えば次のように使用します。

コード 4.35：TabControl の使用例

MainView.xaml

```
1  <Window x:Class="Tips_TabControl.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      <TabControl>
6          <TabItem Header="Item1">
7              <TextBlock Text="Item1 のコンテンツ" />
8          </TabItem>
9          <TabItem Header="Item2">
10             <TextBlock Text="Item2 のコンテンツ" />
11         </TabItem>
12     </TabControl>
13 </Window>
```

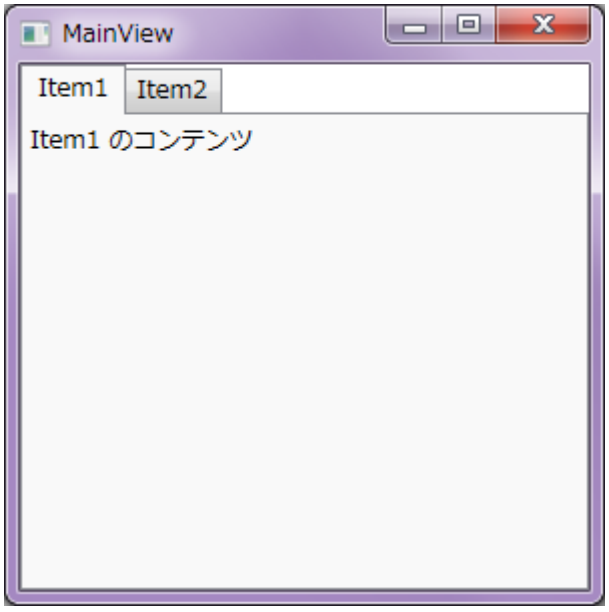


図 4.18：TabControl でコンテンツを切り替えられる

TabControl 内に TabItem を並べることで簡単にコンテンツを切り替えるコントロールを実現できます。
ところで、TabItem 内のコンテンツが単純な場合はこのような記述でも十分ですが、複雑になってくると、ViewModel をわけて管理したり、そもそもコンテンツとしてユーザーコントロールを指定したりすることもあります。
方法は色々ありますが、ここでは ItemsSource プロパティに ViewModel のコレクションを指定することを考えます。

4.12.1 ViewModel のコレクションを ItemsSource プロパティに指定する

ここでは、TabControl コントロールで切り替えるコンテンツに対する ViewModel を次のように定義します。

コード 4.36：コンテンツに対する ViewModel の定義

ContentViewModel.cs

```
1  namespace Tips_TabControl.ViewModels
```

```
 2 {
 3     public class ContentViewModel : NotificationObject
 4     {
 5         /// <summary>
 6         /// 新しいインスタンスを生成します。
 7         /// </summary>
 8         /// <param name="title">タイトルを指定します。</param>
 9         public ContentViewModel(string title)
10         {
11             this.Title = title;
12         }
13 
14         private string _title;
15         /// <summary>
16         /// タイトルを取得します。
17         /// </summary>
18         public string Title
19         {
20             get { return this._title; }
21             private set { SetProperty(ref this._title, value); }
22         }
23     }
24 }
```

Title プロパティを持ち、インスタンス生成時に必ず指定するようなクラスになっています。
続いて、このクラスをコレクションとして持つ MainViewModel を定義します。

コード 4.37：ContentViewModel をコレクションとして保持する MainViewModel

MainViewModel.cs

```
 1 namespace Tips_TabControl.ViewModels
 2 {
 3     using System.Collections.ObjectModel;
 4 
 5     public class MainViewModel : NotificationObject
 6     {
 7         /// <summary>
 8         /// 新しいインスタンスを生成します。
 9         /// </summary>
10         public MainViewModel()
11         {
12             this.Contents.Add(new ContentViewModel("Item1"));
13             this.Contents.Add(new ContentViewModel("Item2"));
14             this.Contents.Add(new ContentViewModel("Item3"));
15         }
16 
17         private ObservableCollection<ContentViewModel> _contents = new
    ObservableCollection<ContentViewModel>();
18         /// <summary>
19         /// コンテンツのコレクションを取得します。
20         /// </summary>
21         public ObservableCollection<ContentViewModel> Contents
22         {
23             get { return this._contents; }
24         }
25     }
26 }
```

これで準備が整ったので、View の XAML コードを記述してみます。

コード 4.38： ViewModel のコレクションをデータバインディングする

MainView.xaml

```
1  <Window x:Class="Tips_TabControl.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      <TabControl ItemsSource="{Binding Contents}" />
6  </Window>
```

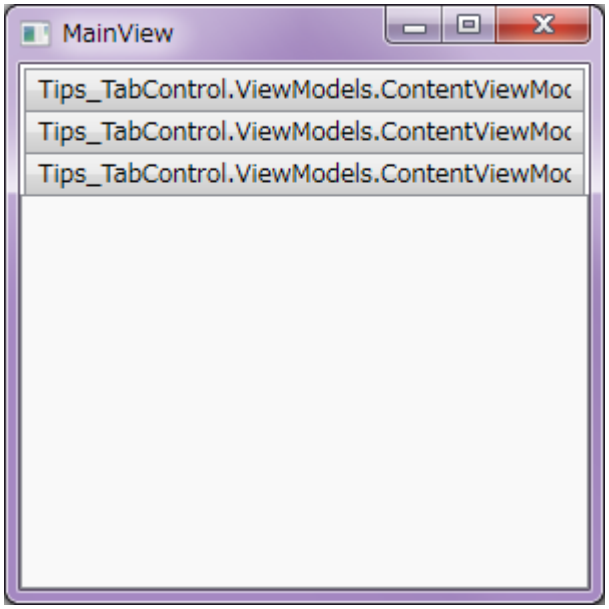


図 4.19：表示方法をカスタマイズする必要がある

お察しの通り、ただ ItemsSource を指定するだけでは足りません。TabControl にはヘッダ部を変更する ItemTemplate プロパティと、コンテンツ部を変更する ContentTemplate プロパティがあるので、これらを適切に指定します。

コード 4.39：ItemTemplate プロパティと ContentTemplate プロパティを指定する

MainView.xaml

```
1  <Window x:Class="Tips_TabControl.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      <TabControl ItemsSource="{Binding Contents}" SelectedIndex="0">
6          <TabControl.ItemTemplate>
7              <DataTemplate>
8                  <TextBlock Text="{Binding Title}" />
9              </DataTemplate>
10         </TabControl.ItemTemplate>
11         <TabControl.ContentTemplate>
12             <DataTemplate>
13                 <TextBlock Text="{Binding Title, StringFormat={}これは{0}のコンテンツです。}" />
14             </DataTemplate>
15         </TabControl.ContentTemplate>
16     </TabControl>
17 </Window>
```

図 4.20：表示方法がカスタマイズされている

TabControl の ItemsSource プロパティに ContentViewModel のコレクションを指定しているため、個々のアイテムの DataContext は ContentViewModel になります。したがって、Title プロパティとデータバインディングすることで、ContentViewModel の Title プロパティが参照されるようになります。

4.12.2 タブを追加/削除する

切り替えられるコンテンツの数は固定であることもありますが、変化することもあります。ここでは、ContentViewModel を用いたタブの追加/削除方法について紹介します。

まず、タブが追加されるということは、TabControl コントロールの ItemsSource プロパティに指定されているコレクションの数が増えるということです。つまり、コレクションを保持している MainViewModel で、新たな ContentViewModel をコレクションに追加することで実現できます。

そういうわけで、MainViewModel に AddContentCommand を追加します。

コード 4.40：AddContentCommand でコレクションに追加する

MainViewModel.cs

```
1  namespace Tips_TabControl.ViewModels
2  {
3      using System.Collections.ObjectModel;
4
5      public class MainViewModel : NotificationObject
6      {
7          private ObservableCollection<ContentViewModel> _contents = new
   ObservableCollection<ContentViewModel>();
8          /// <summary>
9          /// コンテンツのコレクションを取得します。
10         /// </summary>
11         public ObservableCollection<ContentViewModel> Contents
12         {
13             get { return this._contents; }
14         }
15
16         private DelegateCommand _addContentCommand;
17         /// <summary>
18         /// コンテンツ追加コマンドを取得します。
```

```
19          /// </summary>
20          public DelegateCommand AddContentCommand
21          {
22              get
23              {
24                  return this._addContentCommand ?? (this._addContentCommand = new
    DelegateCommand(
25                  _ =>
26                  {
27                      var content = new ContentViewModel("Item" + _count);
28                      _count++;
29                      this.Contents.Add(content);
30                  },
31                  _ => this.Contents.Count < 4));
32              }
33          }
34
35          private int _count;
36      }
37  }
```

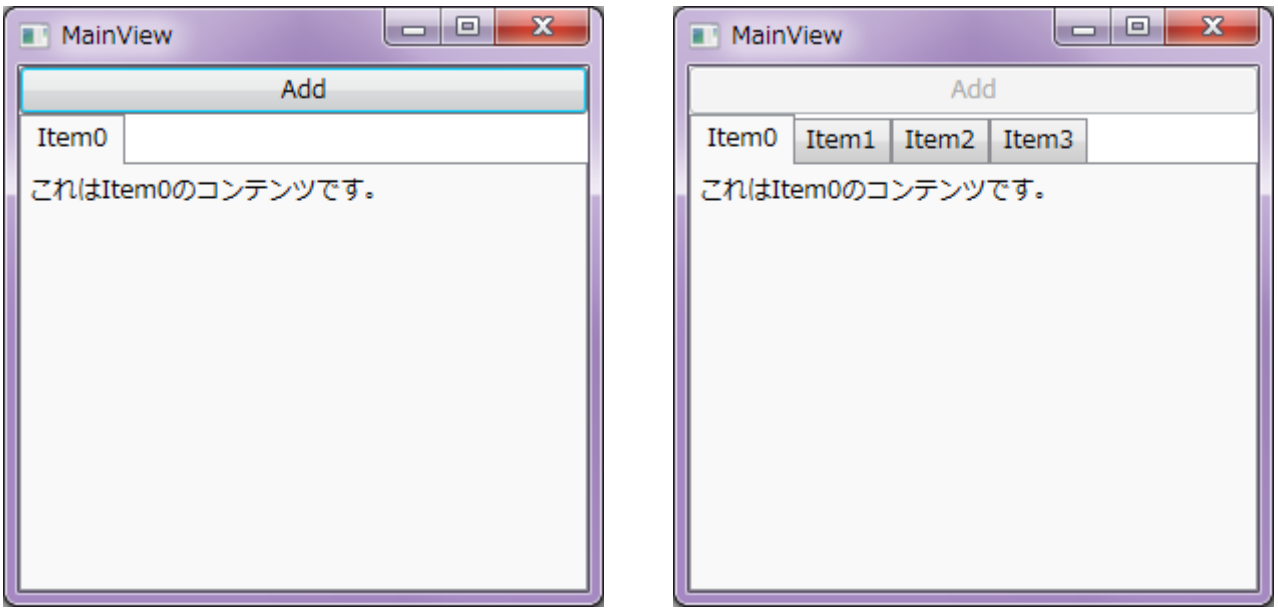
追加されるコンテンツの違いがわかるように、Title プロパティにカウンタの数値を含めます。また、実行条件として、コレクションの数を 4 個までとし、それ以上は追加できなくなるようにしています。

このコマンドを使用してコンテンツが追加できるように View のほうも変更します。

コード 4.41 : AddContentCommand をデータバインディングする

MainView.xaml

```
1  <Window x:Class="Tips_TabControl.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      <DockPanel>
6          <Button DockPanel.Dock="Top" Content="Add" Command="{Binding AddContentCommand}"
   />
7
8          <TabControl ItemsSource="{Binding Contents}" SelectedIndex="0">
9              <TabControl.ItemTemplate>
10                 <DataTemplate>
11                     <TextBlock Text="{Binding Title}" />
12                 </DataTemplate>
13             </TabControl.ItemTemplate>
14             <TabControl.ContentTemplate>
15                 <DataTemplate>
16                     <TextBlock Text="{Binding Title, StringFormat={}これは{0}のコンテンツで
   す。}" />
17                 </DataTemplate>
18             </TabControl.ContentTemplate>
19         </TabControl>
20     </DockPanel>
21 </Window>
```



(a) Add ボタンで追加できる　　(b) 追加できるのは 5 個まで

図 4.21：タブを追加するサンプル

次に、タブを削除する方法を紹介します。タブを削除するということは、MainViewModel が保持しているコレクションの数を減らすということになります。ここでは、タブのヘッダ部に表示される閉じるボタンを押すことで削除するという実装方法を説明します。

タブのヘッダ部に表示するボタンは、ContentViewModel が DataContext になります。したがって、ContentViewModel の中になんらかのコマンドを実装する必要があります。そして、そのコマンドを起点として、MainViewModel のコレクションから該当するアイテムを削除しなくてはなりません。ここでは、ContentViewModel にあるイベントを定義して、それを MainViewModel が購読することでこの一連の動作を実現します。

コード 4.42：閉じるためのコマンドとイベントを定義する

ContentViewModel.cs

```
1  namespace Tips_TabControl.ViewModels
2  {
3      using System;
4
5      public class ContentViewModel : NotificationObject, IDisposable
6      {
7          /// <summary>
8          /// 新しいインスタンスを生成します。
9          /// </summary>
10         /// <param name="title">タイトルを指定します。</param>
11         public ContentViewModel(string title)
12         {
13             this.Title = title;
14         }
15
16         private string _title;
17         /// <summary>
18         /// タイトルを取得します。
19         /// </summary>
20         public string Title
21         {
22             get { return this._title; }
23             private set { SetProperty(ref this._title, value); }
24         }
```

```
25
26          private DelegateCommand _closeCommand;
27          /// <summary>
28          /// 閉じるコマンドを取得します。
29          /// </summary>
30          public DelegateCommand CloseCommand
31          {
32              get
33              {
34                  return this._closeCommand ?? (this._closeCommand = new DelegateCommand(_ =>
35                  {
36                      Dispose();
37                      RaiseClosed();
38                  }));
39              }
40          }
41
42          /// <summary>
43          /// 閉じたときに発生します。
44          /// </summary>
45          public event EventHandler<EventArgs> Closed;
46
47          /// <summary>
48          /// Closed イベントを発行します。
49          /// </summary>
50          private void RaiseClosed()
51          {
52              var h = this.Closed;
53              if (h != null)
54                  h(this, EventArgs.Empty);
55          }
56
57          #region IDisposable のメンバ
58          /// <summary>
59          /// リソースの破棄をおこないます。
60          /// </summary>
61          public void Dispose()
62          {
63              Dispose(true);
64              GC.SuppressFinalize(this);
65          }
66
67          /// <summary>
68          /// アンマネージリソースの破棄をおこないます。
69          /// マネージリソースも同時に破棄できます。
70          /// </summary>
71          /// <param name="disposing">マネージリソースも破棄する場合に true を指定します。
    </param>
72          protected virtual void Dispose(bool disposing)
73          {
74              if (disposing)
75              {
76                  // 管理 (managed) リソースの破棄処理をここに記述します。
77              }
78
79              // 非管理 (unmanaged) リソースの破棄処理をここに記述します。
80          }
81
```

```
82          /// <summary>
83          /// デストラクタ
84          /// </summary>
85          ~ContentViewModel()
86          {
87              Dispose(false);
88          }
89          #endregion IDisposable のメンバ
90      }
91  }
```

Closed イベントを定義し、CloseCommand が実行されたときにこのイベントを発行します。このとき、例えば閉じられた後はもう用済みとなる場合には、保持していたリソースを破棄しておく必要があるため、IDisposable インターフェースを実装し、Dispose() メソッドをコールするようにしています。

ContentViewModel に Closed イベントを定義したので、これを MainViewModel が購読し、発行されたときにコレクションから除外する必要があります。

コード 4.43：Closed イベントを購読してコレクション操作をおこなう

MainViewModel.cs

```
1   namespace Tips_TabControl.ViewModels
2   {
3       using System;
4       using System.Collections.ObjectModel;
5
6       public class MainViewModel : NotificationObject
7       {
8           private ObservableCollection<ContentViewModel> _contents = new
    ObservableCollection<ContentViewModel>();
9           /// <summary>
10          /// コンテンツのコレクションを取得します。
11          /// </summary>
12          public ObservableCollection<ContentViewModel> Contents
13          {
14              get { return this._contents; }
15          }
16
17          private DelegateCommand _addContentCommand;
18          /// <summary>
19          /// コンテンツ追加コマンドを取得します。
20          /// </summary>
21          public DelegateCommand AddContentCommand
22          {
23              get
24              {
25                  return this._addContentCommand ?? (this._addContentCommand = new
    DelegateCommand(
26                  _ =>
27                  {
28                      var content = new ContentViewModel("Item" + _count);
29                      _count++;
30                      content.Closed += OnContentClosed;
31                      this.Contents.Add(content);
32                  },
33                  _ => this.Contents.Count < 4));
34              }
35          }
36
```

```
37          /// <summary>
38          /// ContentViewModel の Closed イベントハンドラ
39          /// </summary>
40          /// <param name="sender">イベント発行元</param>
41          /// <param name="e">イベント引数</param>
42          private void OnContentClosed(object sender, EventArgs e)
43          {
44              var content = sender as ContentViewModel;
45              if (content == null)
46                  throw new Exception("ContentViewModel 以外から飛んでくることはあり得ない。");
47
48              content.Closed -= OnContentClosed;
49              this.Contents.Remove(content);
50          }
51
52          private int _count;
53      }
54  }
```

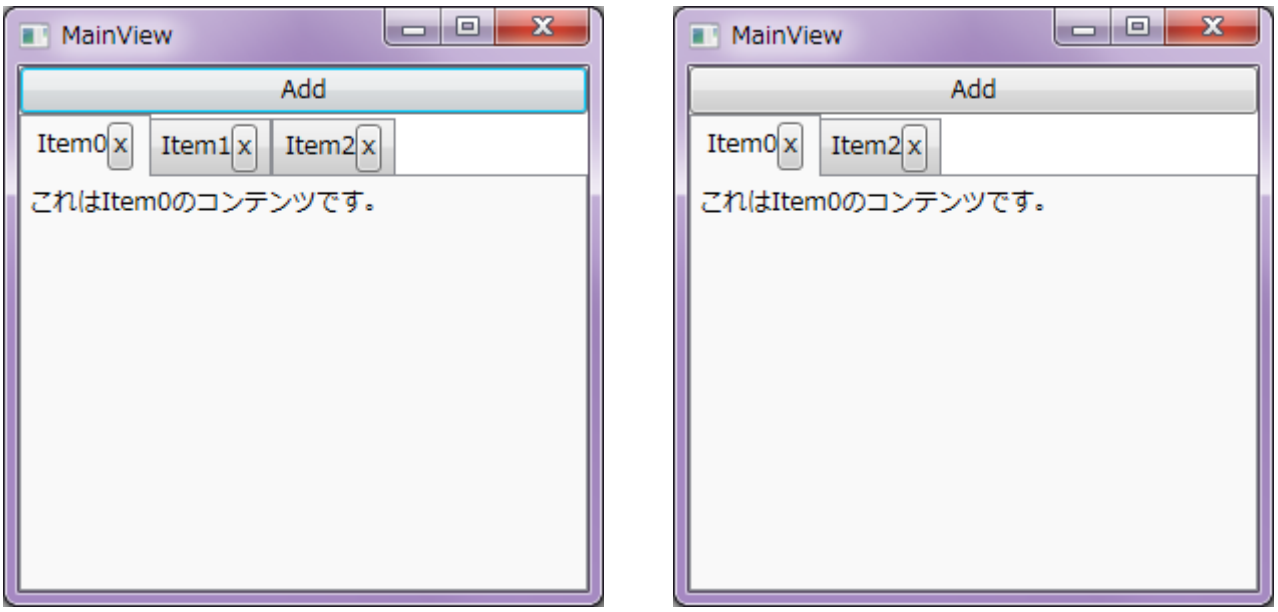
AddContentCommand によってコンテンツを追加するときに Closed イベントを購読することで、すべてのコレクション要素の Closed イベントにイベントハンドラを登録させます。
イベントハンドラ内では、どの要素から発生したイベントなのかを入力引数である sender 変数をアンボックス化することで調べることができます。

閉じるボタンを実装した View は次のように記述します。

コード 4.44 : AddContentCommand をデータバインディングする

MainView.xaml

```
1  <Window x:Class="Tips_TabControl.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      <DockPanel>
6          <Button DockPanel.Dock="Top" Content="Add" Command="{Binding AddContentCommand}"
   />
7
8          <TabControl ItemsSource="{Binding Contents}" SelectedIndex="0">
9              <TabControl.ItemTemplate>
10                 <DataTemplate>
11                     <StackPanel Orientation="Horizontal">
12                         <TextBlock Text="{Binding Title}" VerticalAlignment="Center" />
13                         <Button Content="x" Command="{Binding CloseCommand}" />
14                     </StackPanel>
15                 </DataTemplate>
16             </TabControl.ItemTemplate>
17             <TabControl.ContentTemplate>
18                 <DataTemplate>
19                     <TextBlock Text="{Binding Title, StringFormat={}これは{0}のコンテンツで
   す。}" />
20                 </DataTemplate>
21             </TabControl.ContentTemplate>
22         </TabControl>
23     </DockPanel>
24 </Window>
```



(a) Add ボタンで追加できる　(b) ヘッダ部のボタンを押すとそのタブが削除される

図 4.22：タブを削除するサンプル

4.13 多言語対応にする (Tips_MultiLanguage)

日本語だけでアプリケーションを作っていたら、突然「英語に対応してくれ」と言われたらどうしましょう。固定のテキストを XAML に直接書いていた場合、それを英語で直接書き直すでしょうか。ところが、そうすると日本語版と英語版で異なるソースを管理しないといけなくなります。

WPF では、多言語化されたリソースを用意することで、ひとつのソースで管理できるようになる他、アプリケーション起動後にも動的に言語を切り替えられるようになります。

4.13.1 多言語化されたリソース

WPF ではアセンブリリソースを使用します。図 4.23 のように、プロジェクトを作成すると Properties の中に Resources.resx がデフォルトで生成されているはずなので、これを使います。

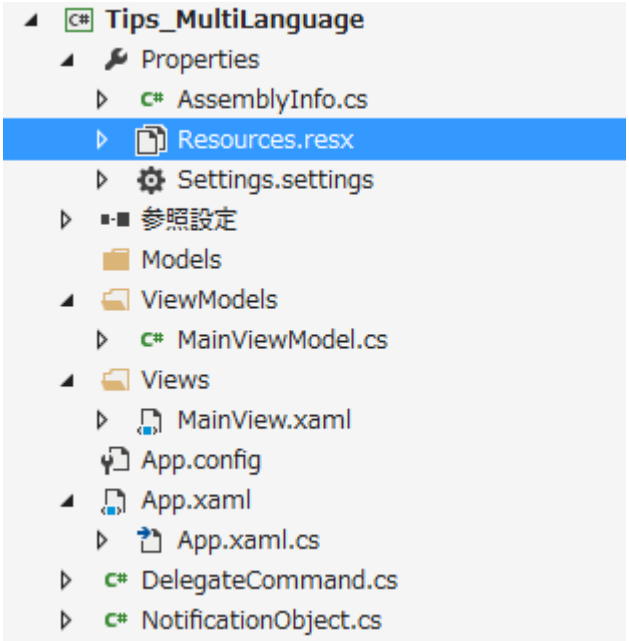


図 4.23：アセンブリリソースファイルを使用する（ない場合は自分で同名ファイルを追加する）

アクセス修飾子を Public にしたら準備完了。例えば 図 4.24 のように名前とそれに対する値を設定します。

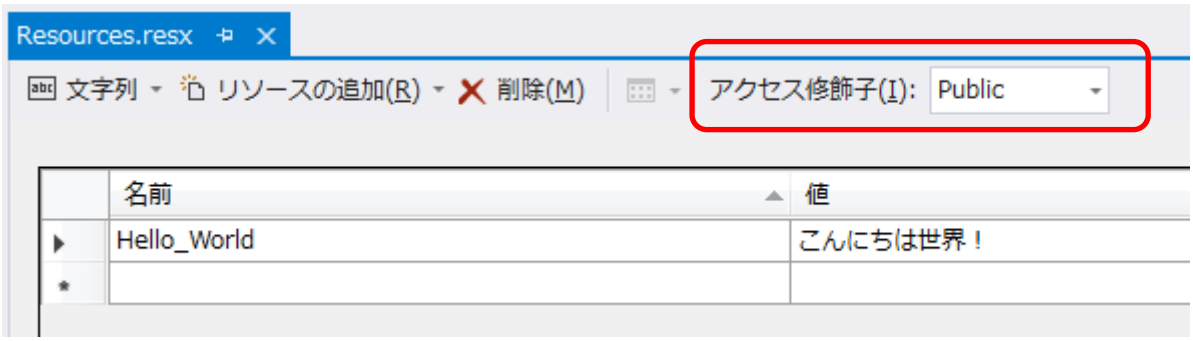


図 4.24：リソースを定義する

次に、例えば英語版のリソースファイルを追加しましょう。別の言語のリソースファイル名は Resources.en-US.resx というように、カルチャ名を間に入れて定義します。Properties 直下には直接ファイルを追加できないので、既にある Resources.resx ファイルをコピー＆ペーストしてリネームするか、一度プロジェクト直下にファイルを追加してから Properties 下に移動するなど工夫してください。図 4.25 は Resources.en-US.resx ファイルを追加した後のツリーです。

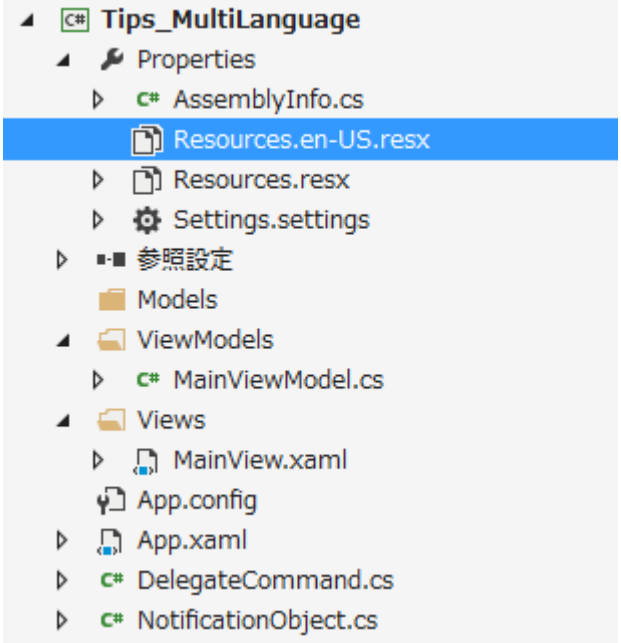


図 4.25：英語用のアセンブリリソースファイルを追加する

こちらのファイルには英語のリソースを定義します。例えば図 4.26 のようにします。ここで、アクセス修飾子はコード生成なしを選択するようにする必要があります。また、先ほど日本語リソースで定義した名前と同じ名前で、値を英語にしたものを定義しておきます。

図 4.26：英語用のリソースを定義する

4.13.2 XAML からリソースを参照する

前節で定義したリソースファイルを XAML から参照するには次のように x:Static を用いて記述します。

コード 4.45：リソースを参照してテキストを表示する

MainView.xaml

```
1  <Window x:Class="Tips_MultiLanguage.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:p="clr-namespace:Tips_MultiLanguage.Properties"
5          Title="MainView" Height="100" Width="200">
6      <StackPanel HorizontalAlignment="Center"
7                  VerticalAlignment="Center">
8          <Button Content="{x:Static p:Resources.Hello_World}" />
9      </StackPanel>
10 </Window>
```

上記のコードを実行すると、図 4.27 のようにリソースで定義した文字列が表示されるようになりました。

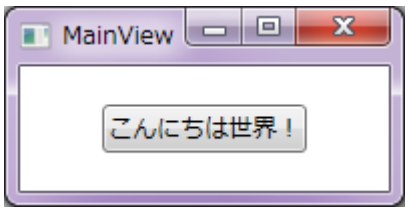


図 4.27：リソースで定義した文字列が表示されている

ところで、リソースは日本語版の Resource.resx と英語版の Resource.en-US.resx を用意していたはずですが、なぜ日本語版のリソースが参照されているのでしょうか。

これは、使用している Windows OS の環境に依存します。このサンプルを起動した環境は日本語版の Windows OS なので、特に指定しない場合は日本語カルチャのリソースを探します。日本語のカルチャ名は ja-JP なので、Resource.ja-JP.resx を探すわけですが、そのようなリソースファイルは存在しません。この場合、実行ファイルに埋め込まれている Resource.resx を参照するようになります。したがって、今回は Resource.resx に日本語の値を定義していたので、結果的に日本語の値が表示されるようになったわけです。

残念ながら英語版の Windows 環境が手元にないため、上記のコードで英語が表示されるという結果を確認できません。ただし、逆に Resource.resx に英語の値、Resource.en-US.resx ではなく、Resource.ja-JP.resx を作成してその中に日本語の値を定義すると、今度は Resource.ja-JP.resx が参照されるため、これもまた日本語が表示されるようになることは確認できますので、やってみてください。

4.13.3 動的な言語切り替え

アプリケーション実行中に言語を切り替えるようにするために、次のようなクラスを用意します。

コード 4.46：リソースを扱うクラスの定義

ResourceServices.cs

```
1   namespace Tips_MultiLanguage
2   {
3       using System.Globalization;
4       using Tips_MultiLanguage.Properties;
5
6       /// <summary>
7       /// 多言語化されたリソースと、言語の切り替え機能を提供するシングルトンクラスを表します。
8       /// </summary>
9       public class ResourceServices : NotificationObject
10      {
11          #region シングルトン
12
13          /// <summary>
14          /// 現在のインスタンスを保持します。
15          /// </summary>
16          private static readonly ResourceServices _current = new ResourceServices();
17
18          /// <summary>
19          /// 現在のインスタンスを取得します。
20          /// </summary>
21          public static ResourceServices Current
22          {
23              get { return _current; }
24          }
25
262         /// <summary>
```

```
27          /// 静的なコンストラクタ
28          /// </summary>
29          static ResourceServices()
30          {
31          }
32
33          /// <summary>
34          /// private なコンストラクタを定義することで
35          /// 外部からこのクラスが生成されることを回避します。
36          /// </summary>
37          private ResourceServices()
38          {
39          }
40
41          #endregion
42
43          /// <summary>
44          /// リソースを保持します。
45          /// </summary>
46          private readonly Resources _resources = new Resources();
47
48          /// <summary>
49          /// 多言語化されたリソースを取得します。
50          /// </summary>
51          public Resources Resources
52          {
53              get { return this._resources; }
54          }
55
56          /// <summary>
57          /// 指定されたカルチャ名を使用して、リソースのカルチャを変更します。
58          /// </summary>
59          /// <param name="name">カルチャの名前を指定します。</param>
60          public void ChangeCulture(string name)
61          {
62              Resources.Culture = CultureInfo.GetCultureInfo(name);
63              this.RaisePropertyChanged("Resources");
64          }
65      }
66  }
```

ChangeCulture() メソッドにカルチャ名を渡すことで、現在のカルチャを変更するようにしています。また、リソースは Resources プロパティで公開するようにしています。

このクラスを利用するために、まず XAML が参照するリソースを ResourceServices クラスの Resources プロパティに変更します。

コード 4.47：リソースの参照先を変更する

MainView.xaml

```
1  <Window x:Class="Tips_MultiLanguage.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:app="clr-namespace:Tips_MultiLanguage"
5          Title="MainView" Height="100" Width="200">
6      <StackPanel HorizontalAlignment="Center"
7                  VerticalAlignment="Center">
8          <Button Content="{Binding Source={x:Static app:ResourceServices.Current},
   Path=Resources.Hello_World}" />
```

```
    </StackPanel>
</Window>
```

この時点では、先ほどの実行結果と同様、Windows OS の環境によって日本語あるいは英語が表示されます。ここで、このボタンに言語切り替えコマンドをデータバインドします。

コード 4.48：言語切り替えコマンドをデータバインドする

MainView.xaml

```
<Window x:Class="Tips_MultiLanguage.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:app="clr-namespace:Tips_MultiLanguage"
        Title="MainView" Height="100" Width="200">
    <StackPanel HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <Button Content="{Binding Source={x:Static app:ResourceServices.Current},
Path=Resources.Hello_World}"
                Command="{Binding ChangeLanguageCommand}"
                />
    </StackPanel>
</Window>
```

コード 4.49：言語切り替えコマンドを実装する

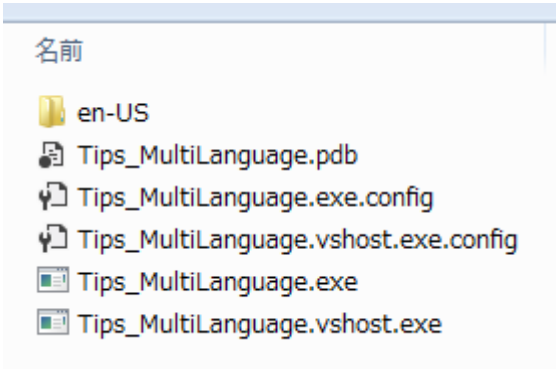
MainViewModel.cs

```
namespace Tips_MultiLanguage.ViewModels
{
    public class MainViewModel : NotificationObject
    {
        private DelegateCommand _changeLanguageCommand;
        /// <summary>
        /// 言語を英語にするコマンドを取得します。
        /// </summary>
        public DelegateCommand ChangeLanguageCommand
        {
            get
            {
                return this._changeLanguageCommand ?? (this._changeLanguageCommand = new
DelegateCommand(_ =>
                {
                    // カルチャを英語に変更する
                    ResourceServices.Current.ChangeCulture("en-US");
                }));
            }
        }
    }
}
```

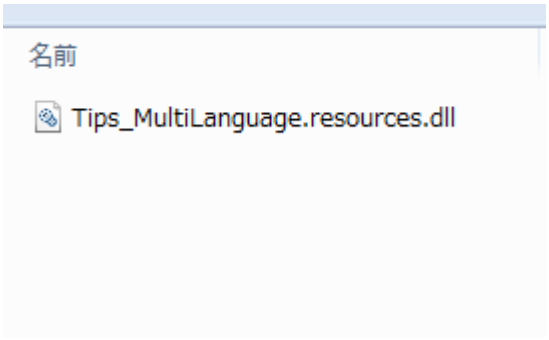
(a) 起動直後　　(b) ボタンを押した後

図 4.28：多言語サンプルの実行結果

途中の説明でも書きましたが、Resource.resx というリソースファイルは実行ファイルの中に埋め込まれますが、後から追加した Resource.en-US.resx などは、ビルドすると図 4.29 のように en-US というフォルダが作られます。



(a) 実行ファイルのあるフォルダ



(b) en-US フォルダの中

図 4.29：ビルド結果

このように実行ファイルに埋め込まれない .dll をサテライトアセンブリといい、カルチャに固有のリソースのみを含んだアセンブリとなっています。つまり、実行ファイルにはニュートラルカルチャのリソース Resources.resx のみが含まれており、多言語化されたリソースについては都度サテライトアセンブリを参照することになります。

したがって、サテライトアセンブリを削除して実行ファイルを起動した場合、削除したアセンブリに該当うする言語の表示ができなくなることになります。

4.14 二重起動させないようにする (Tips_DisabledMultiInstance)

場合によっては同じアプリケーションをいくつも起動させたくないことがあります。WPF でこれを実現するには、Mutex クラスを使用します。

起動させるかどうかを判断させるため、これに関するコードは App.xaml.cs のアプリケーション起動時のイベントハンドラ OnStartup() メソッドに記述します。

コード 4.50：二重起動させないためのサンプルコード

App.xaml.cs

```
namespace Tips_DisabledMultiInstance
{
    using System.Threading;
    using System.Windows;
    using Tips_DisabledMultiInstance.ViewModels;
    using Tips_DisabledMultiInstance.Views;

    /// <summary>
    /// App.xaml の相互作用ロジック
    /// </summary>
    public partial class App : Application
    {
        /// <summary>
        /// 二重起動防止用ミューテックス
        /// </summary>
        private Mutex mutex = new Mutex(false, "Tips_DisabledMultiInstance");

        /// <summary>
        /// 起動時イベントハンドラ
        /// </summary>
        /// <param name="e">イベント引数</param>
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            // ミューテックスの所有権を要求
            if (!mutex.WaitOne(0, false))
            {
                // 既に起動しているため終了させる
                MessageBox.Show(
                    "Tips_DisabledMultiInstance は既に起動しています。",
                    "二重起動防止",
                    MessageBoxButton.OK,
                    MessageBoxImage.Error);
                mutex.Close();
                mutex = null;
                this.Shutdown();
                return;
            }

            var w = new MainView();
            var vm = new MainViewModel();

            w.DataContext = vm;
            w.Show();
        }

        /// <summary>
        /// 終了時イベントハンドラ
```

```
50          /// </summary>
51          /// <param name="e">イベント引数</param>
52          protected override void OnExit(ExitEventArgs e)
53          {
54              if (mutex != null)
55              {
56                  // 使用していたミューテックスを解放、破棄する
57                  mutex.ReleaseMutex();
58                  mutex.Close();
59              }
60
61              base.OnExit(e);
62          }
63      }
64  }
```

Mutex 変数を private フィールドに持ち、このアプリケーションに固有の名前のミューテックスとして作成します。OnStartup() メソッドの中で、ミューテックスの所有権を取得できた場合は通常の起動処理をおこない、取得できなかった場合は Shutdown() メソッドによってアプリケーションを終了させています。

さらに、アプリケーション終了時にコールされる OnExit() メソッドをオーバーライドし、その中で所有していたミューテックスの解放処理をおこなっています。

4.15 Alt+Tab メニューのウィンドウ一覧に表示させないようにする（Tips_AltTabMenuDisable）

特に常駐するようなアプリケーションを作成した場合、Alt+Tab キーで表示されるウィンドウ一覧にアプリケーションを表示させたくないことがあります。このようなとき、対象とするウィンドウの WindowStyle プロパティを ToolWindow にし、ShowInTaskbar プロパティを False にすることで実現できます。

コード 4.51：Alt+Tab メニューのウィンドウ一覧に表示させないようにするコード例

MainView.xaml

```
1  <Window x:Class="Tips_AltTabMenuDisable.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          WindowStyle="ToolWindow"
6          ShowInTaskbar="False">
7      <Grid>
8
9      </Grid>
10 </Window>
```

4.16 「デスクトップの表示」ボタンを押しても最小化しないようにしたい（Tips_ShowDesktopDisable）

Windows7 以降、表示されているすべてのウィンドウを隠してデスクトップを表示するボタンがタスクバーの端っこにあります。マウスカーソルを当てているとツールチップで「デスクトップの表示」と表示されるボタンです。

デスクトップマスコットなど、デスクトップに貼り付いているという想定で作成したアプリケーションでは、「デスクトップの表示」ボタンを押されても、他のウィンドウと一緒に隠れずに、あくまでもデスクトップの一部という位置付けとして表示され続けて欲しいときがあります。そんなときは次のような~~おまじない~~添付ビヘイビアを使いましょう。ただし、対象とするウィンドウは 1 つのみで、そのウィンドウに対して ShowInTaskbar プロパティに False を指定しておく必要があります。

4.16.1 ShowDesktopBehavior 添付ビヘイビア

この添付ビヘイビアでは、アプリケーションのウィンドウを WorkerW あるいは Progman の下に置くようなことをしているらしいのですが、詳細を把握しきれていないため、ここでは説明できません。

コード 4.52：「デスクトップの表示」ボタンで最小化しないようにするための添付ビヘイビア

ShowDesktopBehavior.cs

```
 1  namespace Tips_ShowDesktopDisable.Views.Behaviors
 2  {
 3      using System;
 4      using System.Runtime.InteropServices;
 5      using System.Text;
 6      using System.Windows;
 7
 8      internal static class NativeMethods
 9      {
10          [DllImport("user32.dll")]
11          internal static extern IntPtr SetWinEventHook(uint eventMin, uint eventMax, IntPtr
    hmodWinEventProc, ShowDesktopBehavior.WinEventDelegate lpfnWinEventProc, uint idProcess,
    uint idThread, uint dwFlags);
12
13          [DllImport("user32.dll")]
14          internal static extern bool UnhookWinEvent(IntPtr hWinEventHook);
15
16          [DllImport("user32.dll")]
17          internal static extern int GetClassName(IntPtr hwnd, StringBuilder name, int count);
18      }
19
20      /// <summary>
21      /// 「デスクトップを表示」ボタンを押したとき、ウィンドウを最小化するかどうかを制御するため
    のビヘイビアを表します。
22      /// </summary>
23      public class ShowDesktopBehavior
24      {
25          #region IsEnabled 添付プロパティ
26          /// <summary>
27          /// IsEnabled 添付プロパティの定義
28          /// </summary>
29          public static readonly DependencyProperty IsEnabledProperty =
    DependencyProperty.RegisterAttached("IsEnabled", typeof(bool),
    typeof(ShowDesktopBehavior), new PropertyMetadata(OnIsEnabledPropertyChanged));
30
31          /// <summary>
32          /// IsEnabled 添付プロパティを取得します。
33          /// </summary>
```

```
34          /// <param name="target">対象とする DependencyObject を指定します。</param>
35          /// <returns>取得した値を返します。</returns>
36          public static bool GetIsEnabled(DependencyObject target)
37          {
38              return (bool)target.GetValue(IsEnabledProperty);
39          }
40
41          /// <summary>
42          /// IsEnabled 添付プロパティを設定します。
43          /// </summary>
44          /// <param name="target">対象とする DependencyObject を指定します。</param>
45          /// <param name="value">設定する値を指定します。</param>
46          public static void SetIsEnabled(DependencyObject target, bool value)
47          {
48              target.SetValue(IsEnabledProperty, value);
49          }
50
51          /// <summary>
52          /// IsEnabled 添付プロパティ変更イベントハンドラ
53          /// </summary>
54          /// <param name="sender">イベント発行元</param>
55          /// <param name="e">イベント引数</param>
56          private static void OnIsEnabledPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
57          {
58              var w = sender as Window;
59              if (w == null)
60                  return;
61
62              var isEnabled = (bool)e.NewValue;
63              if (isEnabled)
64              {
65                  RemoveHook(w);
66                  w.SourceInitialized -= OnSourceInitialized;
67              }
68              else
69              {
70                  w.SourceInitialized += OnSourceInitialized;
71              }
72          }
73          #endregion IsEnabled 添付プロパティ
74
75          /// <summary>
76          /// Window クラスのウィンドウハンドルが決定する最速のイベントハンドラ
77          /// </summary>
78          /// <param name="sender">イベント発行元</param>
79          /// <param name="e">イベント引数</param>
80          private static void OnSourceInitialized(object sender, EventArgs e)
81          {
82              var w = sender as Window;
83              AddHook(w);
84          }
85
86          private const uint WINEVENT_OUTOFCONTEXT = 0u;
87          private const uint EVENT_SYSTEM_FOREGROUND = 3u;
88
89          private const string WORKERW = "WorkerW";
90          private const string PROGMAN = "Progman";
```

```
91
92          /// <summary>
93          /// フックを開始します。
94          /// </summary>
95          /// <param name="window">対象とするウィンドウを指定します。</param>
96          public static void AddHook(Window window)
97          {
98              if (IsHooked)
99              {
100                 return;
101             }
102
103             IsHooked = true;
104
105             _window = window;
106             _delegate = new WinEventDelegate(WinEventHook);
107             _hookIntPtr = NativeMethods.SetWinEventHook(EVENT_SYSTEM_FOREGROUND,
    EVENT_SYSTEM_FOREGROUND, IntPtr.Zero, _delegate, 0, 0, WINEVENT_OUTOFCONTEXT);
108         }
109
110         /// <summary>
111         /// フックを解除します。
112         /// </summary>
113         /// <param name="window">対象とするウィンドウを指定します。</param>
114         public static void RemoveHook(Window window)
115         {
116             if (!IsHooked)
117             {
118                 return;
119             }
120
121             IsHooked = false;
122
123             NativeMethods.UnhookWinEvent(_hookIntPtr.Value);
124
125             _delegate = null;
126             _hookIntPtr = null;
127             _window = null;
128         }
129
130         private static string GetWindowClass(IntPtr hwnd)
131         {
132             StringBuilder _sb = new StringBuilder(32);
133             NativeMethods.GetClassName(hwnd, _sb, _sb.Capacity);
134             return _sb.ToString();
135         }
136
137         internal delegate void WinEventDelegate(IntPtr hWinEventHook, uint eventType,
    IntPtr hwnd, int idObject, int idChild, uint dwEventThread, uint dwmsEventTime);
138
139         private static void WinEventHook(IntPtr hWinEventHook, uint eventType, IntPtr hwnd,
    int idObject, int idChild, uint dwEventThread, uint dwmsEventTime)
140         {
141             if (eventType == EVENT_SYSTEM_FOREGROUND)
142             {
143                 string _class = GetWindowClass(hwnd);
144
145                 if (string.Equals(_class, WORKERW, StringComparison.Ordinal) /*||
```

```
    string.Equals(_class, PROGMAN, StringComparison.Ordinal)*/ )
146                 {
147                     _window.Topmost = true;
148                 }
149                 else
150                 {
151                     _window.Topmost = false;
152                 }
153             }
154         }
155 
156         /// <summary>
157         /// フック済みかどうかを取得します。
158         /// </summary>
159         public static bool IsHooked { get; private set; }
160 
161         private static IntPtr? _hookIntPtr { get; set; }
162 
163         private static WinEventDelegate _delegate { get; set; }
164 
165         private static Window _window { get; set; }
166     }
167 }
```

コード 4.53：「デスクトップの表示」ボタンで最小化しないようにするための添付ビヘイビアの使用例

MainView.xaml

```
1  <Window x:Class="Tips_ShowDesktopDisable.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:b="clr-namespace:Tips_ShowDesktopDisable.Views.Behaviors"
5          b:ShowDesktopBehavior.IsEnabled="False"
6          ShowInTaskbar="False"
7          Title="MainView"
8          Height="300" Width="300">
9      <Grid>
10 
11     </Grid>
12 </Window>
```

MainView.xaml で ShowDesktopBehavior.IsEnabled 添付プロパティを False にし、同時に ShowInTaskbar プロパティに False を指定することでデスクトップに貼り付いたような挙動になります。

4.16.2 複数ウィンドウへの適用

前節で紹介した ShowDesktopBehavior 添付ビヘイビアは複数のウィンドウに指定するものではありません。ひとつのアプリケーションで複数のウィンドウを持ち、それらを「デスクトップの表示」ボタンで最小化させないようにする場合は、あるひとつのウィンドウに対して ShowDesktopBehavior 添付ビヘイビアを適用し、その他のウィンドウは添付ビヘイビアを適用したウィンドウをオーナーウィンドウとするようにします。

4.17 ホットキーを登録する (Tips_HotKey)

ホットキーを登録することでユーザーの操作を簡略化させることができます。ここでは、WPF でホットキーを登録する方法を紹介します。

4.17.1 P/Invoke によるネイティブコード呼び出し

結論から言うと、C# ではホットキーを登録するための方法はありません。したがって、P/Invoke を使ってネイティブコードを呼び出す必要があります。

ホットキーを登録または登録解除するには RegisterHotKey 関数および UnregisterHotKey 関数が必要となります。その他にも、登録用の ID と、ホットキーが押されたときのウィンドウメッセージを表す数値が必要です。これらを定義したコードが次のようになります。

コード 4.54：RegisterHotKey 関数と UnregisterHotKey 関数などを組み込む

HotKeyBehavior.cs

```
1  namespace Tips_HotKey.Views.Behaviors
2  {
3      using System;
4      using System.Runtime.InteropServices;
5      using System.Windows;
6      using System.Windows.Input;
7      using System.Windows.Interop;
8
9      public class HotKeyBehavior
10     {
11         /// <summary>
12         /// RegisterHotKey 関数の定義
13         /// </summary>
14         /// <param name="hWnd">ウィンドウハンドル</param>
15         /// <param name="id">固有識別子</param>
16         /// <param name="MOD_KEY">ホットキーに対する修飾キー</param>
17         /// <param name="VK">登録するホットキー</param>
18         /// <returns>0:失敗 (既に他が登録済み)/0以外:成功</returns>
19         [DllImport("user32.dll")]
20         public extern static int RegisterHotKey(IntPtr hWnd, int id, int MOD_KEY, int VK);
21
22         /// <summary>
23         /// UnregisterHotKey 関数の定義
24         /// </summary>
25         /// <param name="hWnd">ウィンドウハンドル</param>
26         /// <param name="id">固有識別子</param>
27         /// <returns>0:失敗/0以外:成功</returns>
28         [DllImport("user32.dll")]
29         public extern static int UnregisterHotKey(IntPtr hWnd, int id);
30
31         #region HotKey 登録用 ID
32         /// <summary>
33         /// HotKey登録時に指定する ID を表します。
34         /// 0x0000～0xbfff で指定してください。
35         /// 0xc000～0xffff は DLL 用に予約済みのため使用できません。
36         /// </summary>
37         public enum HOTKEYs : int
38         {
39             /// <summary>
40             /// HotKey 登録用 ID 1
41             /// </summary>
42             HOTKEY_ID01 = 0x0001,
```

```
43
44              /// <summary>
45              /// HotKey 登録用 ID 2
46              /// </summary>
47              HOTKEY_ID02 = 0x0002,
48
49              /// <summary>
50              /// HotKey 登録用 ID 3
51              /// </summary>
52              HOTKEY_ID03 = 0x0003,
53
54              /// <summary>
55              /// HotKey 登録用 ID 4
56              /// </summary>
57              HOTKEY_ID04 = 0x0004,
58
59              /// <summary>
60              /// HotKey 登録用 ID 5
61              /// </summary>
62              HOTKEY_ID05 = 0x0005,
63
64              /// <summary>
65              /// HotKey 登録用 ID 6
66              /// </summary>
67              HOTKEY_ID06 = 0x0006,
68
69              /// <summary>
70              /// HotKey 登録用 ID 7
71              /// </summary>
72              HOTKEY_ID07 = 0x0007,
73
74              /// <summary>
75              /// HotKey 登録用 ID 8
76              /// </summary>
77              HOTKEY_ID08 = 0x0008,
78
79              /// <summary>
80              /// HotKey 登録用 ID 9
81              /// </summary>
82              HOTKEY_ID09 = 0x0009,
83
84              /// <summary>
85              /// HotKey 登録用 ID 10
86              /// </summary>
87              HOTKEY_ID10 = 0x000A,
88
89              /// <summary>
90              /// HotKey 登録用 ID 11
91              /// </summary>
92              HOTKEY_ID11 = 0x000B,
93
94              /// <summary>
95              /// HotKey 登録用 ID 12
96              /// </summary>
97              HOTKEY_ID12 = 0x000C,
98
99              /// <summary>
100             /// HotKey 登録用 ID 13
```

```
101             /// </summary>
102             HOTKEY_ID13 = 0x000D,
103 
104             /// <summary>
105             /// HotKey 登録用 ID 14
106             /// </summary>
107             HOTKEY_ID14 = 0x000E,
108 
109             /// <summary>
110             /// HotKey 登録用 ID 15
111             /// </summary>
112             HOTKEY_ID15 = 0x000F,
113 
114             /// <summary>
115             /// HotKey 登録用 ID 16
116             /// </summary>
117             HOTKEY_ID16 = 0x0010,
118         }
119         #endregion HotKey 登録用 ID
120 
121         /// <summary>
122         /// WindowMessage - WM_HOTKEY
123         /// </summary>
124         private const int WM_HOTKEY = 0x0312;
    }
}
```

このように、user32.dll をインポートし、必要な関数を extern としてクラスに組み込みます。登録用 ID を列挙体、ウィンドウメッセージを定数として定義しています。

4.17.2 添付ビヘイビアとして機能させる

このクラスを添付ビヘイビアとして機能させるために、Callback、ModifierKey、Key 添付プロパティをそれぞれ定義します。

コード 4.55：添付プロパティの定義

HotKeyBehavior.cs

```
1         #region Callback 添付プロパティ
2         /// <summary>
3         /// Callback 添付プロパティの定義
4         /// </summary>
5         public static readonly DependencyProperty CallbackProperty =
      DependencyProperty.RegisterAttached("Callback", typeof(Action), typeof(HotKeyBehavior),
      new PropertyMetadata(null, OnCallbackChanged));
6 
7         /// <summary>
8         /// Callback 添付プロパティを取得します。
9         /// </summary>
10        /// <param name="target">対象とする DependencyObject を指定します。</param>
11        /// <returns>取得した値を返します。</returns>
12        public static Action GetCallback(DependencyObject target)
13        {
14            return (Action)target.GetValue(CallbackProperty);
15        }
16 
17        /// <summary>
18        /// Callback 添付プロパティを設定します。
19        /// </summary>
```

```
20          /// <param name="target">対象とする DependencyObject を指定します。</param>
21          /// <param name="value">設定する値を指定します。</param>
22          public static void SetCallback(DependencyObject target, Action value)
23          {
24              target.SetValue(CallbackProperty, value);
25          }
26
27          /// <summary>
28          /// Callback 添付プロパティ変更イベントハンドラ
29          /// </summary>
30          /// <param name="sender">イベント発行元</param>
31          /// <param name="e">イベント引数</param>
32          private static void OnCallbackChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
33          {
34              _callback = GetCallback(sender);
35          }
36          #endregion Callback 添付プロパティ
37
38          #region ModifierKey 添付プロパティ
39          /// <summary>
40          /// ModifierKey 添付プロパティの定義
41          /// </summary>
42          public static readonly DependencyProperty ModifierKeyProperty =
    DependencyProperty.RegisterAttached("ModifierKey", typeof(ModifierKeys),
    typeof(HotKeyBehavior), new PropertyMetadata(ModifierKeys.None));
43
44          /// <summary>
45          /// ModifierKey 添付プロパティを取得します。
46          /// </summary>
47          /// <param name="target">対象とする DependencyObject を指定します。</param>
48          /// <returns>取得した値を返します。</returns>
49          public static ModifierKeys GetModifierKey(DependencyObject target)
50          {
51              return (ModifierKeys)target.GetValue(ModifierKeyProperty);
52          }
53
54          /// <summary>
55          /// ModifierKey 添付プロパティを設定します。
56          /// </summary>
57          /// <param name="target">対象とする DependencyObject を指定します。</param>
58          /// <param name="value">設定する値を指定します。</param>
59          public static void SetModifierKey(DependencyObject target, ModifierKeys value)
60          {
61              target.SetValue(ModifierKeyProperty, value);
62          }
63          #endregion ModifierKey 添付プロパティ
64
65          #region Key 添付プロパティ
66          /// <summary>
67          /// Key 添付プロパティの定義
68          /// </summary>
69          public static readonly DependencyProperty KeyProperty =
    DependencyProperty.RegisterAttached("Key", typeof(Key?), typeof(HotKeyBehavior), new
    PropertyMetadata(null, OnKeyPropertyChanged));
70
71          /// <summary>
72          /// Key 添付プロパティを取得します。
```

```
73          /// </summary>
74          /// <param name="target">対象とする DependencyObject を指定します。</param>
75          /// <returns>取得した値を返します。</returns>
76          public static Key? GetKey(DependencyObject target)
77          {
78              return (Key?)target.GetValue(KeyProperty);
79          }
80
81          /// <summary>
82          /// Key 添付プロパティを設定します。
83          /// </summary>
84          /// <param name="target">対象とする DependencyObject を指定します。</param>
85          /// <param name="value">設定する値を指定します。</param>
86          public static void SetKey(DependencyObject target, Key? value)
87          {
88              target.SetValue(KeyProperty, value);
89          }
90
91          /// <summary>
92          /// Key 添付プロパティ変更イベントハンドラ
93          /// </summary>
94          /// <param name="sender">イベント発行元</param>
95          /// <param name="e">イベント引数</param>
96          private static void OnKeyPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
97          {
98              var w = sender is Window ? sender as Window : Window.GetWindow(sender);
99              if (w == null)
100                 return;
101
102             var host = new WindowInteropHelper(w);
103             var _windowHandle = host.Handle;
104
105             var key = GetKey(sender);
106             if (key != null)
107             {
108                 var modifierKey = GetModifierKey(sender);
109                 if (RegisterHotKey(_windowHandle, (int)HOTKEYs.HOTKEY_ID01,
    (int)modifierKey, KeyInterop.VirtualKeyFromKey(key.Value)) != 0)
110                 {
111                     ComponentDispatcher.ThreadPreprocessMessage +=
    ComponentDispatcher_ThreadPreprocessMessage;
112                 }
113             }
114             else
115             {
116                 if (UnregisterHotKey(_windowHandle, (int)HOTKEYs.HOTKEY_ID01) != 0)
117                 {
118                     ComponentDispatcher.ThreadPreprocessMessage -=
    ComponentDispatcher_ThreadPreprocessMessage;
119                 }
120             }
121         }
122
123         #endregion Key 添付プロパティ
124
125         /// <summary>
126         /// コールバック
```

```
127         /// </summary>
128         private static Action _callback;
```

Key 添付プロパティは Key? 型で、null 以外が指定されたときにホットキーを登録、null のときにホットキーを登録解除するようにしています。ホットキーには Key 添付プロパティと ModifierKey 添付プロパティに指定されたキーの組み合わせを指定しています。

キーボードメッセージを受信したときに処理をおこなえるように、ComponentDispatcher クラスの ThreadPreprocessMessage イベントにイベントハンドラを登録しています。このイベントハンドラ内でホットキーかどうかを確認し、該当するホットキーのときに特定の処理をおこなうようにします。
イベントハンドラのコードは次のようになります。

コード 4.56：RegisterHotKey 関数と UnregisterHotKey 関数などを組み込む

HotKeyBehavior.cs

```
1         /// <summary>
2         /// ThreadPreprocessMessage イベントハンドラ
3         /// </summary>
4         /// <param name="msg">ウィンドウメッセージ</param>
5         /// <param name="handled">イベントハンドラ処理をここでストップする場合に処理内で true にします。</param>
6         private static void ComponentDispatcher_ThreadPreprocessMessage(ref MSG msg, ref bool handled)
7         {
8             if (msg.message == WM_HOTKEY)
9             {
10                if (msg.wParam.ToInt32() == (int)HOTKEYs.HOTKEY_ID01)
11                {
12                    if (_callback != null)
13                    {
14                        _callback();
15                        handled = true;
16                    }
17                }
18            }
19        }
```

ウィンドウメッセージが、ホットキーが押された場合であることを確認しています。その中で、自分が登録した ID かどうかを確認した上でコールバックメソッドをコールし、終了したら処理を終えるために handled を true にしています。

以上のように定義した添付ビヘイビアは例えば次のように使用します。

コード 4.57：ホットキーのサンプルに対する ViewModel

MainViewModel.cs

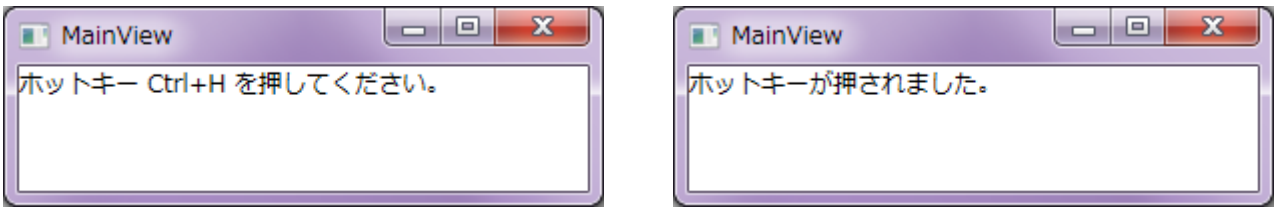
```
1   namespace Tips_HotKey.ViewModels
2   {
3       using System;
4
5       public class MainViewModel : NotificationObject
6       {
7           private string _message = "ホットキー Ctrl+H を押してください。";
8           /// <summary>
9           /// メッセージを取得または設定します。
10          /// </summary>
11          public string Message
12          {
13              get { return this._message; }
14              set { SetProperty(ref this._message, value); }
```

```
15          }
16
17          /// <summary>
18          /// コールバックを取得します。
19          /// </summary>
20          public Action Callback
21          {
22              get { return OnHotKeyDown; }
23          }
24
25          /// <summary>
26          /// コールバック処理をおこないます。
27          /// </summary>
28          private void OnHotKeyDown()
29          {
30              this.Message = "ホットキーが押されました。";
31          }
32      }
33  }
```

コード 4.58：HotKeyBehavior を組み込んだ View

MainView.xaml

```
1  <Window x:Class="Tips_HotKey.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:b="clr-namespace:Tips_HotKey.Views.Behaviors"
5          b:HotKeyBehavior.ModifierKey="Control"
6          b:HotKeyBehavior.Key="H"
7          b:HotKeyBehavior.Callback="{Binding Callback}"
8          Title="MainView" Height="100" Width="300">
9      <StackPanel>
10         <TextBlock Text="{Binding Message}" />
11     </StackPanel>
12 </Window>
```



(a) 起動直後　　(b) ホットキーを押した後

図 4.30：ホットキーサンプルの実行結果

Ctrl+H をホットキーとして指定しています。ホットキーなので、このアプリケーションが非アクティブ状態でもホットキー操作を認識します。

4.18 ハンドルされていない例外を処理する (Tips_UnhandledException)

通常、例外処理をハンドルするために try～catch 構文を用いて、予期せぬエラーに対する処理をおこないます。しかし、この記述を忘れてしまったり、メソッドなどの使い方を間違えてしまったりしていた場合、ハンドルされていない例外が発生してしまうがためにアプリケーションが強制的に終了されてしまうことがあります。もちろん、例外が発生する状況はあってはならないため、アプリケーションが終了することは正しい処理です。しかし、ユーザーにしてみれば、開発者の勝手な都合によって今まで作業していた内容がすべて台無しになってしまう状況はできれば避けて欲しいところです。救済の余地があるのであればそこまで作り込んでおいて欲しいと思います。また、開発中のデバッグ作業においても、ハンドルされていない例外が発生しても、なんらかの処理をしてからアプリケーションを終了させたい場合もあります。

4.18.1 UI スレッドにおける未処理例外の捕捉

WPF では、Application クラスに DispatcherUnhandledException イベントというものがあります。これはまさしく、ハンドルされていない例外が発生したときに発行されるイベントです。イベント引数として、発生した例外を持っているため、ここで必要な処理をおこなうことで、アプリケーションが終了する前に設定を保存したり、ログを残したりすることができます。

下記のサンプルは、ハンドルされている例外を発生させるボタンと、ハンドルされていない例外を発生させるボタンを配置した例を示しています。

コード 4.59：例外を発生させるコマンドを実装した MainViewModel

MainViewModel.cs

```
 1  namespace Tips_UnhandledException.ViewModels
 2  {
 3      using System;
 4      using System.Windows;
 5
 6      public class MainViewModel : NotificationObject
 7      {
 8          private DelegateCommand _handledExceptionCommand;
 9          /// <summary>
10          /// ハンドルされた例外を発生させるコマンドを取得します。
11          /// </summary>
12          public DelegateCommand HandledExceptionCommand
13          {
14              get
15              {
16                  return this._handledExceptionCommand ?? (this._handledExceptionCommand =
    new DelegateCommand(_ =>
17                  {
18                      try
19                      {
20                          throw new Exception("ハンドルされた例外です。");
21                      }
22                      catch (Exception err)
23                      {
24                          MessageBox.Show(err.Message, "例外発生", MessageBoxButton.OK,
    MessageBoxImage.Error);
25                      }
26                  }));
27              }
28          }
29
30          private DelegateCommand _unhandledExceptionCommand;
31          /// <summary>
32          /// ハンドルされていない例外を発生させるコマンドを取得します。
33          /// </summary>
```

```
34          public DelegateCommand UnhandledExceptionCommand
35          {
36              get
37              {
38                  return this._unhandledExceptionCommand ?? (this._unhandledExceptionCommand
    = new DelegateCommand(_ =>
39                  {
40                      throw new Exception("ハンドルされていない例外です。");
41                  }));
42              }
43          }
44      }
45  }
```

コード 4.60：例外を発生させるボタンを配置した MainView

MainView.xaml

```
1  <Window x:Class="Tips_UnhandledException.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          WindowStartupLocation="CenterScreen">
6      <StackPanel>
7          <Button Content="ハンドルされた例外を発生させます" Command="{Binding
    HandledExceptionCommand}" />
8          <Button Content="ハンドルされていない例外を発生させます" Command="{Binding
    UnhandledExceptionCommand}" />
9      </StackPanel>
10 </Window>
```

コード 4.61：未処理例外発生イベントにメソッドを登録しておく

App.xaml.cs

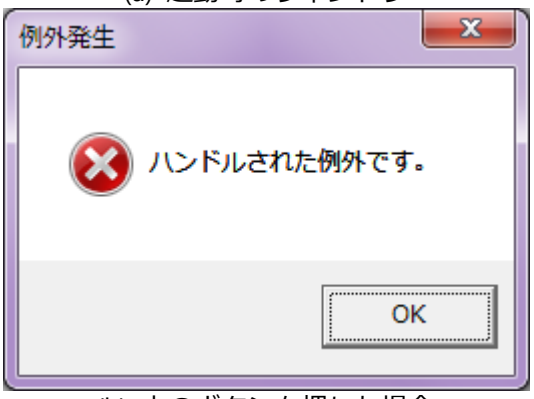
```
1  namespace Tips_UnhandledException
2  {
3      using System.Windows;
4      using System.Windows.Threading;
5      using Tips_UnhandledException.ViewModels;
6      using Tips_UnhandledException.Views;
7
8      /// <summary>
9      /// App.xaml の相互作用ロジック
10     /// </summary>
11     public partial class App : Application
12     {
13         protected override void OnStartup(StartupEventArgs e)
14         {
15             base.OnStartup(e);
16
17             this.DispatcherUnhandledException += OnDispatcherUnhandledException;
18
19             var w = new MainView();
20             var vm = new MainViewModel();
21
22             w.DataContext = vm;
23             w.Show();
24         }
25
26         private void OnDispatcherUnhandledException(object sender,
    DispatcherUnhandledExceptionEventArgs e)
```

```
27          {
28              e.Handled = true;
29              MessageBox.Show(e.Exception.Message, "未処理例外を処理しています。",
    MessageBoxButton.OK, MessageBoxImage.Error);
30          }
31      }
32  }
```

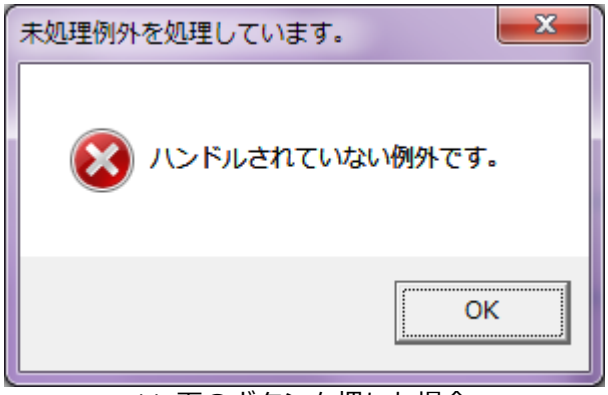
上記のサンプルでは、未処理例外を受け取ったメソッドで、Handled プロパティを true にしています。こうすることで、未処理だった例外を処理済みとして扱うように変更でき、アプリケーションの強制終了を回避することもできます。未処理とはいえ、例えば単なるタイムアウトによる例外だった場合などでは、このように強制終了を回避してもいいかもしれません。

実行結果を下図に示します。ハンドルされていない例外が発生したときに、確かに OnDispatcherUnhandledException() メソッドが実行されている様子がわかります。

(a) 起動時のウィンドウ



(b) 上のボタンを押した場合



(c) 下のボタンを押した場合

図 4.31：未処理例外のサンプル

4.18.2 UI スレッド以外における未処理例外の捕捉

非同期処理をおこなう場合、UI スレッド以外で処理をおこないますが、そのときに例外が発生した場合、前節の DispatcherUnhandledException イベントでは捕捉できません。UI スレッド以外の例外は AppDomain.CurrentDomain.UnhandledException イベントを利用します。

コード 4.62：UI スレッド以外の未処理例外発生イベントにメソッドを登録しておく

App.xaml.cs

```
 1  namespace Tips_UnhandledException
 2  {
 3      using System;
 4      using System.Threading;
 5      using System.Windows;
 6      using System.Windows.Threading;
 7      using Tips_UnhandledException.ViewModels;
 8      using Tips_UnhandledException.Views;
 9
10      /// <summary>
11      /// App.xaml の相互作用ロジック
12      /// </summary>
13      public partial class App : Application
14      {
15          protected override void OnStartup(StartupEventArgs e)
16          {
17              base.OnStartup(e);
18
19              this._uiThreadId = Thread.CurrentThread.ManagedThreadId;
20
21              AppDomain.CurrentDomain.UnhandledException += OnUnhandledException;
22              this.DispatcherUnhandledException += OnDispatcherUnhandledException;
23
24              var w = new MainView();
25              var vm = new MainViewModel();
26
27              w.DataContext = vm;
28              w.Show();
29          }
30
31          private int _uiThreadId;
32
33          private void OnUnhandledException(object sender, UnhandledExceptionEventArgs e)
34          {
35              var caption = string.Format("[UIThread:{0},CurrentThread:{1}]",
    this._uiThreadId, Thread.CurrentThread.ManagedThreadId);
36              MessageBox.Show((e.ExceptionObject as Exception).Message, caption,
    MessageBoxButton.OK, MessageBoxImage.Error);
37              App.Current.Shutdown();
38          }
39
40          private void OnDispatcherUnhandledException(object sender,
    DispatcherUnhandledExceptionEventArgs e)
41          {
42              var caption = string.Format("[UIThread:{0},CurrentThread:{1}]",
    this._uiThreadId, Thread.CurrentThread.ManagedThreadId);
43              MessageBox.Show(e.Exception.ToString(), caption, MessageBoxButton.OK,
    MessageBoxImage.Error);
44              e.Handled = true;
45          }
```

```
46      }
47  }
```

コード 4.63：UI スレッド以外で例外を発生させるコマンドを追加

MainViewModel.cs

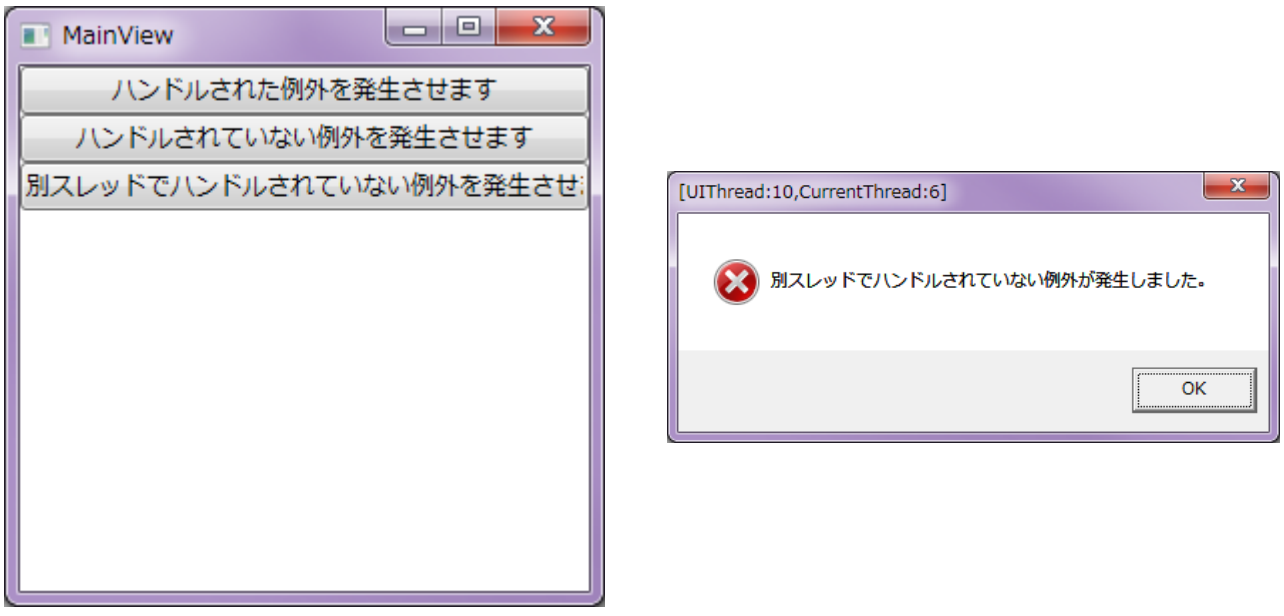
```
 1  namespace Tips_UnhandledException.ViewModels
 2  {
 3      using System;
 4      using System.Windows;
 5      using System.Threading.Tasks;
 6      using System.Threading;
 7
 8      public class MainViewModel : NotificationObject
 9      {
10          private DelegateCommand _handledExceptionCommand;
11          /// <summary>
12          /// ハンドルされた例外を発生させるコマンドを取得します。
13          /// </summary>
14          public DelegateCommand HandledExceptionCommand
15          {
16              get
17              {
18                  return this._handledExceptionCommand ?? (this._handledExceptionCommand =
    new DelegateCommand(_ =>
19                  {
20                      try
21                      {
22                          throw new Exception("ハンドルされた例外です。");
23                      }
24                      catch (Exception err)
25                      {
26                          MessageBox.Show(err.Message, "例外発生", MessageBoxButton.OK,
    MessageBoxImage.Error);
27                      }
28                  }));
29              }
30          }
31
32          private DelegateCommand _unhandledExceptionCommand;
33          /// <summary>
34          /// ハンドルされていない例外を発生させるコマンドを取得します。
35          /// </summary>
36          public DelegateCommand UnhandledExceptionCommand
37          {
38              get
39              {
40                  return this._unhandledExceptionCommand ?? (this._unhandledExceptionCommand
    = new DelegateCommand(_ =>
41                  {
42                      throw new Exception("ハンドルされていない例外です。");
43                  }));
44              }
45          }
46
47          private DelegateCommand _asyncUnhandledExceptionCommand;
48          /// <summary>
49          /// 別スレッドでハンドルされていない例外を発生させるコマンドを取得します。
50          /// </summary>
```

```
51          public DelegateCommand AsyncUnhandledExceptionCommand
52          {
53              get
54              {
55                  return this._asyncUnhandledExceptionCommand ??
    (this._asyncUnhandledExceptionCommand = new DelegateCommand(_ =>
56                  {
57                      var thread = new Thread(() =>
58                      {
59                          throw new Exception("別スレッドでハンドルされていない例外が発生しました。");
60                      });
61                      thread.Start();
62                  }));
63              }
64          }
65      }
66  }
```

コード 4.64：例外を発生させるボタンを配置した MainView

MainView.xaml

```
1  <Window x:Class="Tips_UnhandledException.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          WindowStartupLocation="CenterScreen">
6      <StackPanel>
7          <Button Content="ハンドルされた例外を発生させます" Command="{Binding
    HandledExceptionCommand}" />
8          <Button Content="ハンドルされていない例外を発生させます" Command="{Binding
    UnhandledExceptionCommand}" />
9          <Button Content="別スレッドでハンドルされていない例外を発生させます"
    Command="{Binding AsyncUnhandledExceptionCommand}" />
10     </StackPanel>
11 </Window>
```



(a) 起動時のウィンドウ (b) 追加したボタンを押したとき

図 4.32：UI スレッド以外の未処理例外を捕捉するサンプル

System.Threading.Thread クラスのインスタンスを生成することで UI スレッドとは別のスレッドを立て、その中で例外を発生させています。このとき、App クラスに定義した OnUnhandledException() メソッドが実行されてメッセージダイアログが表示されています。

わかりやすくするために、メッセージダイアログのキャプションに UI スレッドの ID と、例外が発生したスレッドの ID を表示させています。実行結果からわかるように、確かに異なるスレッドで実行されていたことが確認できます。

4.18.3 Task クラスを用いた非同期処理における未処理例外の捕捉

前節で、非同期処理における未処理例外を捕捉できましたが、System.Threading.Tasks.Task クラスを用いた非同期処理をおこなう場合、投げた例外がハンドリングされない場合、Task クラスが吸収してしまい、このままでは虚空の彼方へ消えてしまいます。
例外がなかったことにされないためには、Task.Wait() メソッドまたは Task.Result プロパティにアクセスすることでその例外を捕捉することができます。

コード 4.65：TAP による非同期処理中の例外処理

Program.cs

```
1  /// <summary>
2  /// 非同期処理中に例外を発生させます。
3  /// </summary>
4  public void RaiseExceptionInTAP()
5  {
6      var task = Task.Factory.StartNew(() =>
7      {
8          throw new Exception("非同期処理中に例外が発生してしまいました。");
9      });
10
11     try
12     {
13         task.Wait();
14     }
15     catch (Exception ex)
16     {
17         Console.WriteLine(ex);
18     }
19 }
```

しかし、Task.Wait() メソッドは非同期でおこなわれている処理が終了するまで待機状態となるため、UI フリーズ回避を目的として非同期処理を利用する場合には使えません。この場合、継続タスクを利用してイベントを発生させることで例外発生を通知します。

コード 4.66：非同期処理中の例外発生をイベント通知する

Program.cs

```
1  /// <summary>
2  /// 非同期処理中に例外を発生させます。
3  /// </summary>
4  public void RaiseExceptionInTAP()
5  {
6      Task.Factory.StartNew(() =>
7      {
8          throw new Exception("非同期処理中に例外が発生してしまいました。");
9      })
10     .ContinueWith(task =>
11     {
12         if (task.Exception != null)
13         {
```

```
14              RaiseExceptionOccurred();
15          }
16      }, TaskScheduler.FromCurrentSynchronizationContext());
17  }
18
19  /// <summary>
20  /// 例外が発生したときに発生します。
21  /// </summary>
22  public event EventHandler<EventArgs> ExceptionOccurred;
23
24  /// <summary>
25  /// ExceptionOccurred イベントを発行します。
26  /// </summary>
27  private void RaiseExceptionOccurred()
28  {
29      var h = this.ExceptionOccurred;
30      if (h != null) h(this, EventArgs.Empty);
31  }
```

タスクの中でハンドリングされない例外が発生した場合、タスクがその例外を捕捉します。Task.Exception プロパティが null かどうかでハンドリングされない例外が発生したかどうかを判別できるため、その判定をしてから 14 行目で例外が発生したことを通知するためのイベントを発行しています。ただし、継続タスクは、指定しない限りスレッドプールを利用した非同期処理となってしまうので、UI スレッドで処理されるように TaskScheduler クラスを用いて指定する必要があります。

4.19 独自のマークアップ拡張を作成するには（Tips_MarkupExtension）

4.19.1 基本的な使い方

データバインディングで多用するマークアップ拡張は、XAML 上で {Binding Path} のように記述します。実はこのマークアップ拡張は独自に作成することができます。

マークアップ拡張を作成するときは、MarkupExtension クラスを継承した派生クラスを作成します。例えば次のような CustomBindingExtension クラスを作ります。クラス名は「～Extension」というようにします。
マークアップ拡張の出力は ProvideValue() メソッドの戻り値となります。

コード 4.67：独自のマークアップ拡張

CustomBindingExtension.cs

```
1  namespace Tips_MarkupExtension.Views
2  {
3      using System.Windows.Markup;
4
5      /// <summary>
6      /// 独自のマークアップ拡張を表します。
7      /// </summary>
8      public class CustomBindingExtension : MarkupExtension
9      {
10         /// <summary>
11         /// XAML 上で引数なしでインスタンスを生成するためのコンストラクタです。
12         /// </summary>
13         public CustomBindingExtension()
14         {
15         }
16
17         /// <summary>
18         /// このマークアップ拡張機能で使用するターゲット プロパティの値として提供されるオブジェクトを返します。
19         /// </summary>
20         /// <param name="serviceProvider">マークアップ拡張機能のサービスを提供できるサービス プロバイダー ヘルパー。</param>
21         /// <returns>拡張機能が適用されたプロパティに設定するオブジェクトの値。</returns>
22         public override object ProvideValue(System.IServiceProvider serviceProvider)
23         {
24             return "Hello world.";
25         }
26     }
27 }
```

上記のように定義したクラスは、XAML 上にてマークアップ拡張として記述できるようになります。

コード 4.68：独自のマークアップ拡張使用例

MainView.xaml

```
1  <Window x:Class="Tips_MarkupExtension.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:local="clr-namespace:Tips_MarkupExtension.Views"
5          Title="MainView" Height="300" Width="300">
6      <StackPanel>
7          <TextBlock Text="{local:CustomBinding}" />
8      </StackPanel>
9  </Window>
```

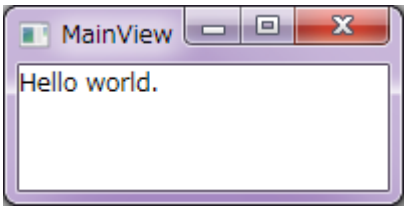


図 4.33：ProvideValue() メソッドの戻り値が表示されている

このままでは固定値しか表示できませんので、少しアレンジしてみましょう。CustomBindingExtension クラスに Text プロパティを追加します。

コード 4.69：独自のマークアップ拡張にプロパティを追加する

CustomBindingExtension.cx

```
1  namespace Tips_MarkupExtension.Views
2  {
3      using System.Windows.Markup;
4
5      /// <summary>
6      /// 独自のマークアップ拡張を表します。
7      /// </summary>
8      public class CustomBindingExtension : MarkupExtension
9      {
10         /// <summary>
11         /// XAML 上で引数なしでインスタンスを生成するためのコンストラクタです。
12         /// </summary>
13         public CustomBindingExtension()
14         {
15         }
16
17         /// <summary>
18         /// 新しいインスタンスを生成します。
19         /// </summary>
20         /// <param name="text">テキストを指定します。</param>
21         public CustomBindingExtension(string text)
22         {
23             this.Text = text;
24         }
25
26         /// <summary>
27         /// テキストを取得または設定します。
28         /// </summary>
29         public string Text { get; set; }
30
31         /// <summary>
32         /// このマークアップ拡張機能で使用するターゲット プロパティの値として提供されるオブジェクトを返します。
33         /// </summary>
34         /// <param name="serviceProvider">マークアップ拡張機能のサービスを提供できるサービス プロバイダー ヘルパー。</param>
35         /// <returns>拡張機能が適用されたプロパティに設定するオブジェクトの値。</returns>
36         public override object ProvideValue(System.IServiceProvider serviceProvider)
37         {
38             return this.Text;
39         }
40     }
41 }
```

外部からアクセス可能なセッターを持つ Text プロパティが追加されたため、XAML は次のように記述できるようになります。

コード 4.70：独自のマークアップ拡張使用例

MainView.xaml

```
<Window x:Class="Tips_MarkupExtension.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Tips_MarkupExtension.Views"
        Title="MainView" Height="100" Width="200">
    <StackPanel>
        <TextBlock Text="{local:CustomBinding Text='Hello world!'}" />
    </StackPanel>
</Window>
```

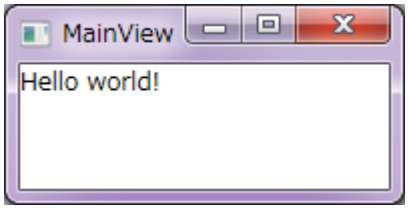


図 4.34：Text プロパティの値がセットされている

一般的にデータバインディングをおこなうとき、マークアップ拡張で {Binding hoge} のように書くと思いますが、これは {Binding Path=hoge} を省略した書き方となります。上記の独自のマークアップ拡張でも、デフォルトのプロパティ名を指定しておくことで、省略した書き方ができるようになります。

次のコードでは、Text プロパティをデフォルトとしているため、XAML 上で少し省略した書き方ができるようになります。引数のあるコンストラクタを定義することで、省略した書き方ができるようになります。さらに、このことについて説明するための属性として ConstructorArgumentAttribute があるので、これも同時に設定しておきましょう。

コード 4.71：独自のマークアップ拡張にプロパティを追加する

CustomBindingExtension.cs

```
namespace Tips_MarkupExtension.Views
{
    using System.Windows.Markup;

    /// <summary>
    /// 独自のマークアップ拡張を表します。
    /// </summary>
    public class CustomBindingExtension : MarkupExtension
    {
        /// <summary>
        /// XAML 上で引数なしでインスタンスを生成するためのコンストラクタです。
        /// </summary>
        public CustomBindingExtension()
        {
        }

        /// <summary>
        /// 新しいインスタンスを生成します。
        /// </summary>
        /// <param name="text">テキストを指定します。</param>
        public CustomBindingExtension(string text)
        {
```

```
23              this.Text = text;
24          }
25
26          /// <summary>
27          /// テキストを取得または設定します。
28          /// </summary>
29          [ConstructorArgument("Text")]
30          public string Text { get; set; }
31
32          /// <summary>
33          /// このマークアップ拡張機能で使用するターゲット プロパティの値として提供されるオブジェクトを返します。
34          /// </summary>
35          /// <param name="serviceProvider">マークアップ拡張機能のサービスを提供できるサービス プロバイダー ヘルパー。</param>
36          /// <returns>拡張機能が適用されたプロパティに設定するオブジェクトの値。</returns>
37          public override object ProvideValue(System.IServiceProvider serviceProvider)
38          {
39              return this.Text;
40          }
41      }
42  }
```

このようなクラスを定義することで、次のようにプロパティ名を省略したマークアップ拡張が記述できるようになります。

コード 4.72：独自のマークアップ拡張でプロパティ名を省略できる

MainView.xaml

```
1  <Window x:Class="Tips_MarkupExtension.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:local="clr-namespace:Tips_MarkupExtension.Views"
5          Title="MainView" Height="100" Width="200">
6      <StackPanel>
7          <TextBlock Text="{local:CustomBinding Hello world!}" />
8      </StackPanel>
9  </Window>
```

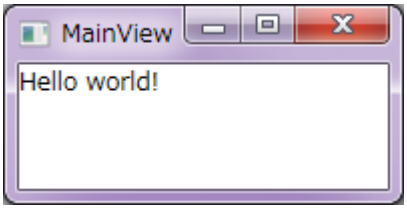


図 4.35：Text プロパティの値がセットされている

4.19.2 Binding マークアップ拡張を自作してみよう

それでは、最小限のコードで標準の Binding マークアップ拡張と同じような機能を持つ独自のマークアップ拡張を作成してみましょう。

コード 4.73：独自のマークアップ拡張にプロパティを追加する

CustomBindingExtension.cs

```
1  namespace Tips_MarkupExtension.Views
2  {
```

```
 3      using System;
 4      using System.Windows;
 5      using System.Windows.Data;
 6      using System.Windows.Markup;
 7
 8      /// <summary>
 9      /// 独自のマークアップ拡張を表します。
10      /// </summary>
11      public class CustomBindingExtension : MarkupExtension
12      {
13          /// <summary>
14          /// XAML 上で引数なしでインスタンスを生成するためのコンストラクタです。
15          /// </summary>
16          public CustomBindingExtension()
17          {
18          }
19
20          /// <summary>
21          /// 新しいインスタンスを生成します。
22          /// </summary>
23          /// <param name="text">プロパティパスを指定します。</param>
24          public CustomBindingExtension(string path)
25          {
26              this.Path = path;
27          }
28
29          /// <summary>
30          /// プロパティパスを取得または設定します。
31          /// </summary>
32          [ConstructorArgument("Path")]
33          public string Path { get; set; }
34
35          /// <summary>
36          /// このマークアップ拡張機能で使用するターゲット プロパティの値として提供されるオブジェクトを返します。
37          /// </summary>
38          /// <param name="serviceProvider">マークアップ拡張機能のサービスを提供できるサービス プロバイダー ヘルパー。</param>
39          /// <returns>拡張機能が適用されたプロパティに設定するオブジェクトの値。</returns>
40          public override object ProvideValue(IServiceProvider serviceProvider)
41          {
42              var service = serviceProvider.GetService(typeof(IProvideValueTarget)) as IProvideValueTarget;
43              if (service == null)
44                  throw new InvalidOperationException();
45
46              var target = service.TargetObject as DependencyObject;
47              var property = service.TargetProperty as DependencyProperty;
48              if ((target == null) || (property == null))
49                  throw new InvalidOperationException();
50
51              var binding = new Binding(this.Path);
52              binding.NotifyOnSourceUpdated = true;
53              binding.Mode = BindingMode.TwoWay;
54              BindingOperations.SetBinding(target, property, binding);
55
56              return target.GetValue(property);
57          }
```

```
58      }
59  }
```

Path プロパティにバインド先のパスを指定させるようにして、XAML 上ではプロパティ名を省略できるように引数付きのコンストラクタを定義しておきます。

ProvideValue() メソッドでは、指定されたプロパティパスを使用して、コード上でバインディングの設定をおこないます。ここで、52 行目で NotifyOnSourceUpdated プロパティを true にしていますが、これはバインド先のデータが変更されたときに、このマークアップ拡張の出力も変更するかどうかを指定するものですので、必ず true にしておきましょう。53 行目でバインディングモードを TwoWay にしていますが、これも XAML 上で指定できるようにプロパティとして持つようにしてもいいかもしれません。54 行目で、以上の設定をおこなった Binding クラスを該当する DependencyProperty に割り当てています。最後にこのメソッドがどんな値を返すかというと、データバインディングを設定したプロパティの値を返すようにします。

作成したマークアップ拡張の動作確認をするために、まず MainViewModel に適当なプロパティを定義します。

コード 4.74：動作確認のために適当なプロパティを定義しておく

MainViewModel.cs

```
1   namespace Tips_MarkupExtension.ViewModels
2   {
3       public class MainViewModel : NotificationObject
4       {
5           private string _text;
6           public string Text
7           {
8               get { return this._text; }
9               set { SetProperty(ref this._text, value); }
10          }
11      }
12  }
```

定義した Text プロパティを操作・表示する UI を次のように定義すると、TextBox の文字列を変更すると、それに追従するように TextBlock の文字列が変化するようになります。

コード 4.75：TextBlock コントロールで表示する

MainViewModel.cs

```
1   <Window x:Class="Tips_MarkupExtension.Views.MainView"
2           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4           xmlns:local="clr-namespace:Tips_MarkupExtension.Views"
5           Title="MainView" Height="100" Width="200">
6       <StackPanel>
7           <TextBlock Text="{local:CustomBinding Text}" />
8           <TextBox Text="{Binding Text, UpdateSourceTrigger=PropertyChanged}" />
9       </StackPanel>
10  </Window>
```

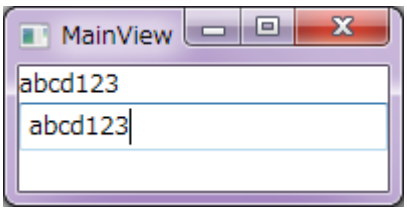


図 4.36：Text プロパティが変化すると表示が更新される

5 アンチパターン

本章では、C# コーディングとしてやってはいけないコードを紹介します。"やってはいけない" というのは、「そのコードで本当に正しく動作していますか？」と疑われてしまうようなコードのことで、そのコードで合っているかもしれませんし、予期せぬ動作をしているかもしれません。他の人や未来の自分が見たとき、あらぬ誤解を招かないようにするために、普段から綺麗なコードを書けるようにしましょう。

5.1 "やってはいけない" コードを検出するコード分析機能を使おう

Visual Studio には静的コード分析という機能があります。これは、ビルドする度に問題を探して警告を表示してくれる機能で、まるでペアプログラミングをしているかのように的確な指摘をしてくれます。

コード分析を使用するためには、プロジェクトの設定を変更する必要があります。図 5.1 のように、「ビルドに対するコード分析の有効化」チェックボックスにチェックを入れることで、ビルド時にコード分析をおこなうようになります。

コード分析のルールとして様々なものが用意されているので、開発するアプリケーションの種類などによって使い分けることができます。

図 5.1：プロジェクト設定でコード分析を有効化する

コード分析を実行すると、"やってはいけない" コードに対して警告を表示してくれるようになります。警告なので無視しても問題ありません。しかし、警告というのは "バグかもしれない" という意味で、本人がそれで良しとしても、後から見直す時や他の人から見ると危うく、"バグかもしれない" と思ってしまいます。これらの警告にきちんと対処することを心掛けて、常に綺麗なコードが書けるようになるためにも、コード分析機能を積極的に使っていきましょう。

5.2 SQL クエリ構文を string.Format で生成してはいけない（CA2100）

"SELECT * FROM" などを代表とする SQL クエリ構文を使用することで、C# からデータベースを参照したり操作したりすることができます。裏を返すと、C# からデータベースを改竄することができるということです。例えば次のようなコードでデータベースを更新したとします。

コード 5.1：string.Format() メソッドによる SQL 構文の構築

Program.cs

```
 1 namespace Tips_CA2100
 2 {
 3     using System;
 4     using MySql.Data.MySqlClient;
 5
 6     class Program
 7     {
 8         static void Main(string[] args)
 9         {
10             var connectionString = "hoge";
11             var connection = new MySqlConnection(connectionString);
12             connection.Open();
13
14             var user = new UserInfo() { Name = "hoge" };
15
16             using (var command = new MySqlCommand())
17             {
18                 command.Connection = connection;
19                 command.CommandText = string.Format("UPDATE `Settings` SET {0} WHERE
   `Name`='{1}';", user.Value, user.Name);
20                 command.ExecuteNonQuery();
21             }
22
23             connection.Close();
24         }
25     }
26
27     class UserInfo
28     {
29         public string Name { get; set; }
30         public int Value { get; set; }
31     }
32 }
```

上記のコードは特定のユーザー名に関するデータを更新するためのコードですが、このコードをコード分析すると、CA2100 の警告が表示されます。CA2100 はセキュリティに関する警告で、「クエリ構文が意図しないものに書き換えられる可能性がありますよ」という意味です。

どういうことかというと、上記のクエリ構文の生成には user.Name プロパティの ToString() メソッドが暗黙的に使用されています。例えば user.Name プロパティの値が "Suzuki" であれば、"Suzuki" という値を Name に持つ Settings テーブルの値を user.Value プロパティの値に更新することになります。

一見どういうことのないコードですが、user.Name プロパティの値にはどのような値が入るのかを考えたことはあるでしょうか。例えば "Suzuki;" のように、最後にセミコロンが入っていると、上記のコードではクエリ構文の途中にセミコロンが入ることになるため、構文エラーとなり、テーブルを更新できなくなります。これだけならそれほど問題にはなりません。では "Suzuki'; DELETE FROM `Settings` WHERE `Name`='Suzuki" ではどうなるでしょうか。"Suzuki" ユーザーの値を更新した後、DELETE 構文によってテーブルデータが削除されてしまいます。ユーザー名とはいえ、なにも制限されていない場合、このようにクエリ構文が改竄され、テーブルデータをユーザーに操作されてしまう危険性が発生してしまいます。また、このようなアプリケーションが想定しない SQL クエリ構文を実行させる攻撃方法を SQL インジェクションと呼びます。

SQL インジェクションの方法はいくつかありますが、中でも CA2100 は ToString() メソッドのオーバーライドによるクエリ文字列改竄の危険性を警告しています。したがって、次のようなパラメータ付きのコマンド文字列を使用することでこの警告を回避できます。

コード 5.2：パラメータ付きコマンド文字列による CA2100 の回避

Program.cs

```
 1 namespace Tips_CA2100
 2 {
 3     using System;
 4     using MySql.Data.MySqlClient;
 5
 6     class Program
 7     {
 8         static void Main(string[] args)
 9         {
10             var connectionString = "hoge";
11             var connection = new MySqlConnection(connectionString);
12             connection.Open();
13
14             var user = new UserInfo() { Name = "hoge" };
15
16             using (var command = new MySqlCommand())
17             {
18                 command.Connection = connection;
19                 command.CommandText = "UPDATE `Settings` SET @value WHERE `Name`=@name";
20
21                 command.Parameters.Add("@value", MySqlDbType.Int32).Value = user.Value;
22                 command.Parameters.Add("@name", MySqlDbType.String).Value = user.Name;
23
24                 command.ExecuteNonQuery();
25             }
26
27             connection.Close();
28         }
29     }
30
31     class UserInfo
32     {
33         public string Name { get; set; }
34         public int Value { get; set; }
35     }
36 }
```

SQL クエリ構文の中に "@" で定義するパラメータを使用します。このパラメータには後から具体的な値を指定します。パラメータの値を指定するには、21 行目や 22 行目のように、Parameters プロパティにパラメータを追加し、その Value プロパティに具体的な値を指定します。

このように実装することで、user.Vaule プロパティや user.Name プロパティの ToString() メソッドを使わずに SQL クエリ構文を構築できるため、CA2100 の警告が回避できます。

しかし、この実装方法だけでは、前述したような SQL インジェクションを完全に回避することはできません。上記のコード例でいえば、user.Name プロパティの文字列にセミコロンなどの特殊文字が存在しないかどうかを検出したり、SQL クエリ構文にストアドプロシージャを使用したりすることも検討する必要があります。

5.3 StreamReader/Writer クラスを using で入れ子にしてはいけない（CA2202）

コード分析では「CA2202 オブジェクトを複数回破棄しない」という警告が表示されるパターンです。どういうことか、次のサンプルコードで詳しく見ていきましょう。

コード 5.3：using が入れ子になったコード

Program.cs

```
 1 namespace Tips_CA2202
 2 {
 3     using System.IO;
 4
 5     class Program
 6     {
 7         static void Main(string[] args)
 8         {
 9             using (var stream = new FileStream("file.txt", FileMode.Open))
10             using (var writer = new StreamWriter(stream))
11             {
12                 writer.Write("hoge");
13             }
14         }
15     }
16 }
```

ここで問題なのは、using が入れ子になっていることではなくて、2 つ目の using 変数が StreamWriter クラスであることです。StreamWriter クラスについて、Microsoft の公式ページでは次のような説明があります。

> **Unless you set the leaveOpen parameter to true, the StreamWriter object calls Dispose() on the provided Stream object when StreamWriter.Dispose is called.**

StreamWriter のコンストラクタには、leaveOpen という bool 型の入力引数があり、これを true に指定しない限り、Dispose() メソッドを呼び出すとき、第 1 引数に渡す Stream オブジェクトの Dispose() メソッドも同時に呼ぶようです。

このことから、上記のサンプルコードでは、using を抜けるとき、StreamWriter クラスの Dispose() メソッドで stream.Dispose() メソッドも同時に呼ばれることになります。そして、その後 Stream クラスの using を抜けるときにも再度 stream.Dispose() メソッドが呼ばれるようになっているため、コード分析で CA2202 の警告が表示されることになります。

この警告の対処法として、leaveOpen フラグを true に指定するように StreamWriter クラスのコンストラクタを変更してもいいのですが、この方法だけではコード分析で CA2202 を回避できないようです。Microsoft の公式ページにもあるように、この場合は try/finally を使い、stream 変数に null を代入するようにしましょう。

コード 5.4：Dispose() メソッドが複数回呼ばれないように stream 変数に null を代入する

Program.cs

```
 1 namespace Tips_CA2202
 2 {
 3     using System.IO;
 4
 5     class Program
 6     {
 7         static void Main(string[] args)
 8         {
 9             Stream stream = null;
10             try
11             {
```

```
12                 stream = new FileStream("file.txt", FileMode.Open);
13                 using (var writer = new StreamWriter(stream))
14                 {
15                     stream = null;
16                     writer.Write("hoge");
17                     // using の直後でないと CA2202 が検出される
18                     //stream = null;
19                 }
20             }
21             finally
22             {
23                 if (stream != null)
24                     stream.Dispose();
25             }
26         }
27     }
28 }
```

ここで、コードのコメントにもあるように、StreamWriter クラスの using の直後に FileStream クラスのインスタンスを参照している stream 変数に null をセットすることがポイントとなります。

まず、12 行目で FileStream クラスのインスタンスを生成しています。この時点で例外が発生した場合、finally の中に入ります。Stream クラスのインスタンスは何も生成されていないため、Dispose() メソッドも呼ばれることなく終了します。

次に、13 行目で StreamWriter クラスのインスタンスを生成しています。ここで例外が発生した場合、StreamWriter クラスのインスタンスは生成されていないため、こちらの Dispose() メソッドは呼ばれません。そして finally の中に入って、FileStream クラスのインスタンスを参照している stream 変数が null になっていないので、こちらの Dispose() メソッドが 1 回だけ呼ばれて終了します。

今度は、16 行目で例外が発生した場合を考えます。using を抜けるので StreamWriter クラスの Dispose() メソッドが呼ばれます。このとき、StreamWriter クラスから FileStream クラスの Dispose() メソッドが呼ばれます。そして finally の中に入るのですが、FileStream クラスのインスタンスを参照していた stream 変数には既に null がセットされているため、ここで Dispose() メソッドが呼ばれることはありません。結果として、FileStream クラスの Dispose() メソッドは 1 回しか呼ばれないことになります。

これに対して、上記のサンプルコードの 15 行目をコメントアウトし、代わりに 18 行目をアンコメントした場合を考えます。このとき、16 行目で例外が発生した場合、using を抜けるので StreamWriter クラスの Dispose() メソッドと FileStream クラスの Dispose() メソッドが呼ばれます。そして finally の中に入り、まだ null がセットされていない stream 変数から FileStream クラスの Dispose() メソッドを呼び出してしまいます。このように、StreamWriter クラスのインスタンスを生成した直後に stream 変数に null をセットしない限り、CA2202 の警告通り、FileStream クラスの Dispose() メソッドが複数回呼ばれる可能性があります。

本来 IDisposable インターフェースを実装したクラスは using を使うべきですが、サンプルコードで使用している StreamWriter クラスのように、渡された Stream オブジェクトの Dispose() メソッドを同時に呼び出すようなクラスを使用する場合は、using を入れ子にせず、従来通り try/finally で記述しなければ安全なコードにできないことに注意する必要があります。

StreamWriter クラスの他に、StreamReader、BinaryReader、BinaryWriter クラスなどが同じ状況のようなので注意が必要です。

蛇足ですが、Dispose() メソッドが正しく実装されていれば、Dispose() メソッドを何回呼び出しても問題はありませんが、動作の保証はできません。また、Dispose() メソッドの仕様変更があった場合に、果たしてその変更に耐え得るかどうかを考えなければならないと思うと、あらかじめ try/finally で Dispose() メソッドが何回も呼び出されないようにしておいたほうが安全であることに変わりはありません。

5.4 コンストラクタ内でオーバーライド可能なメソッドを呼び出してはいけない（CA2214）

コード分析では「CA2214 コンストラクターのオーバーライド可能なメソッドを呼び出しません」という警告が表示されるパターンです。どういうことか、詳しく見ていきましょう。

サンプルとして次のような基本クラスと派生クラスを定義したコンソールアプリケーションを作って見ましょう。

コード 5.5：コンストラクタ内でオーバーライド可能なメソッドを呼び出している派生クラス

Program.cs

```
1  namespace Tips_CA2214
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var derivedClass = new DerivedClass();
10             Console.ReadLine();
11         }
12     }
13
14     public class DerivedClass : BaseClass
15     {
16         public DerivedClass()
17         {
18             Console.WriteLine("継承先のコンストラクタ");
19             this._initialized = "Yes";
20         }
21
22         private string _initialized = "No";
23
24         protected override void Initialize()
25         {
26             Console.WriteLine("初期化されましたか？\r\n - {0}", _initialized);
27         }
28     }
29
30     public class BaseClass
31     {
32         public BaseClass()
33         {
34             Console.WriteLine("基本クラスのコンストラクタ");
35             Initialize();
36         }
37
38         protected virtual void Initialize()
39         {
40             Console.WriteLine("オーバーライドされるのでこちらの処理は実行されません。");
41         }
42     }
43 }
```

基本クラスである BaseClass クラスのコンストラクタでは、Initialize() メソッドが呼び出されています。Initialize() メソッドは仮想メソッドとなっているため、派生クラスからオーバーライドすることができます。

派生クラスである DerivedClass クラスのコンストラクタでは、private フィールドの初期化をおこなっています。また、Initialize() 仮想メソッドをオーバーライドし、初期化したパラメータを表示するようにしています。

このような状況で、派生クラスである DerivedClass クラスのインスタンスを生成すると、次のような結果となります。

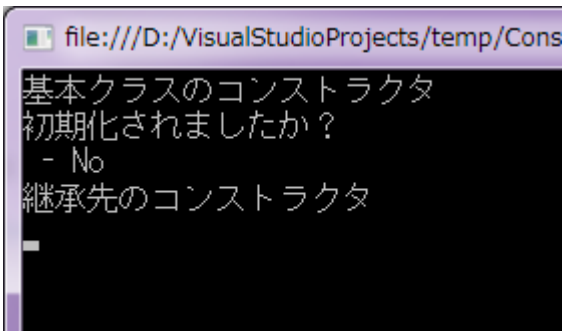


図 5.2：Initialize() メソッドが DerivedClass クラスのコンストラクタが実行される前に処理されている

基本クラスのコンストラクタも呼ばれ、Initialize() メソッドも呼ばれていますが、よく見ると基本クラスで定義している Initialize() メソッドが呼ばれていません。それに、初期化したはずの private フィールドも初期化されていないように見えます。よくよく考えてみると当たり前の結果ですが、順番に説明します。

まず、派生クラスのインスタンスを生成すると、基本クラスのコンストラクタが実行されます。したがって、「基本クラスのコンストラクタ」という表示が最初に現れています。

次に、そのコンストラクタ内で Initialize() 仮想メソッドが呼ばれているため、Initialize() メソッドが処理されます。しかし、このメソッドは派生クラスによってオーバーライドされているため、その処理内容は DerivedClass クラスで定義されている Initialize() メソッドの処理となります。よって、「初期化されましたか？」という表示が現れます。このとき、DerivedClass クラスの private フィールドである _initialized 変数は一度も処理されていないため、宣言時に初期化された値 "No" のままとなっています。

Initialize() メソッドの処理が終わると、基本クラスのコンストラクタ内の処理も終わるため、ようやく継承先のコンストラクタの処理が始まります。

この実行順序は言語規約に沿ったものなので、これはこれで正しいのです。しかし、コンストラクタとしての機能として見たとき、この動作は果たして正しいのでしょうか。

そもそも、コンストラクタとは、そのクラスのインスタンスが生成されたときに自身を初期化するための機能です。その初期化処理の中に、オーバーライド可能な仮想メソッドを使用しているということは、基本クラスからは制御し得ない派生クラスによってその処理内容が改竄されてしまう危険性を含んでいる、ということになります。上記のサンプルでは、base.Initialize() をあえて呼び出していないため、基本クラスの Initialize() メソッドが呼び出されていない結果となっていますが、たとえこれを呼び出していたとしても、その順序は正しいか、同じような処理を二重におこなってないか、など注意しなければいけない点は少なくなさそうです。

こうした理由から、このようなコードはコード分析によって警告として表示されます。できればコンストラクタ内では仮想メソッドを呼び出さないような設計にすべきでしょう。

6 C/C++ による汎用 DLL あれこれ

ここでは C/C++ による汎用 DLL 作成方法と、その DLL を C# から呼び出す方法などについて解説します。

C# は .NET Framework 上で動作するアプリケーションを開発するための強い静的型付け言語であり、自動ボックス化やデリゲートなどの強力な言語支援があるため、アプリケーション開発初心者～中級者にとっては非常に敷居が低い言語であるといえます。

しかし、C# を始めとする .NET 対応言語でビルドされるアプリケーションは、.NET 仮想マシンが直接解釈できる IL マシン語と呼ばれる中間言語にコンパイルされています。そして、実行するときは実際の CPU が直接解釈できるマシン語、つまりネイティブコードにその都度コンパイルする Just-In-Time 方式のコンパイル、通称 JIT コンパイルが実行されることになります。したがって、実行速度はネイティブコードによるアプリケーションよりも必然的に遅くなります。通常の処理であれば特に気にならない性能差でも、計算負荷のオーダーによってはこれがボトルネックになり、アプリケーションの要求仕様を満たせなくなるケースもあります。

そこで、計算負荷の高い処理を C/C++ によるネイティブコードで実行させるために、C/C++ で作成した DLL を C# から参照することができます。C/C++ によって DLL 化された部分はネイティブコードであるため、JIT コンパイルを必要とすることなく、本来の CPU の処理能力そのままに計算をおこなうことができるようになります。

6.1 C/C++ による汎用 DLL 作成用プロジェクト作成手順

Visual Studio のバージョンによっては多少の違いはありますが、概ねここで紹介するような方法でプロジェクトを作成することで C/C++ による汎用 DLL を作成できるようになります。

まず、Visual Studio で新しいプロジェクトを作成し、そのテンプレートとして「Visual C++」→「Win32 プロジェクト」を選択します。

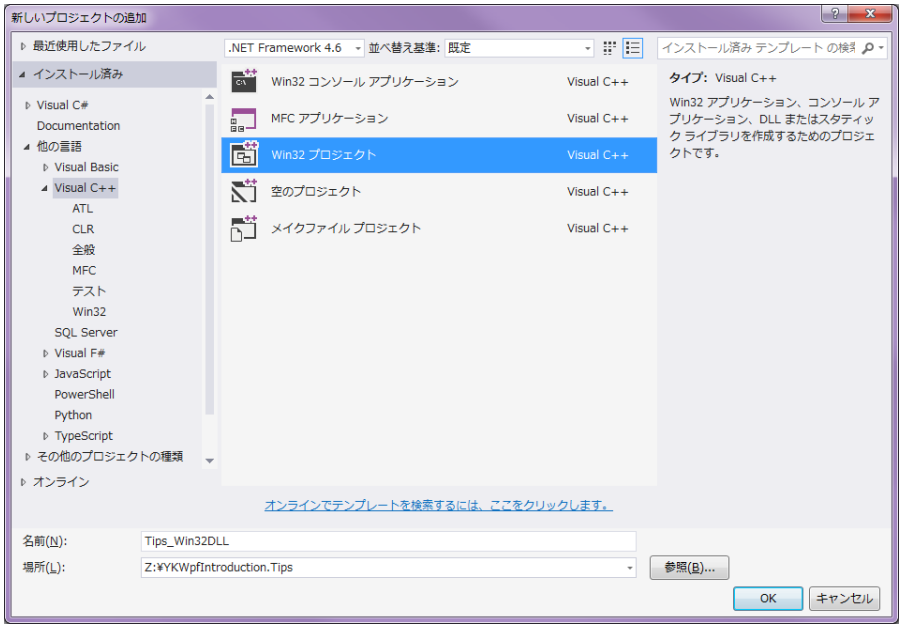


図 6.1：新しいプロジェクトテンプレートを選択するダイアログ

このプロジェクトを追加する際、Win32 アプリケーションウィザードが自動的に開き、プロジェクトの初期設定がおこなわれます。このウィザード上で、アプリケーションの種類を「DLL」に、追加のオプションを「空のプロジェクト」とし、その他のチェックボックスをすべて外してプロジェクトを作成します。

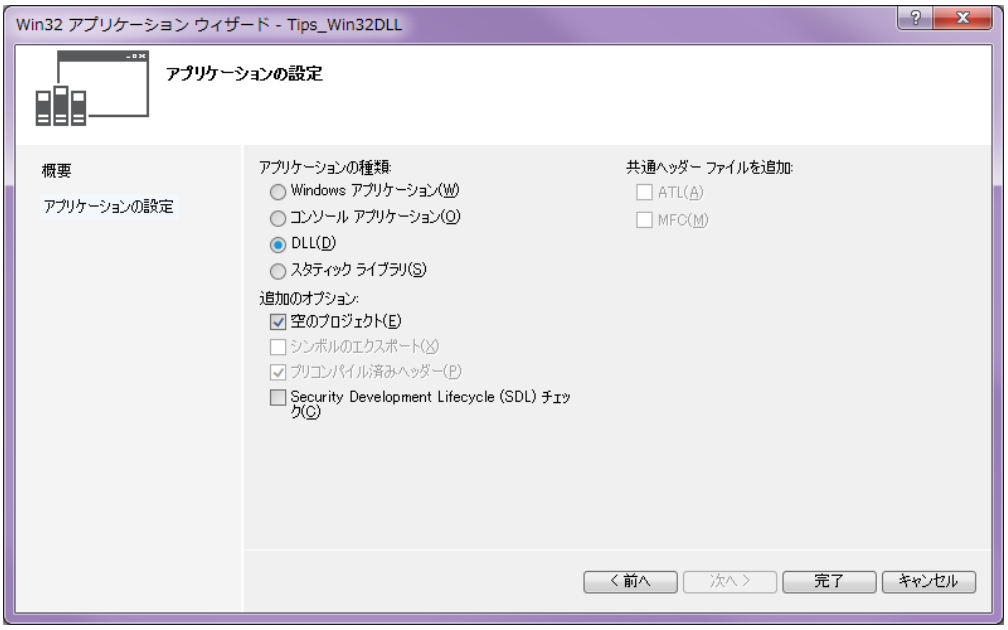


図 6.2：Win32 アプリケーションウィザードの設定例

プロジェクト追加直後はソースファイルやヘッダファイルが空のため、適当な名前のソースファイルおよびヘッダファイルの組を追加します。ここでは「SampleDll.h」、「SampleDll.cpp」を追加しています。

さらに、DLL として公開する関数名を定義するためにモジュール定義ファイルを追加します。このファイルはソリューションエクスプローラで「ソースファイル」を右クリックして「追加」→「新しい項目」メニューから「新しい項目の追加」ダイアログを開き、「Visual C++」→「コード」→「モジュール定義 (.def)」を選択することで追加できます。ここでは「SampleDll.def」を追加しています。

図 6.3：ファイル追加後のソリューションエクスプローラ

モジュール定義ファイルはリンカーへの入力として指定する必要があります。ソリューションエクスプローラからプロジェクトのプロパティを開き、「リンカー」→「入力」→「モジュール定義ファイル」の項目に SampleDll.def と記述します。プロパティの設定は Debug 構成と Release 構成に分かれているため、両方忘れずに設定してください。

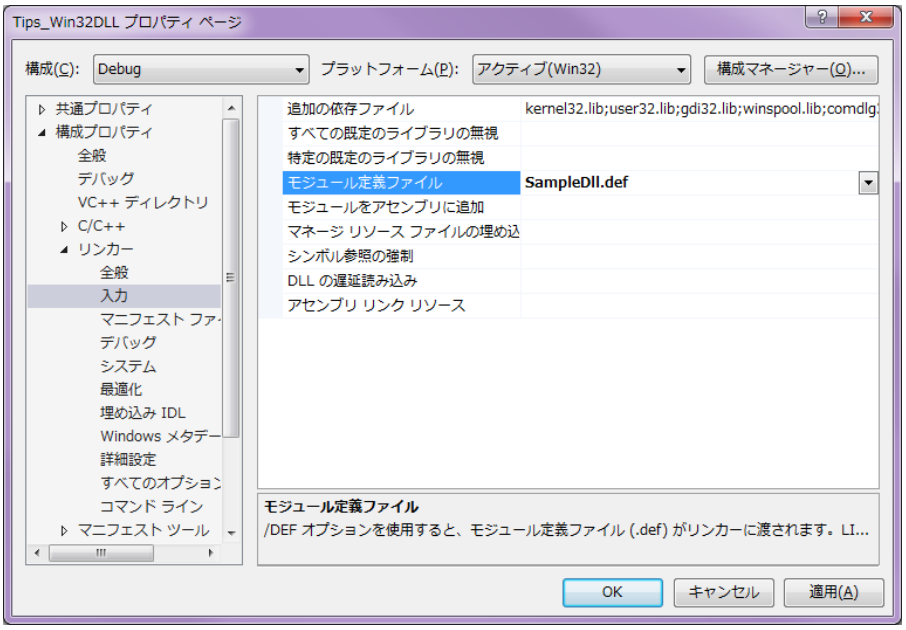


図 6.4：リンカーの設定でモジュール定義ファイルを指定する

また、プロジェクトのプロパティ設定にて、「全般」→「プラットフォーム ツールセット」の項目は必要に応じて WindowsXP 対応のものにしたほうが良いでしょう。

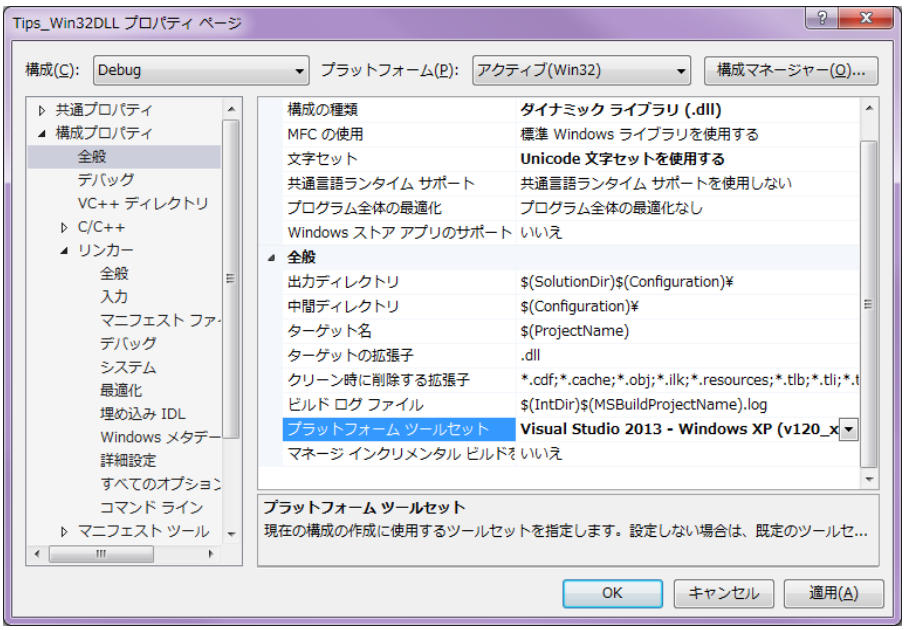


図 6.5：プラットフォームツールセットの設定

以上で C/C++ による汎用 DLL 作成のための準備が整いました。以降では実際のコードを実装していきます。

6.2 DLL として関数を公開するには

C++ で他の言語からでも利用可能な汎用 DLL を作成するには、次のことを守る必要があります。

・クラスではなく関数をエクスポートするようにする
　クラスをエクスポートした場合、DLL 側のコンストラクタ/デストラクタを C# 側から直接呼び出せないため、DLL 側になんらかのヘルパが必要となるため
　クラスのメンバ関数はマングリングによって関数名が自動的に変更され、C# 側は常にその関数名に追従するようにメンテナンスすることが必要となるため

・エクスポートする関数の呼び出し規約は __stdcall とする
　Windows API のデファクトスタンダードであるため

・__declspec(dllexport) は使用せず、モジュール定義ファイル (*.def) でエクスポートする関数を定義する

C++ では異なる名前空間上に同じ関数名が定義されたり、関数のオーバーロード機能によって同じ関数名でも機能が異なる関数が実装されたりします。このため、コンパイル後は定義した関数の名前が自動的に変更されるマングリングという処理がおこなわれてしまいます。したがって、DLL 化するときにも関数の名前が自動的に変更されるため、DLL を使う側が変更された名前を把握しなければならなくなります。しかし、マングリングによる命名規則はコンパイラに依存するため、完全に対応付けることは現実的ではありません。
そこで、関数宣言時に extern "C" を付けることで C++ の関数を C として扱うようにすることができます。つまりオーバーロード機能などがない関数となるため、マングリングによる処理がおこなわれなくなります。結果として、extern "C" を付けて DLL 化することで外部から同じ名前でアクセスできるようになる、というわけです。この場合、関数宣言時に __declspec(dllexport) も同時に付加する必要があります。
一方、extern "C" および __declspec(dllexport) を付けて宣言する代わりに、モジュール定義ファイル「SampleDll.def」に公開する関数名を並べることでも対応できます。どちらの場合でも、関数オーバーロードは使えないようです。

モジュール定義ファイルを使用した場合のコード例を以下に示します。

コード 6.1：ヘッダファイルで関数定義を宣言

SampleDll.h

```
1  #pragma once
2
3  namespace Tips_Win32DLL
4  {
5          double __stdcall Sample01(int a);
6  }
```

コード 6.2：ソースファイルで関数を実装

SampleDll.cpp

```
1  #include <stdio.h>
2  #include "SampleDll.h"
3
4  #define PI              3.1415926536
5
6  namespace Tips_Win32DLL
7  {
8          double __stdcall Sample01(int a)
9          {
10                 printf("--<SampleDll:Sample01>---------------\r\n");
11                 printf("a = %d\r\n", a);
12                 printf("-------------------------------------\r\n");
13
14                 return PI;
15         }
16 }
```

コード 6.3：モジュール定義ファイルに公開する関数名をリストアップする

SampleDll.def

```
1  LIBRARY Tips_Win32DLL
2
3  EXPORTS
4          ; 公開する関数名をリストアップ
5          Sample01
```

モジュール定義ファイルには名前空間を除いた関数名のみを記述します。コメントを書きたい場合は "//" ではなく、";" を記述します。

ヘッダ、ソースファイルともに、エクスポートしたい関数は __stdcall を付けて宣言します。__stdcall は呼び出し規約と呼ばれるキーワードで、その他に次のような呼び出し規約があります。

表 6.1：呼び出し規約

キーワード	スタック維持の責任	パラメータ渡し
__cdecl	呼出元	パラメータをスタックに逆の順序で右から左にプッシュする
__clrcall	適用なし	CLR 式スタックの順に左から右にパラメータを読み込む
__stdcall	呼出先	パラメータをスタックに逆の順序で右から左にプッシュする
__fastcall	呼出先	レジスタに格納されてからスタックにプッシュする
__thiscall	呼出先	スタックにプッシュされる
__vectorcall	呼出先	レジスタに格納されてからスタックに逆の順序で右から左にプッシュされる
__cdecl	呼出元	パラメータをスタックに逆の順序で右から左にプッシュする

呼び出し規約 __stdcall は Windows API で使用されているデファクトスタンダードであり、特別な理由がない限りこれを使用したほうが良いようです。
また、上記のコード例ではクラスを使用していません。というのは、C++ のクラスのメンバ関数の呼び出し規約が __thiscall であり、関数名が自動的に変更されるマングリング処理が働いてしまいます。

以上の作成方法からだいたい察しが付くと思いますが、DLL としてエクスポートできる関数には次のような制約条件があります。

・公開する関数に関数オーバーロードは使えない
・既にグローバルや別の名前空間で定義されている名前の関数はどちらも公開できない
・公開する関数はクラスのメンバ関数にできない

これらの制約があることから、モジュール定義ファイルで公開する関数を管理したほうがメンテナンスしやすいのではないかと思います。また、公開する関数自体はクラスにすることはできませんが、内部実装に関してはクラスを使用することができるので、既にあるクラスが公開している関数を公開するためのラッパーを作ることで既存の資産を流用できます。

6.3 C# から C/C++ DLL で公開されている関数を使用するには

C/C++ DLL は C# による DLL とは異なり、ソリューションエクスプローラの参照設定に追加することができません。したがって、使用したい DLL ファイルを環境設定のパスが通っているディレクトリに保存するか、C# によるアプリケーションファイルと同じディレクトリに保存する必要があります。C# アプリケーション開発中の場合、プロジェクトディレクトリの bin¥Debug ディレクトリに C# アプリケーションが出力されるので、ここに C/C++ で作成した DLL を保存しておくのが一番確実かと思います。

ファイルの準備ができたら、今度は C# コードです。コード内で DLL の関数を使用するには、System.Runtime.InteropServices.DllImport 属性を使用します。

コード 6.4 : C# から C/C++ DLL をインポートする

Program.cs

```
1  namespace Tips_Win32DllImport
2  {
3      using System;
4      using System.Runtime.InteropServices;
5
6      class Program
7      {
8          /// <summary>
9          /// 最も基本的な関数のインポート例
10         /// </summary>
11         /// <param name="a">4 バイト符号付き整数を指定します。</param>
12         /// <returns>倍精度浮動小数を返します。</returns>
13         [DllImport("Tips_Win32DLL.dll")]
14         private static extern double Sample01(int a);
15
16         static void Main(string[] args)
17         {
18             #region Sample01
19             var sample01_a = Sample01(1);
20             Console.WriteLine(sample01_a);
21             #endregion Sample01
22
23             Console.ReadKey();
24         }
25     }
26 }
```

DllImport 属性で対象とする DLL ファイルのファイル名を指定します。エントリポイント名や呼び出し規約などを同時に指定することができます。何も指定しない場合、エントリポイントはここで定義しているメソッド名と同じ名前となり、呼び出し規約は __stdcall となります。

前述のコードによって作成された DLL を用いた場合、上記のコードの実行結果は次のようになります。入力引数がきちんと渡されていること、戻り値をきちんと受け取っていることが確認できます。

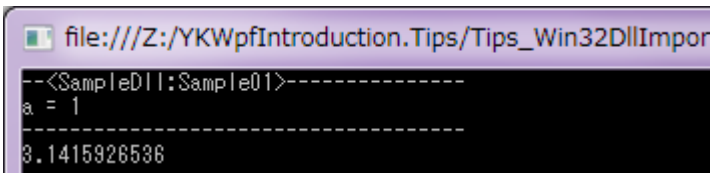


図 6.6 : C# から DLL の関数を呼び出した結果

6.4 文字列を渡す

C/C++ で文字列を扱うには char* 型の変数を使用します。例えば次のような関数を DLL で公開します。

コード 6.5：モジュール定義ファイルに公開する関数名をリストアップする

SampleDll.def

```
LIBRARY Tips_Win32DLL

EXPORTS
        ; 公開する関数名をリストアップ
        Sample02
```

コード 6.6：文字列を受け取る関数

SampleDll.cpp

```
#include <stdio.h>
#include "SampleDll.h"

namespace Tips_Win32DLL
{
        void __stdcall Sample02(int a, char* str)
        {
                printf("--<SampleDll:Sample02>---------------\r\n");
                printf("[%d] %s\r\n", a, str);
                printf("-------------------------------------\r\n");
        }
}
```

与えられた文字列を printf() 関数で表示するだけの関数です。これを C# 側から呼び出すときは、入力引数を string 型として扱います。

コード 6.7：C# から C/C++ DLL をインポートする

Program.cs

```
namespace Tips_Win32DllImport
{
    using System;
    using System.Runtime.InteropServices;

    class Program
    {
        /// <summary>
        /// 文字列を引数に持つ関数のインポート例
        /// </summary>
        /// <param name="a">4 バイト符号付き整数を指定します。</param>
        /// <param name="str">文字列を指定します。</param>
        [DllImport("Tips_Win32DLL.dll")]
        private static extern void Sample02(int a, string str);

        static void Main(string[] args)
        {
            #region Sample02
            var sample02_a = "string 型で文字列を渡すことができます。";
            Sample02(2, sample02_a);
            #endregion Sample02

            Console.ReadKey();
        }
    }
}
```

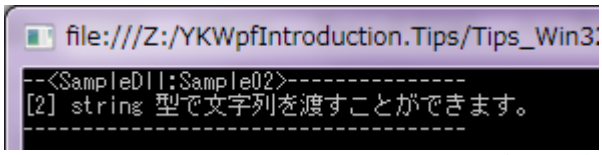


図 6.7：C# から文字列を渡す DLL の関数を呼び出した結果

もし DLL 側で文字列を Unicode として扱う場合は、C# 側では DllImport 属性に CharSet を追加する必要があります。

コード 6.8：文字コードを明示的に指定する

Program.cs

```
13          [DllImport("Tips_Win32DLL.dll", CharSet = CharSet.Unicode)]
14          private static extern void Sample02(int a, string str);
```

6.5 文字列を返してもらう

C/C++ で処理された文字列を C# 側で取得する場合は System.Text.StringBuilder クラスを用いてデータの受け渡しをおこないます。
例えば次のような関数を DLL で公開します。

コード 6.9：モジュール定義ファイルに公開する関数名をリストアップする

SampleDll.def

```
1  LIBRARY Tips_Win32DLL
2
3  EXPORTS
4          ; 公開する関数名をリストアップ
5          Sample03
```

コード 6.10：文字列を受け渡す関数

SampleDll.cpp

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "SampleDll.h"
4
5  namespace Tips_Win32DLL
6  {
7          void __stdcall Sample03(int a, char* str)
8          {
9                  printf("--<SampleDll:Sample03>---------------\r\n");
10                 printf("[%d] %s\r\n", a, str);
11                 sprintf_s(str, 256, "DLL 側から文字列を返す場合は StringBuilder クラスを使用
   します。");
12                 printf("-------------------------------------\r\n");
13         }
14 }
```

与えられた文字列バッファに対して sprintf_s() 関数で文字列をセットしています。これを C# 側から呼び出すときは、入力引数を StringBuilder クラスにする必要があります。

コード 6.11：C# から C/C++ DLL をインポートする

Program.cs

```
1  namespace Tips_Win32DllImport
2  {
3      using System;
4      using System.Runtime.InteropServices;
5      using System.Text;
6
7      class Program
8      {
9          /// DLL 側から文字列を受け取る関数のインポート例
10         /// </summary>
11         /// <param name="a">4 バイト符号付き整数を指定します。</param>
12         /// <param name="str">文字列を受け渡すバッファを指定します。</param>
13         [DllImport("Tips_Win32DLL.dll")]
14         private static extern void Sample03(int a, StringBuilder str);
15
16         static void Main(string[] args)
17         {
18             #region Sample03
19             var sample03_a = new System.Text.StringBuilder(256);
20             sample03_a.Append("文字列のバッファを渡す場合は StringBuilder クラスで受け渡しま
   す。");
```

```
21              Sample03(3, sample03_a);
22              Console.WriteLine(sample03_a);
23              #endregion Sample03
24
25              Console.ReadKey();
26          }
27      }
28  }
```

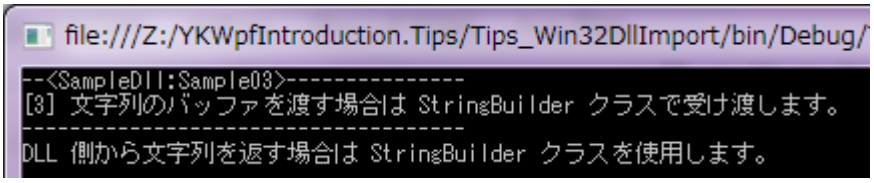


図 6.8：C# から文字列を渡して書き換えてもらう DLL の関数を呼び出した結果

6.6 構造体を渡す

例えば次のような構造体を扱う関数を DLL で公開します。

コード 6.12：モジュール定義ファイルに公開する関数名をリストアップする

SampleDll.def

```
1  LIBRARY Tips_Win32DLL
2
3  EXPORTS
4          ; 公開する関数名をリストアップ
5          Sample04
```

コード 6.13：構造体の定義

structdef.h

```
1  #pragma once
2
3  typedef struct _SampleStruct
4  {
5          int index;
6          char name[128];
7          int data[50];
8  } SampleStruct, *PSampleStruct;
```

コード 6.14：構造体を受け取る関数

SampleDll.cpp

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "SampleDll.h"
4
5  namespace Tips_Win32DLL
6  {
7          void __stdcall Sample04(SampleStruct st)
8          {
9                  printf("--<SampleDll:Sample04>---------------\r\n");
10                 printf("index = %d\r\n", st.index);
11                 printf("name = %s\r\n", st.name);
12                 printf("data[0] = %d, data[1] = %d, data[2] = %d, data[3] = %d\r\n",
   st.data[0], st.data[1], st.data[2], st.data[3]);
13                 printf("-------------------------------------\r\n");
14         }
15 }
```

構造体の場合、C++ では構造体のサイズはコンパイル時に決定されますが、C# では実行時に決定されます。したがって、C# 側で構造体のサイズをあらかじめ指定しておく必要があります。この場合、この構造体は固定長サイズとなります。具体的には次のようなコードになります。

コード 6.15：C# から C/C++ DLL をインポートする

Program.cs

```
1  namespace Tips_Win32DllImport
2  {
3      using System;
4      using System.Runtime.InteropServices;
5
6      class Program
7      {
8          /// <summary>
9          /// 構造体を引数に持つ関数のインポート例
```

```
10          /// </summary>
11          /// <param name="st">DLL 側に渡す構造体を指定します</param>
12          [DllImport("Tips_Win32DLL.dll")]
13          private static extern void Sample04(SampleStruct st);
14
15          /// <summary>
16          /// DLL との取り合いのために定義する構造体です。
17          /// LayoutKind.Sequential を指定することで、
18          /// C/C++ 同様、変数の宣言順通りにメモリに配置されるようにされます。
19          /// </summary>
20          [StructLayout(LayoutKind.Sequential)]
21          private struct SampleStruct
22          {
23              /// <summary>
24              /// 4 バイト符号付整数
25              /// </summary>
26              [MarshalAs(UnmanagedType.I4)]
27              public int index;
28
29              /// <summary>
30              /// 固定長文字配列 (SizeConst は配列のサイズを示す)
31              /// </summary>
32              [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
33              public string name;
34
35              /// <summary>
36              /// 固定長配列 (SizeConst は配列の要素数を示す)
37              /// </summary>
38              [MarshalAs(UnmanagedType.ByValArray, SizeConst = 50)]
39              public int[] data;
40          }
41
42          static void Main(string[] args)
43          {
44              #region Sample04
45              var sample04_a = new SampleStruct()
46              {
47                  index = 4,
48                  name = "構造体サンプル",
49                  data = new int[50],
50              };
51              sample04_a.data[0] = 11;
52              sample04_a.data[1] = 22;
53              sample04_a.data[2] = 33;
54              Sample04(sample04_a);
55              #endregion Sample04
56
57              Console.ReadKey();
58          }
59      }
60  }
```

構造体を定義するとき、MarshalAs 属性を付加することで各フィールドのサイズをコンパイル時に決定させることができます。もちろんこの場合配列の長さは指定した長さでしかインスタンス化できません。

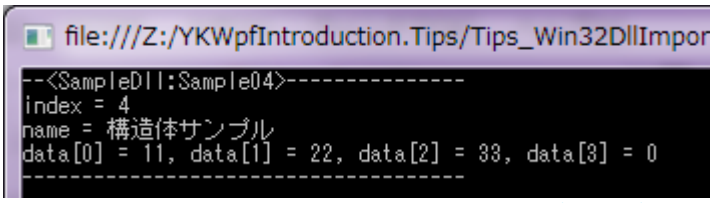


図 6.9：C# から構造体を渡す DLL の関数を呼び出した結果

6.7 構造体を返してもらう

構造体を返してもらう場合、C/C++ では一般的にポインタを用いて受け渡しをおこないます。例えば次のような関数を DLL で公開します。

コード 6.16：モジュール定義ファイルに公開する関数名をリストアップする

SampleDll.def

```
1  LIBRARY Tips_Win32DLL
2
3  EXPORTS
4          ; 公開する関数名をリストアップ
5          Sample05
```

コード 6.17：構造体の定義

structdef.h

```
1  #pragma once
2
3  typedef struct _SampleStruct
4  {
5          int index;
6          char name[128];
7          int data[50];
8  } SampleStruct, *PSampleStruct;
```

コード 6.18：構造体を受け渡す関数

SampleDll.cpp

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "SampleDll.h"
4
5  namespace Tips_Win32DLL
6  {
7          void __stdcall Sample05(SampleStruct* st)
8          {
9                  printf("--<SampleDll:Sample05>---------------\r\n");
10                 //memset(st, 0, sizeof(SampleStruct));
11                 (*st).index = 5;
12                 sprintf_s((*st).name, 128, "構造体ポインタサンプル");
13                 (*st).data[0] = 11;
14                 (*st).data[1] = 22;
15                 (*st).data[2] = 33;
16                 printf("-------------------------------------\r\n");
17         }
18 }
```

前節と同様、構造体を定義するときに MarshalAs 属性を付けてサイズを固定化することに加え、関数の入力引数は SampleStruct 構造体ではなく IntPtr 構造体によるポインタを与えます。

コード 6.19：C# から C/C++ DLL をインポートする

Program.cs

```
1  namespace Tips_Win32DllImport
2  {
3      using System;
4      using System.Runtime.InteropServices;
5
6      class Program
7      {
```

```
8         /// <summary>
9         /// DLL 側から構造体を受け取る関数のインポート例
10        /// </summary>
11        /// <param name="st">受け渡す構造体の先頭アドレスを示すポインタを指定します。</param>
12        [DllImport("Tips_Win32DLL.dll")]
13        private static extern void Sample05(IntPtr st);
14
15        /// <summary>
16        /// DLL との取り合いのために定義する構造体です。
17        /// LayoutKind.Sequential を指定することで、
18        /// C/C++ 同様、変数の宣言順通りにメモリに配置されるようにされます。
19        /// </summary>
20        [StructLayout(LayoutKind.Sequential)]
21        private struct SampleStruct
22        {
23            /// <summary>
24            /// 4 バイト符号付整数
25            /// </summary>
26            [MarshalAs(UnmanagedType.I4)]
27            public int index;
28
29            /// <summary>
30            /// 固定長文字配列 (SizeConst は配列のサイズを示す)
31            /// </summary>
32            [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
33            public string name;
34
35            /// <summary>
36            /// 固定長配列 (SizeConst は配列の要素数を示す)
37            /// </summary>
38            [MarshalAs(UnmanagedType.ByValArray, SizeConst = 50)]
39            public int[] data;
40        }
41
42        static void Main(string[] args)
43        {
44            #region Sample05
45            var sample05_a = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(SampleStruct)));
46            try
47            {
48                Sample05(sample05_a);
49                var sample05_b = (SampleStruct)Marshal.PtrToStructure(sample05_a,
50    typeof(SampleStruct));
51                Console.WriteLine("index = " + sample05_b.index);
52                Console.WriteLine("name = " + sample05_b.name);
53                Console.WriteLine("data[0] = {0}, data[1] = {1}, data[2] = {2}, data[3] =
        {3}", sample05_b.data[0], sample05_b.data[1], sample05_b.data[2], sample05_b.data[3]);
54                Console.WriteLine("DLL 側できちんと初期化していないと data[3] に値が現れるこ
    とに注意する必要がある。");
55            }
56            catch (Exception ex)
57            {
58                System.Diagnostics.Debug.WriteLine(ex);
59            }
60            finally
61            {
62                // 必ずメモリを解放するようにする
63                Marshal.FreeHGlobal(sample05_a);
```

```
64              }
65              #endregion Sample05
66
67              Console.ReadKey();
68          }
69      }
70  }
```

Marshal.SizeOf() メソッドで SampleStruct 構造体のサイズを取得し、Marshal.AllocHGlobal() メソッドでそのサイズ分だけメモリ領域を確保し、その先頭アドレスをポインタ変数 sample05_a に格納しています。このとき、変数 sample05_a が用済みになった段階で必ず Marshal.FreeHGlobal() メソッドでメモリ領域を解放する必要があります。

また、その先頭アドレスから SampleStruct 構造体の情報に構築し直すために、Marshal.PtrToStructhre() メソッドを使用して変数 sample05_b に格納しています。

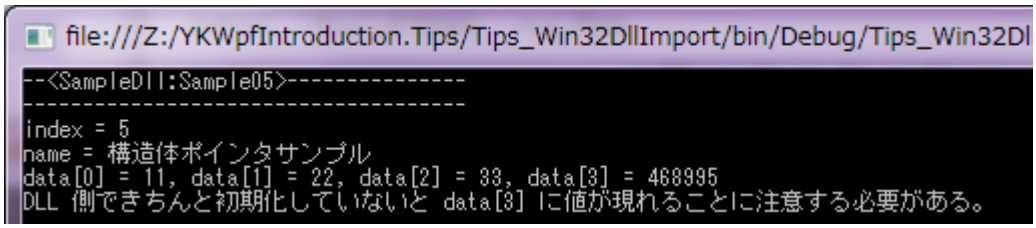


図 6.10：C# から構造体を書き換えてもらう DLL の関数を呼び出した結果

6.8 ポインタを含む構造体を受け渡すには

構造体を受け渡すときに使用した IntPtr 構造体を応用することで、構造体にポインタを含めたデータをやり取りすることもできます。例えば DLL 側で次のような関数を公開します。

コード 6.20：モジュール定義ファイルに公開する関数名をリストアップする

SampleDll.def

```
1  LIBRARY Tips_Win32DLL
2
3  EXPORTS
4          ; 公開する関数名をリストアップ
5          Sample06
```

コード 6.21：メンバにポインタを含む構造体の定義

structdef.h

```
1  #pragma once
2
3  typedef struct _SampleStruct2
4  {
5          int length;
6          double* data;
7  } SampleStruct2, *PSampleStruct2;
```

コード 6.22：メンバにポインタを含む構造体を受け渡す関数

SampleDll.cpp

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "SampleDll.h"
4
5  namespace Tips_Win32DLL
6  {
7          double g_dData[256];
8
9          void __stdcall Sample06(SampleStruct2* st)
10         {
11                 printf("--<SampleDll:Sample06>---------------\r\n");
12                 memset(st, 0, sizeof(SampleStruct2));
13                 memset(g_dData, 0, sizeof(g_dData));
14                 (*st).length = 10;
15                 (*st).data = g_dData;
16                 for (int i = 0; i < (*st).length; i++)
17                 {
18                         g_dData[i] = (i + 1) / 10.0;
19                 }
20                 printf("-------------------------------------\r\n");
21         }
22 }
```

構造体メンバにはアドレスのみを格納し、実体は別の場所にあるような場合を想定しています。C# 側では、構造体に IntPtr 構造体を含めて定義します。

コード 6.23：C# から C/C++ DLL をインポートする

Program.cs

```
1  namespace Tips_Win32DllImport
2  {
3      using System;
4      using System.Runtime.InteropServices;
```

```
5
6       class Program
7       {
8           /// <summary>
9           /// DLL 側からメンバにポインタを含む構造体を受け取る関数のインポート例
10          /// </summary>
11          /// <param name="st">受け渡す構造体の先頭アドレスを示すポインタを指定します。</param>
12          [DllImport("Tips_Win32DLL.dll")]
13          private static extern void Sample06(IntPtr st);
14
15          /// <summary>
16          /// DLL との取り合いのために定義する構造体です。
17          /// LayoutKind.Sequential を指定することで、
18          /// C/C++ 同様、変数の宣言順通りにメモリに配置されるようにされます。
19          /// </summary>
20          [StructLayout(LayoutKind.Sequential)]
21          private struct SampleStruct2
22          {
23              public int length;
24              public IntPtr data;
25          }
26
27          static void Main(string[] args)
28          {
29              #region Sample06
30              var sample06_a = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(SampleStruct2)));
31              try
32              {
33                  Sample06(sample06_a);
34                  var sample06_b = (SampleStruct2)Marshal.PtrToStructure(sample06_a,
    typeof(SampleStruct2));
35                  for (var i = 0; i < sample06_b.length; i++)
36                  {
37                      var v = Marshal.ReadInt64(sample06_b.data, i * sizeof(double));
38                      Console.WriteLine("data[{0}] = {1}", i,
    BitConverter.Int64BitsToDouble(v));
39                  }
40              }
41              catch (Exception ex)
42              {
43                  System.Diagnostics.Debug.WriteLine(ex);
44              }
45              finally
46              {
47                  // 必ずメモリを解放するようにする
48                  Marshal.FreeHGlobal(sample06_a);
49              }
50              #endregion Sample06
51
52              Console.ReadKey();
53          }
54      }
55  }
```

DLL 側から構造体を返してもらうため、その方法に関しては前節と同様、Marshal.AllocHGlobal() メソッドを始めとした IntPtr 構造体を利用した方法を取ります。取得した構造体メンバにもまた IntPtr 構造体が含まれているため、これを利用して値を取得します。

上記のサンプルでは、double 型の配列の先頭アドレスが IntPtr 構造体となっています。IntPtr 構造体から double 型の数値を取得するときは、一度 Int64 に変換し、これを BitConverter.Int64BitsToDouble() メソッドで double 型に変換します。

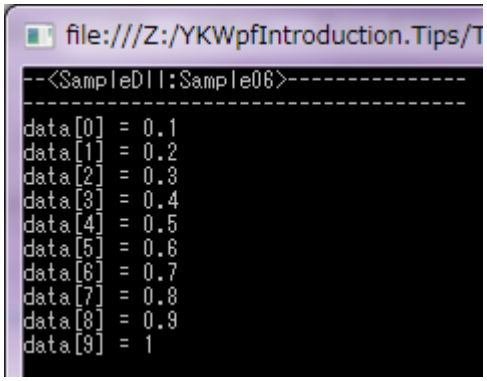


図 6.11：C# からメンバにポインタを含む構造体を書き換えてもらう DLL の関数を呼び出した結果

7 Visual Studio に関するあれこれ

開発環境である Visual Studio に関する設定などを紹介します。

7.1 設定のインポートとエクスポート

「ツール」→「設定のインポートとエクスポート」メニューから、Visual Studio の各種設定をインポートしたりエクスポートしたりすることができます。また、すべての設定をリセットすることもできます。

図 7.1：簡単に設定を保存/読込できる

7.2 ネットワーク共有におけるアクセス許可

ネットワークの共有スペースに保存されているソリューションファイルを直接開いた場合、XAML デザイナーが正常に動作しない場合があります。これは、そのネットワークの信頼性が不明である場合に起き、デザイン実行の際にエラーを出力してしまうようになります。

解決策として、ネットワークの共有スペースにあるソリューションファイルは開かない、という方法が一番ですが、どうしても開きたい場合は、ネットワークドライブとして接続し、ローカルディレクトリのようなパスに配置することでも回避できるようです。

8 おわりに

本書は自分がどうやったらいいのかわからなかったことや、これは便利だと思ったやり方や機能などについてまとめたもので、自分としてはメモ代わりのようなものでもあり、後から何度でも参照できる参考書のようなものでもあります。こういったノウハウは初級者向けに共有することはもちろんですが、上級者となってもすぐに振り返られるようにしておくことが大切ではないかと思います。

ここで紹介したひとつひとつの機能などは web で調べれば簡単に出てくるものもあるかと思います。それでもこの一冊にまとめておくことで自分がどれだけ学んできたかを残しておくことができますし、こうやっていろんな方へ共有することで少しでも技術を伝えていけたらと思います。

改訂履歴

改訂年月日	バージョン	概　要
2015.12.16	1.0.0	初版リリース。
2016.03.03	1.0.1	ドラッグ操作でゴーストを表示する（Tips_Adorner）を追加。 CA2100 に関する節を追加。 節の順序入替、誤植修正。 装飾のコントロールレイアウトを XAML で指定する（Tips_AdornerCore）を追加。
2016.07.08	1.0.2	元に戻す/やり直し機能を実装するには（Tips_Undo）を追加 コントロールを変形させたい（Tips_Transform）を追加 コントロールを変形したときのマウス位置を特定したい を追加 「デスクトップの表示」ボタンを押しても最小化しないようにしたい（Tips_ShowDesktopDisable） を追加 Alt+Tab メニューのウィンドウ一覧に表示させないようにする（Tips+AltTabMenuDisable） を追加 設定のインポートとエクスポート を追加
2016.08.08	1.0.3	第 3 章と第 4 章の節構成を変更 「アンチパターン」の概要を若干修正 独自のマークアップ拡張を作成するには（Tips_MarkupExtension）を追加 シリアル通信をおこなうには（Tips_Serial）を追加 ハンドルされていない例外を処理する（Tips_UnhandledException）の後半を追記
2016.08.25	1.0.4	第 6 章「C/C++ による汎用 DLL あれこれ」を追加 ListBox のアイテム追加/削除のときにアニメーションする（Tips_AnimatedListBoxItem）を追加 外部公開のため細かい部分を修正
2016.10.25	1.0.5	4.9 節と 4.10 節の間に別の節が数ページに渡って挿入されていたのを修正

Windows Presentation Foundation 逆引き集

2015.12.16 初版
2016.10.25 改訂