Windows Presentation Foundation

実践

Ver.1.0.0

```
new WPFer();
```

WPF によるアプリケーション開発を始める方を応援しています

目次

図目次

表目次

コード目次

ワンポイント目次

1 はじめに

この章では本書の目的および執筆環境を掲載します。

1.1 目的

本書は C# によるコンソールアプリケーションを作成することで C# の基本的な機能について学習し、Windows Presentation Foundation（以降 WPF）アプリケーションを作成することで WPF の基本的な使い方や少し複雑な使い方を実践しながら学習することで、開発グループ内の WPF+C# による開発技術力を向上・促進することを目的としています。

1.2 開発環境

本書は以下の環境で執筆しています。

・Windows7 Professional SP1 32 ビットオペレーティングシステム
・Visual Studio Professional 2013 Update5
・.NET Framework 4.6

2 コンソールアプリケーションで学ぶ基礎

この章ではコンソールアプリケーションを作成する過程で C# の基礎を習得することを目的としています。

2.1 コンソールアプリケーションプロジェクトを作成する

Visual Studio を起動後、「ファイル」→「新規作成」→「プロジェクト」を選択すると、次のようなダイアログが開きます。このダイアログの左側のツリーメニューから「Visual C#」を選択すると、真ん中のリストに「コンソールアプリケーション」という項目があるので、これを選択します。そして、ダイアログ下部でプロジェクト名、保存場所、ソリューション名を指定して OK ボタンを押します。

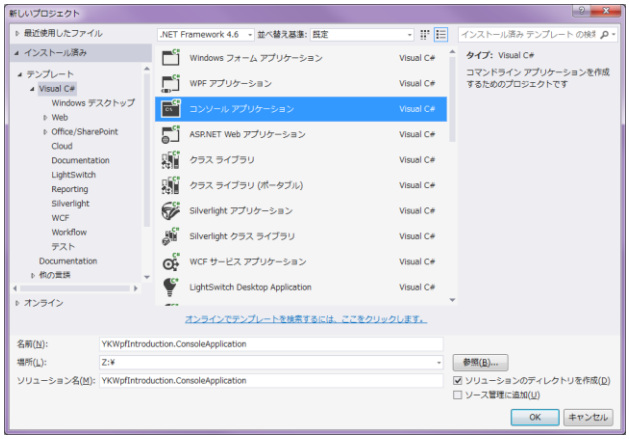


図 2.1：新しいプロジェクトを追加する

Visual Studio では、ひとつの開発案件をソリューションという単位で扱います。ソリューションには複数のプロジェクトを含めることができます。各プロジェクトは WPF アプリケーションだったり、クラスライブラリだったりします。一般的に各プロジェクトは相互に依存し、ひとつのシステムを構築しています。もちろんまったく関係のないプロジェクトをひとつのソリューションに含めても構いません。ソリューションには複数のプロジェクトがあり、各プロジェクトが実際のアプリケーションとなるということさえ覚えておけば問題ありません。

ソリューションには複数のプロジェクトを含めることができますが、始めのうちは単純に WPF アプリケーションだけを作成することが多いと思うので、ソリューションの中にプロジェクトがひとつだけとなるケースが多くなると思います。そのため、ソリューション名とプロジェクト名が同じであるほうがわかりやすいでしょう。ここではサンプルとして YKWpfIntroduction.ConsoleApplication という名前を付けています。

さて、コンソールアプリケーションプロジェクトを新規作成すると、Program.cs というソースファイルがひとつだけのプロジェクトが自動生成され、Program.cs ファイルが開かれた状態になります。Program.cs の中は以下のようになっています。

コード 2.1：自動生成された Program.cs の初期状態

Program.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace YKWpfIntroduction.ConsoleApplication
 8  {
 9      class Program
10      {
11          static void Main(string[] args)
12          {
```

```
13          }
14      }
15  }
```

"using" や "namespace" などなにやらおまじないが書いてありますが、これらに関しては後ほど説明します。重要なのは 11 行目の Main() メソッドです。このメソッドがコンソールアプリケーションのエントリーポイント、つまり起動後に実行されるメソッドで、このメソッドが終了するとアプリケーションが終了することになります。今はこの関数の中が空っぽなので、このまま実行するとすぐに終了するアプリケーションになっています。

それでは試しに実行してみましょう。実行するときは F5 キーを押します。すると、一瞬だけコンソールウィンドウが立ち上がりますが、即座に終了するため、すぐにウィンドウが閉じられて終了します。

すぐに終了してしまうと何かと扱いにくいので、次のように Console.ReadKey() メソッドを追加して、何かキーが押されたら終了するように変更しましょう。

コード 2.2：Console.ReadKey() メソッドでただちに終了しないようにする

Program.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace YKWpfIntroduction.ConsoleApplication
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             Console.ReadKey();
14         }
15     }
16 }
```

コードを追加してから再度実行すると、今度は図 2.2 のようにコンソールウィンドウが開いたままの状態となります。これは、Console.ReadKey() メソッドによって何かキーが押されるまで待機している状態となっているからです。試しにキーボードで何か一つキーを押してみてください。するとアプリケーションが終了してコンソールウィンドウが閉じられます。

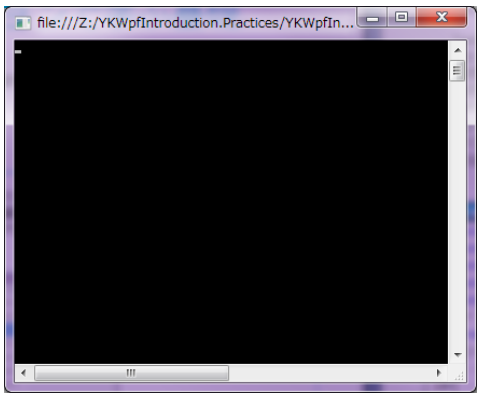


図 2.2：ただちに終了せずキー入力待ち状態となっている

ただちに終了しなくなりましたが、何も表示されていないため、キー入力待ち状態なのか、何か処理中なのかがよくわかりません。わかりやすくするために、"何かキーを押すと終了します。" というメッセージを表示しましょう。メッセージを表示するときは Console.WriteLine() メソッドを使います。

コード 2.3：Console.WriteLine() メソッドでメッセージを表示する

Program.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
```

```
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace YKWpfIntroduction.ConsoleApplication
 8  {
 9      class Program
10      {
11          static void Main(string[] args)
12          {
13              Console.WriteLine("何かキーを押すと終了します。");
14              Console.ReadKey();
15          }
16      }
17  }
```

実行して確認してみましょう。図 2.3 のようになったでしょうか。

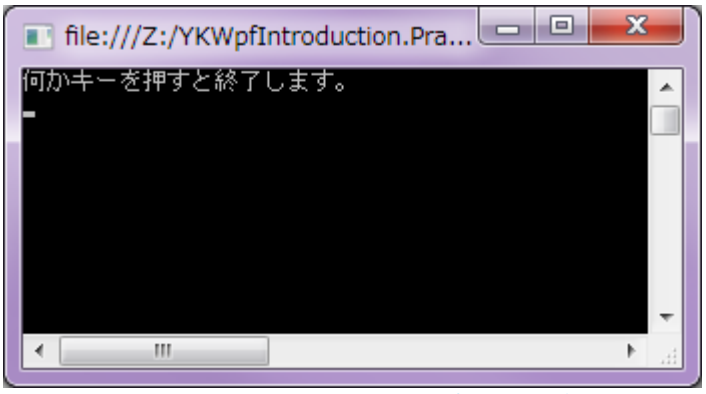


図 2.3：表示されるメッセージで現在の状態がわかる

先ほどから使っている "Console." という表記は、Console クラスのメンバにアクセスする方法です。メンバとは、そのクラスが持っているメソッドやプロパティ、フィールドなどのことを指します。"Console.ReadKey()" というのは、Console クラスの ReadKey() メソッドを呼び出していることになります。Console クラスには他にもいろいろなメンバがあります。これを確認するときは、"Console" と入力したところにキーボードカーソルを置いた状態で F12 キーを押しましょう。すると図 2.4 のように Console クラスの定義ファイルにジャンプします。12 行目から Console クラスの定義が始まり、その中に ReadKey() メソッドや WriteLine() メソッドも含まれていることがわかります。

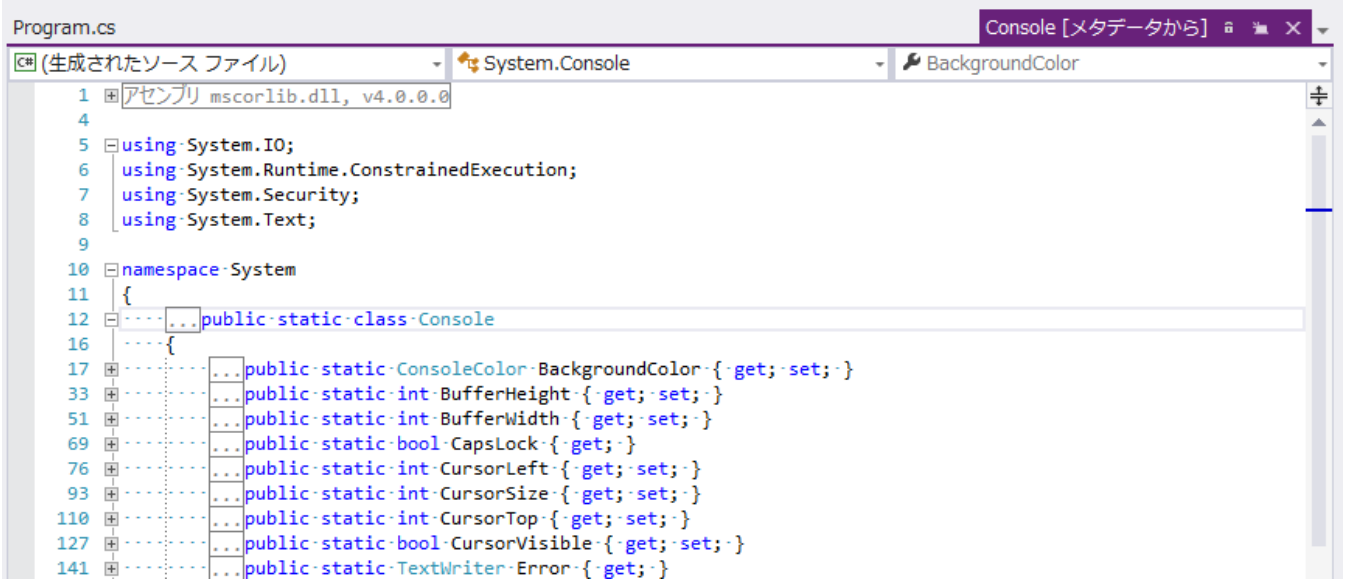


図 2.4：F12 キーで選択されているオブジェクトの定義にジャンプする

ところで、図 2.4 の 10 行目を見てみると、"namespace System" となっています。これは名前空間が System である

ことを表しています。その中に Console クラスの定義があることから、Console クラスは System 名前空間に属していることになります。一方、先ほど自分で作成したコンソールアプリケーションのソースファイル Program.cs で定義されている Main() メソッドを見てみましょう。コード 2.3 の 11 行目に Main() メソッドが定義されていますが、このメソッドは 9 行目にあるように Program クラスのメンバとして定義されています。さらに、この Program クラスは 7 行目にあるように YKWpfIntroduction.ConsoleApplication 名前空間に属しています。Console クラスとは違う名前空間になっている、ということがポイントです。

C# では、クラスを名前空間で分類することができ、.NET Framework では System 名前空間を始め、いくつかの名前空間によって多くのクラスを分類して提供しています。このとき、あるクラスにアクセスするときは必ず名前空間を指定する必要があります。例えば Console クラスは System 名前空間に属しているため、"System.Console" というように書く必要があり、Console クラスのメンバを呼び出す場合は "System.Console.ReadKey()" というように書く必要があります。しかし、クラスが登場する度に "System.…" などとフルネームを書かなければいけないのはとても非効率です。これを省略するための機能が Program.cs の冒頭に書かれている "using" という機能です。

ソースファイルの冒頭に書かれる using ディレクティブは、「この名前空間を使用します」という宣言です。宣言された名前空間に属するクラスを呼び出すときは、その名前空間を省略して記述できるようになります。コード 2.3 をもう一度確認してください。1 行目に "using System;" とあるのがわかります。つまり、Program.cs では System 名前空間を使用することを事前に宣言していたため、13 行目のように唐突に "Console" と記述して System.Console クラスを呼び出すことができています。

ワンポイント 2.1：異なる名前空間の同名のクラス

C# では、クラスはすべてある名前空間に属することになります。このとき、異なる名前空間であれば、同じ名前のクラスを定義することもできます。しかし、その両方の名前空間を using で宣言した場合、どちらのクラスを呼び出しているのか自動判別できなくなるため、コード上では名前空間からフルネームで記述する必要があります。名前空間からフルネームで記述されたクラス名を完全修飾名といいます。

ところで、コード 2.3 の冒頭では "using System;" 以外にもいくつか using で宣言されています。これは、よく使うであろう名前空間があらかじめ宣言されているだけで、実際のコードでは使用していません。使用していない名前空間を書く必要はないので、これらを削除しましょう。削除するときは右クリックメニューの「using の整理」→「削除および並べ替え」が便利です。その名の通り、不要な using を削除し、さらにアルファベット順に並べ替えてくれる機能です。

コード 2.4：「削除および並べ替え」メニューで using ディレクティブを整理

Program.cs

```
1  using System;
2
3  namespace YKWpfIntroduction.ConsoleApplication
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("何かキーを押すと終了します。");
10             Console.ReadKey();
11         }
12     }
13 }
```

また、この using ディレクティブは名前空間の内側に含めることもできます。このようにすることで、もしひとつのソースファイルで複数の名前空間を記述したとき、それぞれで使用する名前空間が異なるときにそれぞれに対して using ディレクティブを記述することができます。もし名前空間の外側に記述する場合は、ソースファイルに含まれるすべての名前空間で使用している using ディレクティブが一緒くたになります。

コード 2.5：名前空間の内側に using ディレクティブを書く

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
```

```
 7          static void Main(string[] args)
 8          {
 9              Console.WriteLine("何かキーを押すと終了します。");
10              Console.ReadKey();
11          }
12      }
13  }
```

2.2 Person クラスを作ってみよう

さて、ここでは独自のクラスとして Person という名前のクラスを作成します。自動生成された Program.cs に追加しても動作上問題ありませんが、どのソースにどのクラスの定義が書いてあるのかわかりにくくなるため、ファイル単位でクラスを定義するようにしましょう。

ソリューションエクスプローラーでプロジェクトを右クリックし、「追加」→「クラス」を選択します。「新しい項目の追加」ダイアログで名前を "Person" として追加ボタンを押します。

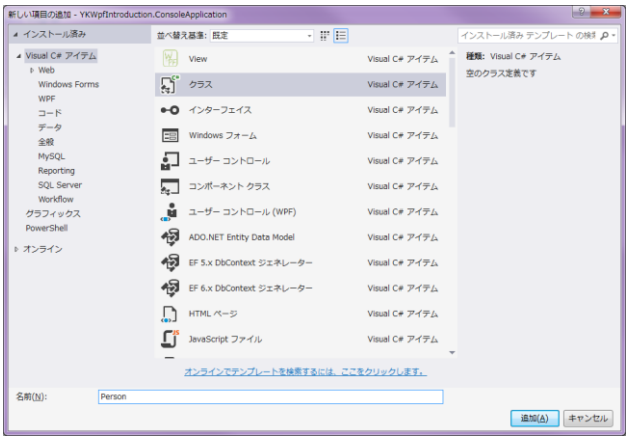


図 2.5：Person.cs ファイルを追加する

すると、自動生成された Program.cs のようにあらかじめ using ディレクティブがいくつか記述された Person クラスのスケルトンが自動生成されます。

コード 2.6：自動生成された Person.cs

Person.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace YKWpfIntroduction.ConsoleApplication
 8  {
 9      class Person
10      {
11      }
12  }
```

とりあえず不要な using ディレクティブをすべて削除し、次のように編集しましょう。

コード 2.7：中身が空っぽの Person クラス

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5      }
6  }
```

コード 2.6 とコード 2.7 を見比べると、Person クラスの定義が若干異なります。コード 2.7 のほうには "internal" という修飾子が付いています。これは、このクラスをどのように宣言するかを修飾するためのキーワードで、他にも "public" や "private" などがあります。これらの修飾子を下表にまとめます。

表 2.1：アクセス修飾子

名称	意味	備考
public	異なるアセンブリやクラスからアクセスできる	
private	同じクラスだけで使用できる	
protected	同じクラスまたは派生クラスだけで使用できる	
internal	同じアセンブリだけで使用できる	
protected internal	同じアセンブリまたは派生クラスだけで使用できる	派生クラスは異なるアセンブリでも問題ない

アセンブリとは .exe や .dll といったひとつのアプリケーションを指します。例えば本章のコンソールアプリケーションプロジェクトをビルドすると YKWpfIntroduction.ConsoleApplication.exe というファイルが出力されます。これがアセンブリです。これに対して、前節で使っていた System.Console クラスは mscorlib.dll というファイルが提供しているクラスなので、YKWpfIntroduction.ConsoleApplication.exe というアセンブリとは異なる mscorlib.dll というアセンブリに含まれています。ところで、図 2.4 の System.Console クラスの定義を見てみると、"public static Console" と記述されています。"public" とあるので、異なるアセンブリからアクセスできるようになっています。このことから、YKWpfIntroduction.ConsoleApplication.exe というアセンブリから System.Console クラスにアクセスできるようになっています。

あるアプリケーションを作成するとき、それが dll でない限りは異なるアセンブリに公開することはないので、public 修飾子をクラス定義に対して使用することはほとんどないと思います。その代わり、同じアセンブリのすべてのクラスに公開するという意味合いの internal 修飾子が良く使われます。

2.2.1 メンバ変数とプロパティ

さて、追加した Person クラスにひとつフィールドを追加しましょう。

コード 2.8：public フィールドとしてメンバ変数 Statement を追加

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 発言内容
7          /// </summary>
8          public string Statement;
9      }
10 }
```

メンバ変数 Statement は string 型で文字列を格納します。この人は何か言いたいことがあるようで、その意見を public フィールドであるメンバ変数 Statement に保持するようにしています。public 修飾子を付けているので、異なるクラスからアクセスできるようになっています。

それではこの Person クラスを実際に使ってみましょう。Program.cs の Main() メソッドに戻って、この Person クラスをインスタンス化します。

コード 2.9：Program クラスで Person クラスをインスタンス化する

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              // Person クラスをインスタンス化
```

```
10              var person = new Person();
11
12              Console.WriteLine("何かキーを押すと終了します。");
13              Console.ReadKey();
14          }
15      }
16  }
```

クラスは定義するだけではメモリ上に配置されません。上記のコードのように new 演算子でコンストラクタを呼び出すことで初めてメモリ上に配置されて動作するようになります。メモリ上に実際にクラスが配置されるので、「クラスを実体化する」や「クラスをインスタンス化する」などと言います。

10 行目で変数 person に Person クラスの実体を格納しています。本来は "Person person = new Person();" というように 変数名の前はその変数の型を指定しなければいけませんが、C# ではコンパイラが自動的に型を推理してくれる "var" というキーワードがあり、右辺の型が明確な場合に "var" キーワードが使えます。今回の場合、右辺は "new Person()" のように Person クラスが返ってくることが明確になっているため、var で宣言された変数 person の型が自動的に Person クラスになります。

Person クラスにはメンバ変数がひとつ公開されているので、これを使ってみたいと思います。

コード 2.10：Person クラスを使ってみる

Person.cs

```
1   namespace YKWpfIntroduction.ConsoleApplication
2   {
3       using System;
4
5       class Program
6       {
7           static void Main(string[] args)
8           {
9               // Person クラスをインスタンス化
10              var person = new Person();
11
12              Console.WriteLine(person.Statement);
13              person.Statement = "こんにちは。";
14              Console.WriteLine(person.Statement);
15
16              Console.WriteLine("何かキーを押すと終了します。");
17              Console.ReadKey();
18          }
19      }
20  }
```

Person クラスをインスタンス化した後、public フィールドのメンバ変数 Statement を表示し、その後 "こんにちは。" という文字列を与えて再度表示させています。実行結果は図 2.6 のようになります。インスタンス化直後はメンバ変数 Statement は初期状態、つまり null であるため何も表示されていませんが、"こんにちは。" という文字列を与えた後に表示させるとその通りに表示されていることが確認できます。

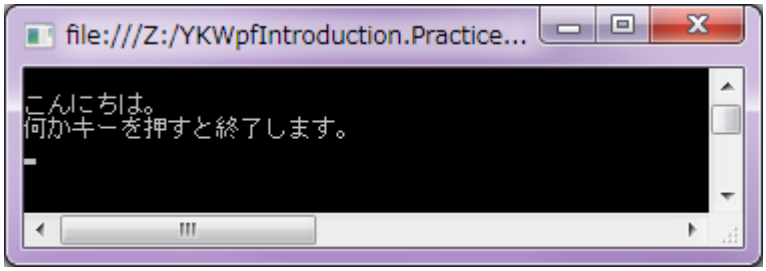


図 2.6：Person クラスのメンバ変数 Statement にアクセスできている

ところで、Statement は Person クラスのメンバ変数として定義されていますが、メンバ変数をそのまま公開することはシステム設計として良くありません。メンバ変数はそのクラスの内部状態を表すもので、それが外部から直接変更されるということは、クラスの中身を外部から直接操作できるようになっているからです。メンバー変数はクラス外部から直接操

作できないようにすべきで、クラスの内部状態の変更はすべてメソッドを通しておこなうべきです。

内部状態の変更をメソッドでおこなうように書き換えた Person クラスは次のようになります。

コード 2.11：メンバ変数を公開しないようにする

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 発言内容
7          /// </summary>
8          private string _statement;
9
10         /// <summary>
11         /// 発言内容を取得します。
12         /// </summary>
13         /// <returns>現在の発言内容を返します。</returns>
14         public string GetStatement()
15         {
16             return this._statement;
17         }
18
19         /// <summary>
20         /// 発言内容を設定します。
21         /// </summary>
22         /// <param name="value">設定する発言内容を指定します。</param>
23         public void SetStatement(string value)
24         {
25             this._statement = value;
26         }
27     }
28 }
```

まず public 修飾子を付けて宣言していた変数 Statement を private 修飾子にして公開しないようにします。また、同時に名前を _statement に変更しています。そして、_statement を取得するための GetStatement() メソッドと、設定するための SetStatement() メソッドを public 修飾子で公開するようにしました。GetStatement() メソッドは _statement を返すだけで、SetStatement() メソッドは _statement に与えられた文字列をそのまま入れるだけなので、これならメンバ変数をそのまま公開したほうが、効率が良いように思われます。しかし、例えば設定する発言内容に文字数制限などがあったとき、SetStatement() メソッドの中に処理を追加することで制限をかけることができます。また、今は何も言いたくない状態のとき、GetStatement() メソッドで発言内容を返さないようにすることもできます。このように、取得と設定をメソッドにすることで、クラスの内部状態によって処理を追加することができるようになります。

しかし、たったひとつのメンバ変数に対して一々 Get○○() メソッドと Set○○() メソッドを用意するのはとても手間がかかり、コーディング作業も大変になってしまいます。また、このクラスを使用する側も、あるメンバ変数を操作するために一々 Get○○() メソッドや Set○○() メソッドを呼び出す必要があり、非効率的です。そこで C# の特徴のひとつであるプロパティを使用します。プロパティを使用したコードは次のようになります。

コード 2.12：Statement プロパティの定義

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          private string _statement;
6          /// <summary>
7          /// 発言内容を取得または設定します。
8          /// </summary>
9          /// <returns>現在の発言内容を返します。</returns>
10         public string Statement
```

```
11          {
12              get { return this._statement; }
13              set { this._statement = value; }
14          }
15      }
16  }
```

プロパティを定義するときは、10 行目のようにまるで変数のように宣言しますが、その後 "{"、"}" で get アクセサと set アクセサを括るように記述します。get アクセサはコード 2.11 で定義した Get○○() メソッドの役割を、set アクセサは Set○○() メソッドの役割を担います。Statement プロパティを使ってもう一度 Main() メソッドを書きなおしてみましょう。

コード 2.13：Person クラスの Statement プロパティを使ってみる

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              // Person クラスをインスタンス化
10             var person = new Person();
11
12             Console.WriteLine(person.Statement);
13             person.Statement = "こんにちは。";
14             Console.WriteLine(person.Statement);
15
16             Console.WriteLine("何かキーを押すと終了します。");
17             Console.ReadKey();
18         }
19     }
20 }
```

実はコード 2.10 とまったく同じコードです。ただし、person.Statement はプロパティとなっているため、13 行目で値を代入するときは Statement プロパティの set アクセサが処理され、14 行目で値を取得するときは get アクセサが処理されることになります。このように、プロパティとはクラス内部から見るとメソッドのように振る舞い、クラス外部から見るとメンバ変数のように振る舞うものであることがわかります。

ワンポイント 2.2：プロパティの自動実装

コード 2.12 では Statement プロパティの実体であるメンバ変数 _statement を明示的に定義していますが、プロパティの get アクセサおよび set アクセサがただ単に値を返し、値を代入するだけの場合、次のようにコードを省略することができます。

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 発言内容を取得または設定します。
7          /// </summary>
8          /// <returns>現在の発言内容を返します。</returns>
9          public string Statement { get; set; }
10     }
11 }
```

2.2.2 コンストラクタ

コード 2.13 などでは、Person クラスをインスタンス化するために new 演算子を使って "new Person()" という記述をしました。このクラス名と同じメソッドのようなもの Person() をコンストラクタと呼びます。Person クラスの定義には Person() というコンストラクタを明示的に定義していませんが、クラスを定義するとこのコンストラクタは自動的に実装されます。もちろん明示的に定義することもできます。次のコードでは、コンストラクタを明示的に定義し、その中で Statement プロパティを初期化しています。

コード 2.14：コンストラクタを明示的に定義する

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          public Person()
9          {
10             this.Statement = "どうも。";
11         }
12
13         /// <summary>
14         /// 発言内容を取得または設定します。
15         /// </summary>
16         public string Statement { get; set; }
17     }
18 }
```

コンストラクタは他のクラスから new 演算子によって呼び出されるメソッドなので、public 修飾子を付けて定義します。ただし、コンストラクタは特殊なメソッドなので戻り値に対する型を表す void や int といったものは必要ありません。上記の例では、コンストラクタが呼び出されると同時に Statement プロパティを "どうも。" という文字列で初期化しています。このように変更してからコード 2.10 を実行すると次のように出力されます。

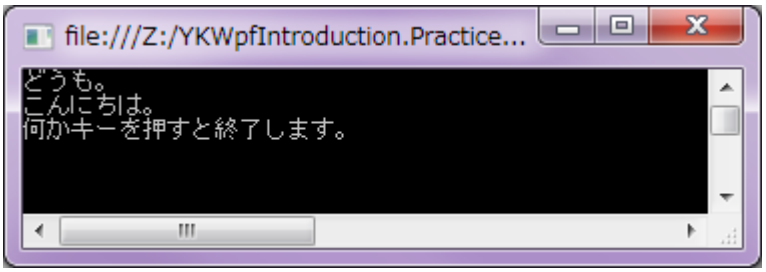


図 2.7：Statement プロパティが初期化されている

前節では Statement プロパティが初期化されていなかったため、最初の行は空欄でしたが、今回は "どうも。" という文字列で初期化されたため、Statement プロパティに値を入れる前にコンソールに表示すると初期値が入っていることが確認できます。

ワンポイント 2.3：private や static なコンストラクタ

public なコンストラクタが定義されていれば外部から new 演算子によってインスタンス化できますが、もし private なコンストラクタが定義されている場合は外部でインスタンス化することができなくなります。これは二重にインスタンスを持つことを防ぐためのシングルトンクラスで使用されます。static なコンストラクタは new 演算子によってインスタンス化される前でも、そのクラスにアクセスされた時点で処理されるものです。

public なコンストラクタと private なコンストラクタはどちらか一方しか定義できませんが、static なコンストラクタはいつでも定義できます。

コンストラクタには入力引数を指定させることもできます。例えば次のコードでは必ず Statement プロパティに対する初期値をコンストラクタの入力引数として指定させています。

コード 2.15：入力引数を持つコンストラクタを定義する

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          /// <param name="statement">発言内容の初期値を指定します。</param>
9          public Person(string statement)
10         {
11             this.Statement = statement;
12         }
13
14         /// <summary>
15         /// 発言内容を取得または設定します。
16         /// </summary>
17         public string Statement { get; set; }
18     }
19 }
```

このとき、コンストラクタを明示的に実装しているため、引数なしのコンストラクタは自動的に実装されなくなります。したがって、このままではコード 2.10 のように "new Person()" でインスタンス化することができなくなります。このような場合、例えば次のように修正することでインスタンス化できるようになります。

コード 2.16：コンストラクタに入力引数を与えてインスタンス化する

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              // Person クラスをインスタンス化
10             var person = new Person("どうもどうも。");
11
12             Console.WriteLine(person.Statement);
13             person.Statement = "こんにちは。";
14             Console.WriteLine(person.Statement);
15
16             Console.WriteLine("何かキーを押すと終了します。");
17             Console.ReadKey();
18         }
19     }
20 }
```

これを実行すると、確かに入力引数に与えた文字列で Statement プロパティが初期化されていることが確認できます。

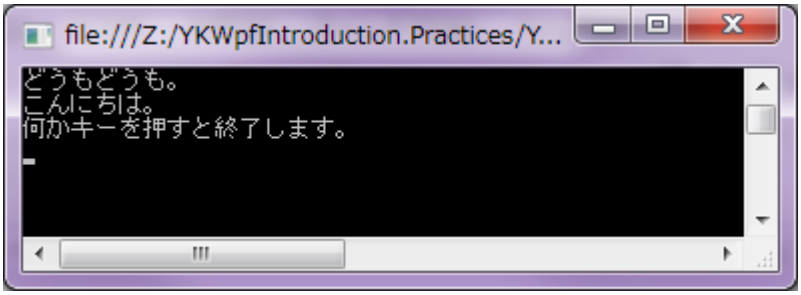


図 2.8：Statement プロパティが指定した文字列で初期化されている

関数オーバーロードによって、引数なしのコンストラクタも同時に定義することもできます。

コード 2.17：コンストラクタを複数定義する

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          public Person()
9          {
10             this.Statement = "おはよう。";
11         }
12
13         /// <summary>
14         /// 新しいインスタンスを生成します。
15         /// </summary>
16         /// <param name="statement">発言内容の初期値を指定します。</param>
17         public Person(string statement)
18         {
19             this.Statement = statement;
20         }
21
22         /// <summary>
23         /// 発言内容を取得または設定します。
24         /// </summary>
25         public string Statement { get; set; }
26     }
27 }
```

上記では引数なしの場合は固定の文字列で Statement プロパティを初期化しようとしています。どちらのコンストラクタも Statement プロパティを初期化するという点では共通の処理をおこなっているため、引数なしのコンストラクタを次のように記述することもできます。

コード 2.18：複数のコンストラクタで同じ処理をおこなう

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          public Person()
9              : this("おはよう。")
10         {
11         }
```

```
12
13          /// <summary>
14          /// 新しいインスタンスを生成します。
15          /// </summary>
16          /// <param name="statement">発言内容の初期値を指定します。</param>
17          public Person(string statement)
18          {
19              this.Statement = statement;
20          }
21
22          /// <summary>
23          /// 発言内容を取得または設定します。
24          /// </summary>
25          public string Statement { get; set; }
26      }
27  }
```

コンストラクタの定義の直後に ":" で区切り、"this()" と書くことで、別のコンストラクタの処理をおこなわせることができます。ここでは "this("おはよう。")" というように string 型を与えているため、string 型をひとつ持つコンストラクタを実行することになります。

このように複数のコンストラクタを定義することで初期化処理を一ヶ所にまとめて書くことができます。また、コード 2.13 でもコード 2.16 でも動作できるようになります。

2.2.3 派生クラスと仮想メソッド

派生クラスを説明するために、Person クラスを次のように変更します。

コード 2.19：基本クラスとなる Person クラス

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          public Person()
9          {
10             System.Console.WriteLine("Person クラスのコンストラクタが処理されました。");
11         }
12
13         /// <summary>
14         /// 発言内容を取得します。
15         /// </summary>
16         public string Statement { get; private set; }
17
18         /// <summary>
19         /// 独り言をつぶやきます。
20         /// </summary>
21         /// <param name="statement">つぶやく独り言を指定します。</param>
22         public void SpeakingToMyself(string statement)
23         {
24             this.Statement = statement;
25         }
26     }
27 }
```

Statement プロパティは public 修飾子が付いているので外部からアクセスできますが、set アクセサに private 修飾子が付いているため、外部からは取得のみできる読取専用プロパティとなっています。

SpeakingToMyself() メソッドは public 修飾子が付いているので外部からアクセスできるメソッドで、与えられた文字列を Statement プロパティに設定するようになっています。

この Person クラスに対して、PersonX クラスを新たに定義します。別のクラスを定義するため、ソリューションエクスプローラー上で PersonX.cs ファイルを追加するようにしましょう。

コード 2.20：Person クラスから派生する PersonX クラス

PersonX.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class PersonX : Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          public PersonX()
9          {
10             System.Console.WriteLine("PersonX クラスのコンストラクタが処理されました。");
11         }
12     }
13 }
```

基本クラスを指定するときはクラス名に続いて ":" の後に基本クラス名を記述します。PersonX クラスは Person クラスから派生したクラスです。Person クラスから派生しているため、PersonX クラスは Person クラスと同じ機能を持っています。つまり、Person クラスが Statement プロパティを持っているように、PersonX クラスも Statement プロパ

ティを持っています。
この Person クラスと PersonX クラスを使って Main() メソッドを編集しましょう。

コード 2.21：Person クラスと PersonX クラスに対して同じ処理をおこなう

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              var statement = "おなかすいた。";
10
11             // Person クラスをインスタンス化
12             var person = new Person();
13             person.SpeakingToMyself(statement);
14             Console.WriteLine(person.Statement);
15
16             // PersonX クラスをインスタンス化
17             var personX = new PersonX();
18             personX.SpeakingToMyself(statement);
19             Console.WriteLine(personX.Statement);
20
21             Console.WriteLine("何かキーを押すと終了します。");
22             Console.ReadKey();
23         }
24     }
25 }
```

Person クラスも PersonX クラスも同じようにインスタンス化し、SpeakingToMyself() メソッドに同じ文字列を与え、Statement プロパティを出力するようにしています。実行結果は次のようになります。

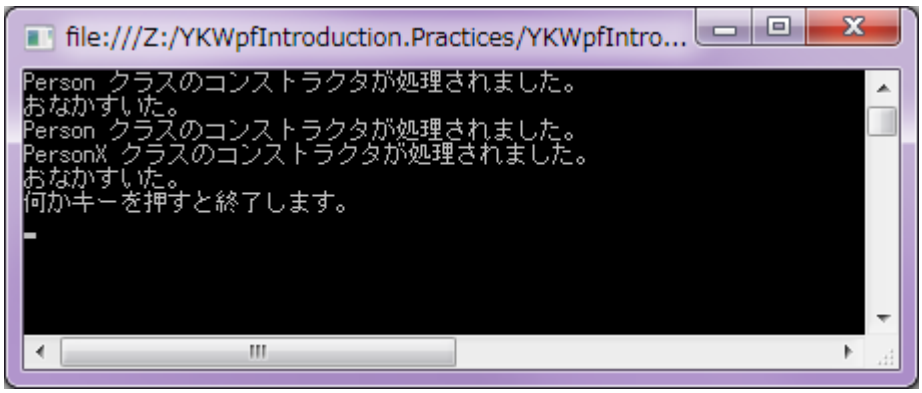


図 2.9：Person クラスと PersonX クラスを使った実行例

Person クラスをインスタンス化するとそのコンストラクタが処理されます。そして、事前に与えられた文字列によって Statement プロパティが設定されているため、その文字列がそのまま出力されています。PersonX クラスは、インスタンス化するときにまず基本クラスである Person クラスのコンストラクタが処理されます。その次に PersonX クラスとしてのコンストラクタが処理されています。その後は Person クラスと同様です。

ワンポイント 2.4：コンストラクタで初期化する場合の注意点

コンストラクタ内でプロパティなどの初期化をする場合、そのクラスが基本クラスなのか派生クラスなのかを意識する必要があります。派生クラスの場合、まず基本クラスのコンストラクタが処理されるため、基本クラスが持っているプロパティなどが既に初期化されている場合があります。そのプロパティが一度インスタンス化したクラスであったりすることもありますし、コンストラクタ内でイベント購読をおこなう場合、既に購読しているイベントを二重に購読してしまうことも起こり得ます。

次に、Person クラスで定義した SpeakingToMySelf() メソッドの定義を少し変えましょう。

コード 2.22：SpeakingToMySelf() メソッドを仮想メソッドにする

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          public Person()
9          {
10             System.Console.WriteLine("Person クラスのコンストラクタが処理されました。");
11         }
12
13         /// <summary>
14         /// 発言内容を取得します。
15         /// </summary>
16         public string Statement { get; protected set; }
17
18         /// <summary>
19         /// 独り言をつぶやきます。
20         /// </summary>
21         /// <param name="statement">つぶやく独り言を指定します。</param>
22         public virtual void SpeakingToMyself(string statement)
23         {
24             this.Statement = statement;
25         }
26     }
27 }
```

16 行目の Statement プロパティの set アクセサを private ではなく protected としています。また、22 行目では virtual 修飾子を付けることで SpeakingToMyself() メソッドを仮想メソッドとしています。仮想メソッドとは、派生クラスによってその処理内容を変更することができるメソッドのことです。この機能を確認するために、PersonX クラスで SpeakingToMyself() メソッドを書き換えてみましょう。

コード 2.23：SpeakingToMySelf() メソッドをオーバーライドする

PersonX.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      internal class PersonX : Person
4      {
5          /// <summary>
6          /// 新しいインスタンスを生成します。
7          /// </summary>
8          public PersonX()
9          {
10             System.Console.WriteLine("PersonX クラスのコンストラクタが処理されました。");
11         }
12
13         /// <summary>
14         /// 口癖を含む独り言をつぶやきます。
15         /// </summary>
16         /// <param name="statement">つぶやく独り言を指定します。</param>
17         public override void SpeakingToMyself(string statement)
18         {
19             //base.SpeakingToMyself(statement);
20             this.Statement = statement + "別にいいけど。";
21         }
```

```
22      }
23  }
```

仮想メソッドを書き換えることをオーバーライドといい、override 修飾子を付けることで実装できます。オーバーライドされたメソッドは、書き換えられる前の処理もおこなうことができます。上記の例では 19 行目にあるコードで、"base.SpeakingToMyself(statement);" のように基本クラス base を使用することで元の処理を呼び出すことができます。ここでは元の処理をおこなう必要がないため、この行をコメントアウトしています。

このように仮想メソッドに変更した状態でコード 2.21 のプログラムを実行してみましょう。実行結果は次のようになります。

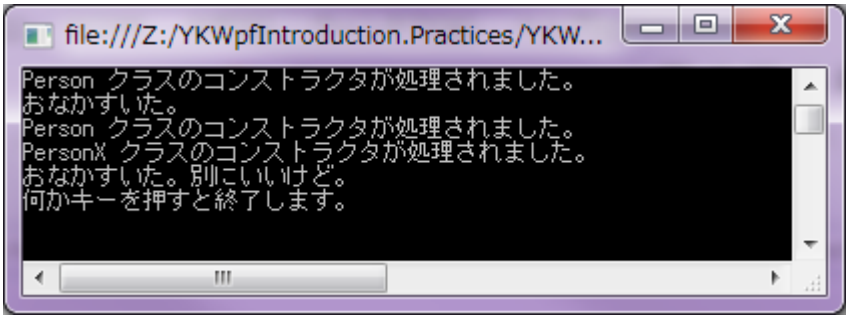


図 2.10：仮想メソッドがオーバーライドされて処理が変わっている

Main() メソッドからは同じように SpeakingToMySelf() メソッドを呼び出しているだけですが、内部実装が異なるため、出力結果が変わっていることがわかります。

ワンポイント 2.5：仮想メソッドによる多態性の実現

多態性とは、複数のインスタンスを同一のメソッドで利用できるようにすることです。例えば図形を描画する Figure クラスがあるとします。図形を描画するには Figure クラスが持つ Draw() メソッドを呼び出します。ここで、Figure クラスを基本クラスとした Rectangle クラスや Triangle クラスがあったとします。これらのクラスも同じく、図形を描画するときは Draw() メソッドを呼び出しますが、それぞれ四角形を描画したり三角形を描画したりしなければなりません。このとき、Draw() メソッドが仮想メソッドとして定義されていれば、派生クラスである Rectangle クラスや Triangle クラスが Draw() メソッドをオーバーライドすることで実現できます。

これらのクラスを使う側としては、同じメソッドを呼ぶだけでそれぞれが異なる描画をしてくれることに利便性を感じるはずです。クラスを作る側としては、このように便利なクラスとなるように仮想メソッドを活用できるようにしましょう。

2.3 for 文と foreach 文

C/C++ では、特に配列の各要素に対する処理を書くときに、繰り返し処理として for 文を用いて記述することがよくあります。C# でも同様に for 文によって繰り返し処理を記述することができます。例えば配列の各要素を用いた処理は Array.Length プロパティを用いて次のように書きます。

コード 2.24：for 文による配列へのアクセス

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Collections.Generic;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var values = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
11             for (var i = 0; i < values.Length; i++)
12             {
13                 Console.WriteLine(values[i]);
14             }
15         }
16     }
17 }
```

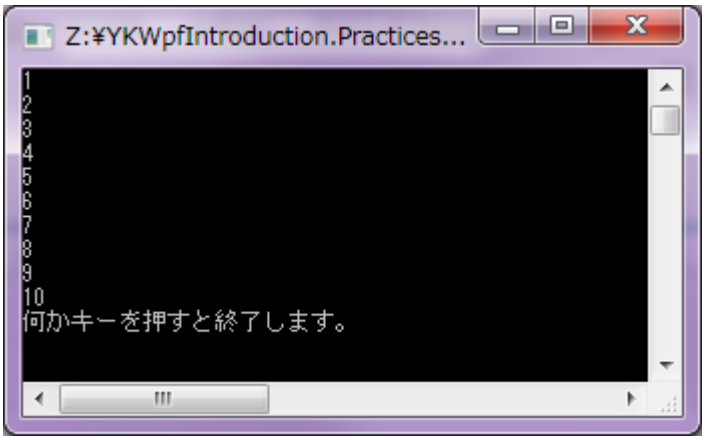


図 2.11：for 文による処理結果

C# では配列は Array クラスで表され、その要素数は Length プロパティによって知ることができます。したがって、for 文で配列の各要素を用いた処理を記述する場合は、インデックス用の変数の上限として Length プロパティを使う方法が一般的です。

しかし、C# ではオブジェクトの集合を表現するための方法は配列だけではなく、例えば IEnumerable<T> インターフェースによるコレクションとして表現する場合があります。この場合、コレクションの各要素を用いた処理をおこなうには for 文ではなく、foreach 文を用いて各要素を抽出しながら処理をおこないます。このときのコードは次のようになります。

コード 2.25：foreach 文による配列へのアクセス

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Collections.Generic;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
```

```
10              var values = SampleObjects();
11              foreach (var value in values)
12              {
13                  Console.WriteLine(value);
14              }
15
16              Console.WriteLine("何かキーを押すと終了します。");
17              Console.ReadKey();
            }

            /// <summary>
            /// あるコレクションを返すメソッドです。
            /// </summary>
            /// <returns>サンプル用のコレクションを返します。</returns>
            private static IEnumerable<int> SampleObjects()
            {
                yield return 1;
                yield return 2;
                yield return 3;
                yield return 4;
                yield return 5;
                yield return 6;
                yield return 7;
                yield return 8;
                yield return 9;
                yield return 10;
            }
        }
    }
```

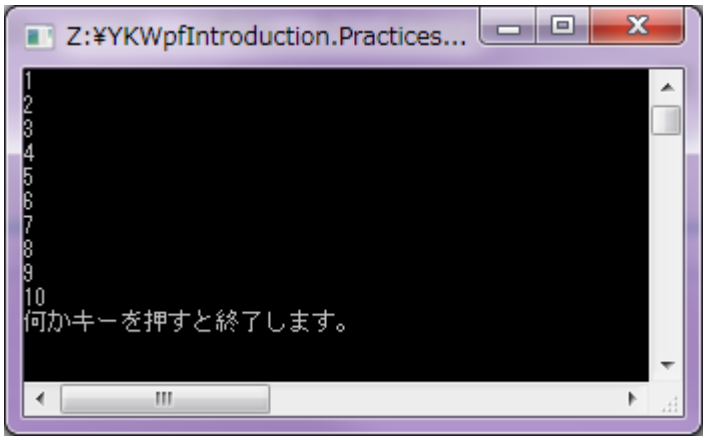


図 2.12：foreach 文による処理結果

foreach 文の場合、コレクションの各要素を順番に抽出するコードが内部実装されているため、開発者がコレクションの要素数を気にすることなくコードを記述することができます。foreach 文は IEnumerable インターフェースを介した要素へのアクセスを簡単化するための構文なので、IEnumerable インターフェースを実装したクラスによるコレクションであればなんでも使用することができます。もちろん配列もコレクションの一種であるため、配列も同様に foreach 文で処理することができます。foreach 文を用いることで、配列の要素数を超えたバッファオーバーランによる例外発生を防止することもできるため、C# でコレクションの各要素を用いた処理を記述するときは必ず foreach 文で記述したほうが良いでしょう。ただし、foreach 文を用いた処理では、そのコレクション要素そのものを書き換えることはできません。上記のサンプルでは、例えば 13 行目で "value = 0;" というように、foreach 文で操作中のコレクション要素を変更することは禁止されています。そういった操作をおこなうときはやはり for 文を使わないといけません。コレクションの各要素を用いた処理をおこなうときは foreach 文、コレクションの各要素を書き換えるときは for 文または別の実装方法にするように使い分けるようにしましょう。

ところで、foreach 文はコレクションへのアクセスを簡単におこなうことができますが、コレクションの種類によっては、今操作している要素はコレクションに対する何番目の要素なのかを知ることが直接できない場合があります。このような場合、System.Linq.Enumerable クラスによる拡張メソッドを使用することで、次のように書くことができます。

コード 2.26：foreach 文で何番目の要素なのかを知るサンプルコード

Program.cs

```
 1  namespace YKWpfIntroduction.ConsoleApplication
 2  {
 3      using System;
 4      using System.Collections.Generic;
 5      using System.Linq;
 6
 7      class Program
 8      {
 9          static void Main(string[] args)
10          {
11              var values = SampleObjects();
12              foreach (var value in values.Select((x, i) => new { x, i }))
13              {
14                  // 匿名クラスから値とインデックスを取り出す
15                  Console.WriteLine("{0} 番目は {1} です。", value.i, value.x);
16              }
17
18              Console.WriteLine("何かキーを押すと終了します。");
19              Console.ReadKey();
20          }
21
22          /// <summary>
23          /// あるコレクションを返すメソッドです。
24          /// </summary>
25          /// <returns>サンプル用のコレクションを返します。</returns>
26          private static IEnumerable<int> SampleObjects()
27          {
28              yield return 1;
29              yield return 2;
30              yield return 3;
31              yield return 4;
32              yield return 5;
33              yield return 6;
34              yield return 7;
35              yield return 8;
36              yield return 9;
37              yield return 10;
38          }
39      }
40  }
```

Enumerable.Select() 拡張メソッドはコレクションの各要素を射影するためのメソッドで、要素と要素に対するインデックス番号を組にしたものを改めてコレクションとしています。これによって foreach 文の内側では今何番目の要素なのかを匿名クラスのメンバ変数 i で知ることができるようになっています。

2.4　デリゲートによる処理の委譲

デリゲートとは、処理を別の誰かに委譲することを意味します。つまり、他のオブジェクトにメソッドを参照させるための仕組みです。ここで紹介するデリゲートは C#2.0 の古い書き方で、実際にこのようなコードを使ってデリゲートを実装することはないと思います。あくまでもデリゲートの基礎を知るということを目的としています。

それでは、サンプルコードを見てみましょう。

コード 2.27：デリゲートによる処理の委譲

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
7          /// <summary>
8          /// 入力引数なし、戻り値なしのデリゲート
9          /// </summary>
10         delegate void SomeDelegate1();
11
12         /// <summary>
13         /// int 型の入力引数、int 型の戻り値のデリゲート
14         /// </summary>
15         /// <param name="i"></param>
16         delegate int SomeDelegate2(int i);
17
18         static void Main(string[] args)
19         {
20             SomeDelegate1 d1 = Func1;
21             d1();
22
23             SomeDelegate2 d2 = Func2;
24             var d2_return = d2(3);
25             Console.WriteLine(d2_return + "が返ってきました。");
26
27             Console.WriteLine("何かキーを押すと終了します。");
28             Console.ReadKey();
29         }
30
31         static void Func1()
32         {
33             Console.WriteLine("Func1() が呼ばれました。");
34         }
35
36         static int Func2(int x)
37         {
38             Console.WriteLine("Func2({0}) が呼ばれました。", x);
39
40             return 2 * x;
41         }
42     }
43 }
```

delegate というキーワードを用いて宣言されたメソッドはデリゲートによって他のメソッドを参照するためのオブジェクトになります。SomeDelegate1 と名付けたデリゲートは入力引数なし、戻り値なしのメソッドを参照するためのデリゲートで、SomeDelegate2 と名付けたデリゲートは int 型の入力引数を持ち、int 型の戻り値を持つメソッドを参照するためのデリゲートとなります。

デリゲートに対して参照したいメソッドを指定するときは 20 行目や 23 行目のように宣言時に "=" でメソッド名を指定します。すると、宣言したデリゲート変数 d1 や d2 をメソッドのように使うことで指定されたメソッドが実行されることになります。

このサンプルコードの実行結果は次のようになります。

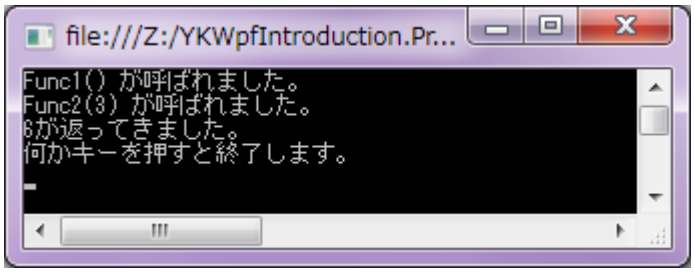


図 2.13：それぞれのデリゲートに指定されたメソッドが実行されている

デリゲートは複数のメソッドを参照することもできます。サンプルコードを見てみましょう。

コード 2.28：マルチキャストデリゲートによる処理の委譲

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
7          /// <summary>
8          /// 入力引数なし、戻り値なしのデリゲート
9          /// </summary>
10         delegate void SomeDelegate1();
11
12         static void Main(string[] args)
13         {
14             SomeDelegate1 d1 = Func1;
15             d1 += Func2;
16             d1 += Func3;
17
18             d1();
19
20             Console.WriteLine("参照を外すこともできます。");
21             d1 -= Func2;
22
23             d1();
24
25             Console.WriteLine("何かキーを押すと終了します。");
26             Console.ReadKey();
27         }
28
29         static void Func1()
30         {
31             Console.WriteLine("Func1() が呼ばれました。");
32         }
33
34         static void Func2()
35         {
36             Console.WriteLine("Func2() が呼ばれました。");
37         }
38
39         static void Func3()
40         {
41             Console.WriteLine("Func3() が呼ばれました。");
42         }
43     }
44 }
```

デリゲート変数 d1 には始めに Func1() メソッドを参照させるようにしています。そして 15、16 行目で Func2() メソッドと Func3() メソッドへの参照を "+=" 演算子で追加しています。実行結果は次のようになります。

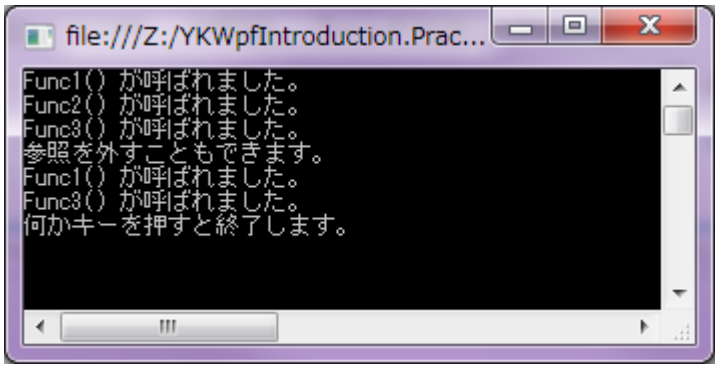


図 2.14：複数のメソッドを参照して順番に実行されている

18 行目で d1() を実行すると、デリゲート変数 d1 に登録した順に参照されているメソッドが実行されます。また、21 行目で、"-=" 演算子で Func2() メソッドへの参照を解除しているため、23 行目で d1() を実行すると Func2() メソッドが実行されなくなっていることが確認できます。

このようなデリゲートの機能をマルチキャストデリゲートと呼び、イベントハンドラの機能として応用されています。

2.5 イベント

前節までのデリゲートを利用することでイベントを作成することができます。サンプルコードを見てみましょう。まずイベントを持つ EventTest クラスを定義します。

コード 2.29：イベントを持つクラスのサンプルコード

EventTest.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class EventTest
6      {
7          private string _name;
8          /// <summary>
9          /// 名前を取得または設定します。
10         /// </summary>
11         public string Name
12         {
13             get { return this._name; }
14             set
15             {
16                 if (this._name != value)
17                 {
18                     this._name = value;
19                     Console.WriteLine("名前が変更されました。");
20                     RaiseNameChanged();
21                 }
22             }
23         }
24
25         /// <summary>
26         /// 名前が変更されたときに発生します。
27         /// </summary>
28         public event EventHandler NameChanged;
29
30         /// <summary>
31         /// NameChanged イベントを発行します。
32         /// </summary>
33         private void RaiseNameChanged()
34         {
35             var h = this.NameChanged;
36             if (h != null)
37             {
38                 Console.WriteLine("イベントを発生させます。");
39                 h(this, EventArgs.Empty);
40             }
41         }
42     }
43 }
```

28 行目に event キーワードを付けた変数 NameChanged がイベントになります。EventHandler クラスはデリゲートで、イベントを処理するためのメソッドを参照します。

33 行目に NameChanged イベントを発行するための RaiseNameChanged() メソッドを定義しています。35 行目で NameChanged デリゲートからメソッドへの参照を取得し、39 行目で参照しているメソッドを実行しています。これは前節で説明したマルチキャストデリゲートの仕組みを応用しています。

このイベントを使うサンプルコードは次のようになります。

コード 2.30：イベントにイベントハンドラを登録

Program.cs

```
 1 namespace YKWpfIntroduction.ConsoleApplication
 2 {
 3     using System;
 4
 5     class Program
 6     {
 7         static void Main(string[] args)
 8         {
 9             var test = new EventTest();
10             test.NameChanged += OnNameChanged;
11
12             Console.WriteLine("処理をおこないます。");
13             test.Name = "Mike";
14
15             Console.WriteLine("何かキーを押すと終了します。");
16             Console.ReadKey();
17         }
18
19         /// <summary>
20         /// NameChanged イベントハンドラ
21         /// </summary>
22         /// <param name="sender">イベント発行元</param>
23         /// <param name="e">イベント引数</param>
24         private static void OnNameChanged(object sender, EventArgs e)
25         {
26             Console.WriteLine("NameChanged イベントハンドラが実行されました。");
27         }
28     }
29 }
```

10 行目のようにイベントへメソッドの参照を登録するときは "+=" 演算子を使用します。サンプルコードにはありませんが、メソッドの参照の登録を解除するときは "-=" 演算子を使用します。このサンプルコードの実行結果は次のようになります。

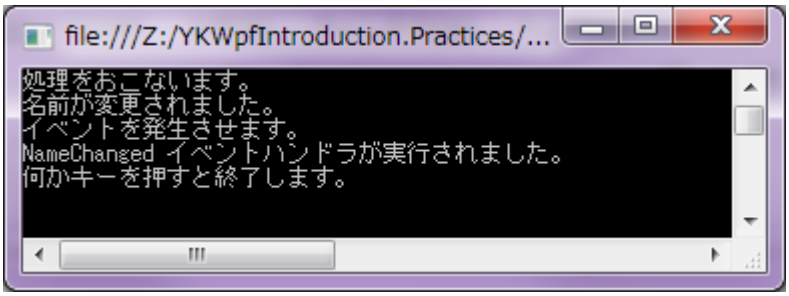


図 2.15：イベントに登録されたメソッドが実行されている

イベントの仕組みは、イベントハンドラがイベントを発行する側から実行されていることに注意しなければいけません。上記のサンプルコードでは、イベントハンドラである OnNameChanged() メソッドは Program クラスが持っていますが、実際に実行するのは EventTest クラスの RaiseNameChanged() メソッドとなります。

ワンポイント 2.6：イベントハンドラとメモリリーク

例えばクラス A がクラス B のイベントに自分のメソッドをイベントハンドラとして登録した状態で、クラス A が誰からも参照されなくなったとき、本来はガベージコレクションによってクラス A のあったメモリ領域は解放される対象となるべきですが、クラス B がクラス A のメソッドを参照したままになっているため、クラス A のあったメモリ領域はいつまでも解放されることなく残り続けてしまいます。このような現象をメモリリークと呼び、イベントハンドラの登録の裏には必ず登録解除がなければなりません。

2.6 Action クラスと Func<TResult> クラスによるデリゲート

前節までで、デリゲートの機能について紹介しましたが、メソッドを参照する方法として Action クラスや Func<TResult> クラスを使う方法もあります。サンプルコードを見てみましょう。

コード 2.31：Action クラスを用いた処理の委譲

Person.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              DoSomething(DoWork);
10
11             Console.WriteLine("何かキーを押すと終了します。");
12             Console.ReadKey();
13         }
14
15         static void DoWork()
16         {
17             Console.WriteLine("DoWork() が実行されました。");
18         }
19
20         static void DoSomething(Action action)
21         {
22             action();
23         }
24     }
25 }
```

DoWork() メソッドで何か処理をおこないますが、Main() メソッドからはこのメソッドを直接呼ばずに、DoSomething() メソッドを呼び出しています。このとき、入力引数として DoWork メソッドを指定しています。実は DoSomething() メソッドの入力引数である Action クラスは、入力引数なし、出力引数なしのメソッドに対するデリゲートを表しています。DoSomething() メソッドとしては入力引数に渡されたデリゲートを実行するだけで、実際にどんな処理がおこなわれるかを把握していません。このようなデリゲートは、イベントハンドラやコールバックなどの仕組みでよく使われています。

これを実行すると次のように、DoWork() メソッドがきちんと呼び出されていることが確認できます。

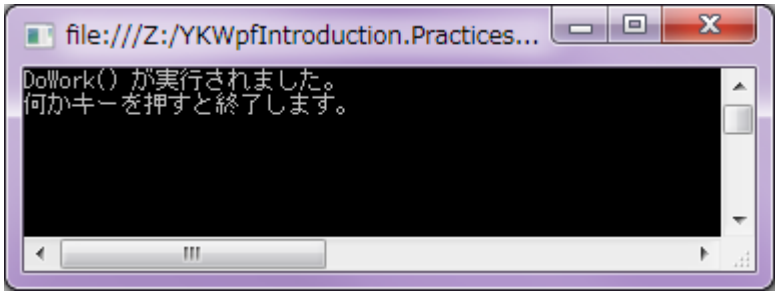


図 2.16：委譲された DoWork() の処理がきちんと実行されている

Action クラスは戻り値なしのデリゲートを表していて、Action<T> クラスは T 型の入力引数を持つメソッドに対するデリゲートを表しています。<T> というのはジェネリックを表していて、どのような型でも指定することができます。つまり、Action<int> クラスは int 型を入力引数に持つデリゲート、Action<string> は string 型を入力引数に持つデリゲートとなります。さらに、Action<Person> クラスは Person クラスを入力引数に持つデリゲートを表すことになります。いずれも Action クラスなので戻り値は持ちません。同様に入力引数を複数持つメソッドを参照するデリゲートとして Action<T1, T2> クラスや Action<T1, T2, T3> クラスが用意されています。

戻り値を持つデリゲートは Func<TResult> クラスで表されます。Func<TResult> クラスは TResult 型の戻り値を持つメソッドに対するデリゲートを表しています。また、Func<T, TResult> クラスは T 型の入力引数を持ち、TResult 型の戻り値を持つメソッドに対するデリゲートを表しています。同様に Func<T1, T2, TResult> クラス、Func<T1, T2, T3, TResult> クラスが用意されています。

2.7　匿名関数とラムダ式

2.7.1　匿名関数

通常、関数を定義するときはクラスのメンバとして定義することになりますが、あるメソッド内だけでしか使わない処理をメソッド化する場合、メソッド内で定義することができる匿名関数を利用すると効率的なコードが書けるようになります。

例えば次のような Count<T>() メソッドを考えてみましょう。

コード 2.32：要素数を数える汎用的なメソッド

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Collections.Generic;
5
6      class Program
7      {
8          /// <summary>
9          /// 指定された判定基準を満たすものの要素数を数えます。
10         /// </summary>
11         /// <typeparam name="T">判定対象とするオブジェクトの型を指定します。</typeparam>
12         /// <param name="enumerator">判定対象とするオブジェクトのコレクションを指定します。</param>
13         /// <param name="predicate">判定処理を指定します。</param>
14         /// <returns>判定基準を満たす要素数を返します。</returns>
15         private static int Count<T>(IEnumerable<T> enumerator, Func<T, bool> predicate)
16         {
17             int count = 0;
18             foreach (T item in enumerator)
19             {
20                 if (predicate(item))
21                     count++;
22             }
23             return count;
24         }
25     }
26 }
```

Count<T>() メソッドは、指定されたコレクションの要素を、指定された比較方法で判定し、判定基準を満たす要素数を返すメソッドです。どういう方法で比較するかは入力引数の predicate デリゲートで処理を委譲し、Count<T>() メソッド内ではとにかく predicate によって参照されているメソッドの結果が true である要素数を数えるだけの処理を記述しています。このように記述することで、「ある値より小さい要素数を数える」だけでなく、「ある値より大きい要素数」を数えたり、もっと複雑な条件を満たす要素数を数えたりする場合にも使える汎用的なメソッドとなっています。

このメソッドを使って、整数 1 ～ 10 の中で、5 より小さいものの数を数えるコードを考えてみましょう。

コード 2.33：デリゲートにメンバ関数を渡す

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Collections.Generic;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var values = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
11
12             // 比較対象とする値を記憶させる
13             _comparedValue = 5;
14             // 比較対象より小さい要素数を数える
15             var count = Count<int>(values, IsLessThanValue);
```

```
16              Console.WriteLine("values の中で判定基準を満たす要素は " + count + " 個あります。");
17
18              Console.WriteLine("何かキーを押すと終了します。");
19              Console.ReadKey();
20          }
21
22          /// <summary>
23          /// 指定された判定基準を満たすものの要素数を数えます。
24          /// </summary>
25          /// <typeparam name="T">判定対象とするオブジェクトの型を指定します。</typeparam>
26          /// <param name="enumerator">判定対象とするオブジェクトのコレクションを指定します。</param>
27          /// <param name="predicate">判定処理を指定します。</param>
28          /// <returns>判定基準を満たす要素数を返します。</returns>
29          private static int Count<T>(IEnumerable<T> enumerator, Func<T, bool> predicate)
30          {
31              int count = 0;
32              foreach (T item in enumerator)
33              {
34                  if (predicate(item))
35                      count++;
36              }
37              return count;
38          }
39
40          /// <summary>
41          /// 比較対象とする値を保持します。
42          /// </summary>
43          private static int _comparedValue;
44
45          /// <summary>
46          /// ある値より小さいかどうかを判定します。
47          /// </summary>
48          /// <param name="x">判定対象の数値を指定します。</param>
49          /// <returns>指定された数値が 5 より小さい場合に true を返します。</returns>
50          private static bool IsLessThanValue(int x)
51          {
52              return x < _comparedValue;
53          }
54      }
55  }
```

「ある値より小さい」ということを判定するために、メンバ関数を定義しています。また、「ある値」を保持する必要があるため、メンバ変数も用意する必要があります。43 行目の変数 _comparedValue で比較対象の値を保持し、50 行目の IsLessThanValue() メソッドで判定処理を定義しています。そして、15 行目で Count<T>() メソッドに対して対象とするコレクション values と、判定処理 IsLessThanValue() メソッドを指定し、その結果を 16 行目で表示するようにしています。

このサンプルコードの実行結果は次のようになります。サンプルでは 5 より小さい値を数えているので、1 ～ 4 の 4 個で、確かにカウントできています。

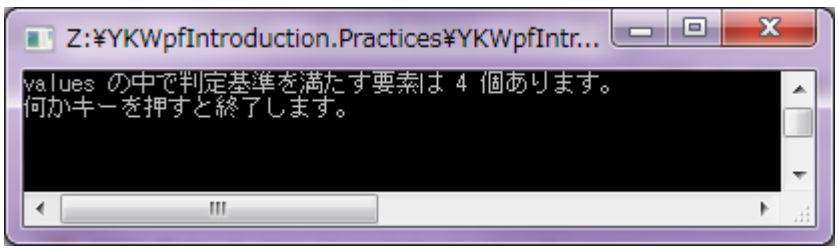


図 2.17：比較判定処理によって要素数を数えている

しかし、デリゲートにメソッドを渡すためだけにメンバ関数およびメンバ変数を定義するのはあまり効率が良いとはいえません。また、定義したメンバ関数やメンバ変数がここでしか使わないものだとしたら可読性も良くありません。そこで、

匿名関数という機能を利用することで、上記のサンプルコードは次のように書き換えることができます。

コード 2.34：匿名関数を利用したコード

Program.cs

```
 1  namespace YKWpfIntroduction.ConsoleApplication
 2  {
 3      using System;
 4      using System.Collections.Generic;
 5
 6      class Program
 7      {
 8          static void Main(string[] args)
 9          {
10              var values = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
11
12              // 比較対象とする値を記憶させる
13              var comparedValue = 5;
14              // 匿名関数による比較判定基準
15              Func<int, bool> predicate = delegate (int x)
16              {
17                  // Main() メソッドのスコープにある変数も使える
18                  return x < comparedValue;
19              };
20              // 比較対象より小さい要素数を数える
21              var count = Count<int>(values, predicate);
22              Console.WriteLine("values の中で判定基準を満たす要素は " + count + " 個あります。");
23
24              Console.WriteLine("何かキーを押すと終了します。");
25              Console.ReadKey();
26          }
27
28          /// <summary>
29          /// 指定された判定基準を満たすものの要素数を数えます。
30          /// </summary>
31          /// <typeparam name="T">判定対象とするオブジェクトの型を指定します。</typeparam>
32          /// <param name="enumerator">判定対象とするオブジェクトのコレクションを指定します。</param>
33          /// <param name="predicate">判定処理を指定します。</param>
34          /// <returns>判定基準を満たす要素数を返します。</returns>
35          private static int Count<T>(IEnumerable<T> enumerator, Func<T, bool> predicate)
36          {
37              int count = 0;
38              foreach (T item in enumerator)
39              {
40                  if (predicate(item))
41                      count++;
42              }
43              return count;
44          }
45      }
46  }
```

判定処理をメンバ関数としてわざわざ定義する必要はなく、delegate による匿名関数を宣言的に記述することで非常に簡潔なコードとなっています。また、メソッド内で新たな処理を追加しているため、そのスコープ上にある変数も使えるということから、不要なメンバ変数の定義もなくなります。

このように、匿名関数を使うことは非常に便利ですが、実際のコードで delegate 修飾子を用いた匿名関数の定義をおこなうことはあまりありません。C# 3.0 以降では匿名関数を定義するときにラムダ式を用いるからです。

2.7.2 ラムダ式による匿名関数の定義

ラムダ式とは、匿名関数のデリゲートを作成したり、式木を作成したりするための C# 3.0 以降に実装された機能です。式木は LINQ to SQL などで利用されていますが、応用的な技術となるためここでは説明を割愛します。ラムダ式はデリゲートを使用するときに多用するため、使いこなせるようになりましょう。

前節では、値を比較するための判定処理を匿名関数で表現していましたが、これをラムダ式で書いてみましょう。コード 2.34 の 15 ～ 19 行目をラムダ式で記述すると次のようになります。

コード 2.35：ラムダ式による匿名関数のデリゲート作成

Program.cs

```
14              // ラムダ式による比較判定基準の匿名関数へのデリゲート作成
15              Func<int, bool> predicate = x =>
16              {
17                  // Main() メソッドのスコープにある変数も使える
18                  return x < comparedValue;
19              };
```

delegate 修飾子を使う代わりに "=>" という記号で匿名関数の開始を表しています。これだけだと delegate 修飾子を使う場合とあまり変わりませんが、ラムダ式ではさらに次のように省略することができます。

コード 2.36：ラムダ式による匿名関数のデリゲート作成の省略記法

Program.cs

```
14              // ラムダ式による比較判定基準の匿名関数へのデリゲート作成
15              Func<int, bool> predicate = x => x < comparedValue;
```

匿名関数内の処理が戻り値を返すための 1 行のみのコードの場合、このように return も省略したごくシンプルな書き方で完結できるようになります。

入力引数が必要ない場合は "() =>" としてラムダ式を開始します。例えば次のようになります。

コード 2.37：入力引数がない場合のラムダ式の記法

Program.cs

```
1               Func<string> getString = () =>
2               {
3                   return "戻り値";
4               };
```

また、入力引数がある匿名関数を指定するときに、入力引数を特に使わないとき、入力引数をアンダースコア "_" で定義することがよくあります。

コード 2.38：入力引数を使わないときは "_" で定義することが多い

Program.cs

```
1               Func<object, string> getString = _ =>
2               {
3                   return "戻り値";
4               };
```

2.8 System.Linq による拡張メソッド

System.Linq 名前空間を使うことで、コレクションの操作が格段に簡単になります。C# といえば System.Linq というほど System.Linq はなくてはならない存在となっています。この機能は必ず修得しましょう。

2.8.1 Range() メソッドで連続する整数を出力する

Enumerable.Range() メソッドは連続する整数を出力するメソッドです。

コード 2.39：Range() メソッドで連続する整数のコレクションを出力する

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Linq;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var numbers = Enumerable.Range(0, 10);
11             foreach (var number in numbers)
12             {
13                 Console.WriteLine(number);
14             }
15
16             Console.WriteLine("何かキーを押すと終了します。");
17             Console.ReadKey();
18         }
19     }
20 }
```

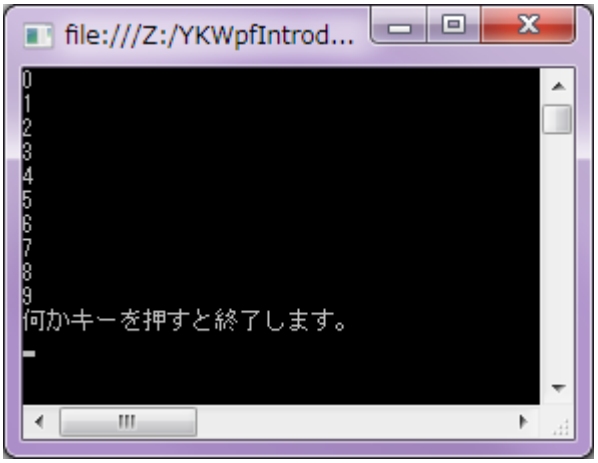


図 2.18：整数のコレクションが得られている

Range() メソッドの第 1 引数に開始する数値、第 2 引数に出力するデータ数を指定します。

2.8.2 Select() メソッドで射影する

Enumerable.Select() メソッドはコレクションの各要素を使って別のコレクションに射影することができます。

コード 2.40：Range() メソッドで連続する整数のコレクションを出力する

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Linq;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var numbers = Enumerable.Range(-5, 11);
11             foreach (var number in numbers)
12             {
13                 Console.WriteLine(number);
14             }
15
16             Console.WriteLine("Select() メソッドで射影します。");
17             var absolutes = numbers.Select(x => Math.Abs(x));
18             foreach (var number in absolutes)
19             {
20                 Console.WriteLine(number);
21             }
22
23             Console.WriteLine("何かキーを押すと終了します。");
24             Console.ReadKey();
25         }
26     }
27 }
```

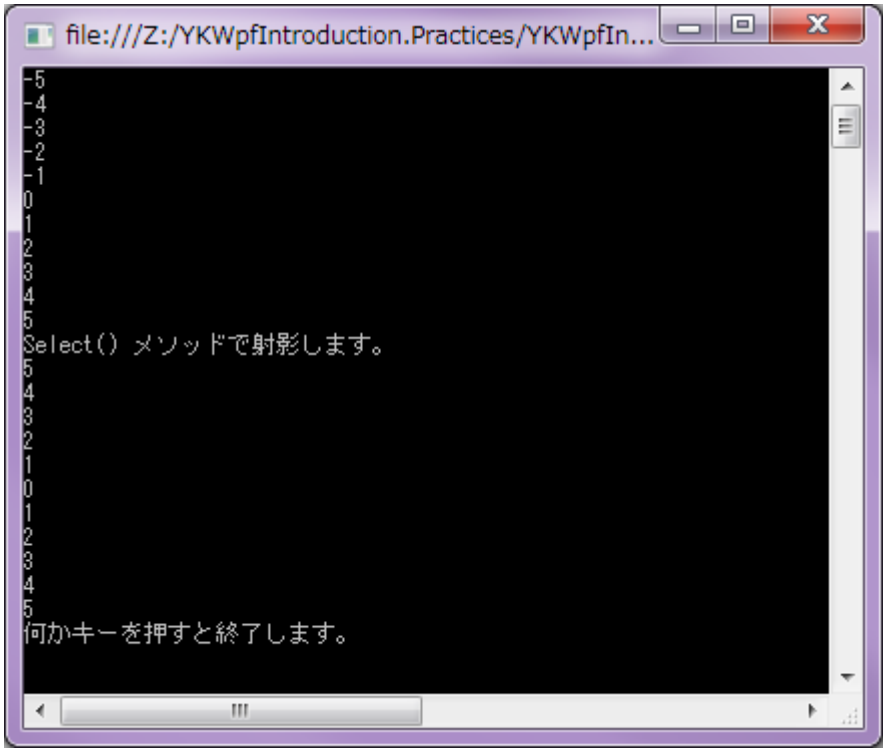


図 2.19：負の値を含む整数が絶対値を取ったコレクションに変更されている

Select() メソッドの入力引数にはコレクションの要素を入力引数、それを変換して新たなコレクションの要素とする戻

り値を持つ Func<object, object> 型のデリゲートを指定します。ここではラムダ式による匿名関数のデリゲートを渡しています。"x => Math.Abs(x)" と記述されているので、コレクションの要素 x を使ってその絶対値を新たなコレクションの要素とするように指定しています。

また、Select() メソッドの入力引数に Func<object, int, object> 型のデリゲートを指定することもできます。このとき、入力引数の第 2 引数はその要素のインデックス番号になります。

コード 2.41：Select() メソッドの中で各要素のインデックスも取得できる

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Linq;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var numbers = Enumerable.Range(-5, 11);
11             foreach (var number in numbers)
12             {
13                 Console.WriteLine(number);
14             }
15
16             Console.WriteLine("Select() メソッドで射影します。");
17             var absolutes = numbers.Select((x, i) => i);
18             foreach (var number in absolutes)
19             {
20                 Console.WriteLine(number);
21             }
22
23             Console.WriteLine("何かキーを押すと終了します。");
24             Console.ReadKey();
25         }
26     }
27 }
```

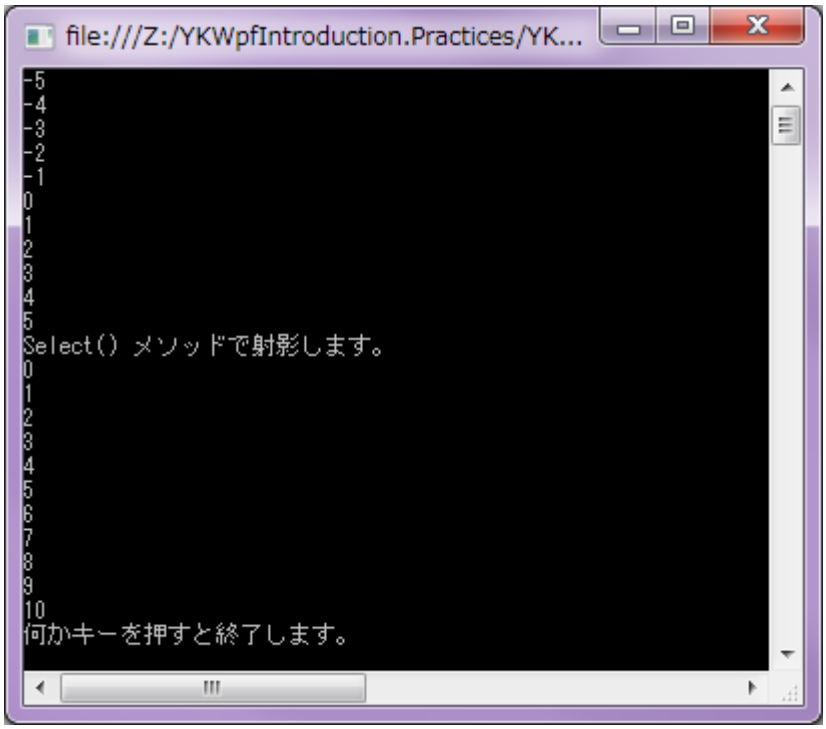


図 2.20：インデックス番号が出力されている

2.8.3 メソッドチェーン

Linq は Range() や Select() の他にも様々な拡張メソッドが用意されています。これらはコード 2.40 のようにいちいち別の変数を用意してやる必要はなく、すべてを繋げて記述することができるようになっています。コード 2.40 をメソッドチェーンを用いて書き直すと次のようになります。

コード 2.42：Range() メソッドに Select() メソッドを繋げて記述できる

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Linq;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var numbers = Enumerable.Range(-5, 11).Select(x => Math.Abs(x));
11             foreach (var number in numbers)
12             {
13                 Console.WriteLine(number);
14             }
15
16             Console.WriteLine("何かキーを押すと終了します。");
17             Console.ReadKey();
18         }
19     }
20 }
```

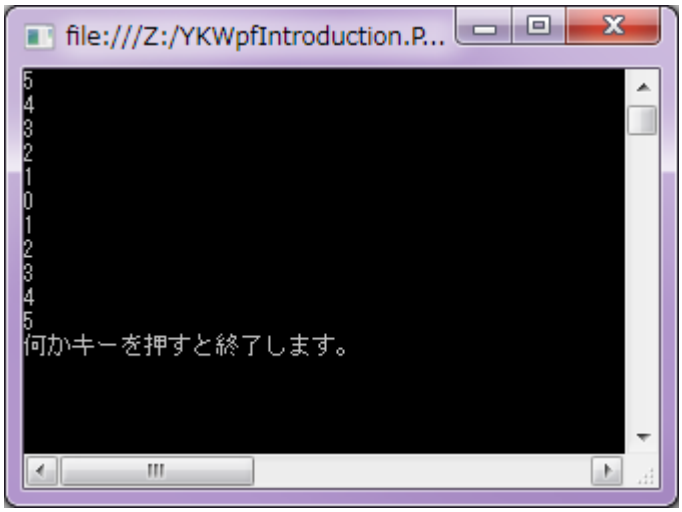


図 2.21：最終的に得られる結果は同じ

Range() メソッドの後に続けて Select() メソッドをドット "." で繋げて書くことができます。このようにメソッドを鎖のように繋げて記述するため、メソッドチェーンと呼ばれます。

2.8.4 Where() メソッドでフィルタリングする

Enumerable.Where() メソッドはコレクションの要素をフィルタリングすることができる拡張メソッドです。

コード 2.43：Where() メソッドでフィルタリングをおこなう

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Linq;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var numbers = Enumerable.Range(-5, 11).Where(x => x < 3);
11             foreach (var number in numbers)
12             {
13                 Console.WriteLine(number);
14             }
15
16             Console.WriteLine("何かキーを押すと終了します。");
17             Console.ReadKey();
18         }
19     }
20 }
```

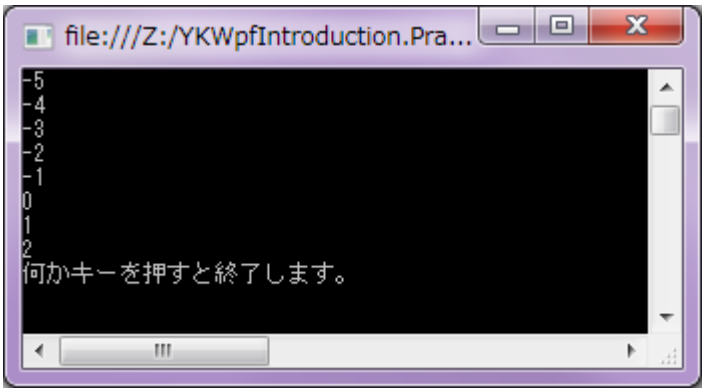


図 2.22：フィルタ条件を満たすものだけのコレクションとなっている

Where() メソッドはフィルタ条件を Func<object, bool> 型のデリゲートとして入力引数に与えます。コレクションの各要素が入力引数となり、true を返す要素のみを抽出します。

2.8.5 Person クラスを操作してみよう

メソッドチェーンを使って Person クラスを操作してみましょう。

コード 2.44：Where() メソッドでフィルタリングをおこなう

Program.cs

```
1  namespace YKWpfIntroduction.ConsoleApplication
2  {
3      using System;
4      using System.Linq;
5
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             var person = Enumerable.Range(0, 100).Select(x =>
11             {
12                 var p = new Person();
13                 p.SpeakingToMyself("私は Person " + x + " 号だ。");
14                 return new
15                 {
16                     index = x,
17                     person = p,
18                 };
19             }).Where(x => x.index < 30).Skip(20).Select(x => x.person).First();
20
21             Console.WriteLine(person.Statement);
22
23             Console.WriteLine("何かキーを押すと終了します。");
24             Console.ReadKey();
25         }
26     }
27 }
```

Range() メソッドで 100 個の整数を用意し、その整数を利用して Select() メソッドで Person クラスのコレクションを生成しています。ただし、index プロパティと person プロパティを持つ匿名クラスをコレクションの要素としてます。続いて Where() メソッドでコレクションの index プロパティの値が 30 未満のもののみを抽出し、Skip() メソッドで先頭 20 個の要素を除外しています。さらに Select() メソッドで匿名クラスの person プロパティのみを抽出し、最後に First() メソッドでコレクションの最初の要素を取り出しています。最終的に 10 行目の変数 person の型は Person クラスになっています。

21 行目で得られた Person クラスの Statement プロパティを確認すると "私は Person 20 号だ。" と表示されています。結局、先頭 20 個の要素を除外したときの先頭の要素、つまり 21 番目の Person クラスを抽出していることになっています。

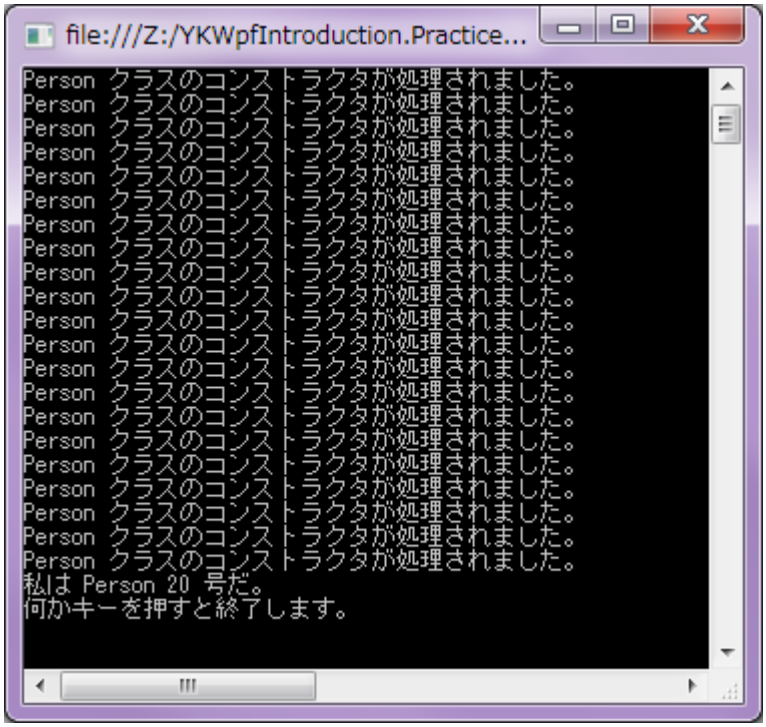


図 2.23：フィルタ条件を満たすものだけのコレクションとなっている

`Enumerable` クラスにはここで紹介した以外の拡張メソッドが多く用意されています。一度自分で調査して、ぜひマスターしてください。

2.9 まとめ

ここでは C# によるコンソールアプリケーションを作成する過程で C# の基本的な機能などについて学習しました。

クラスやプロパティの作り方、デリゲートによるイベントハンドラの仕組みなどについて理解できたでしょうか。匿名関数を使うことで効率的なコードが記述できるようになっており、ラムダ式によって匿名関数のデリゲートを作成できるようになるとより一層レベルの高いコードを記述できるようになるので必ずマスターしましょう。C# を扱うからには System.Linq が提供する拡張メソッドは必ず使えるようになったほうが良いでしょう。

3 MVVM パターンで学ぶ基礎

この章では Visual Studio で WPF アプリケーションを作成するためのプロジェクトを作成します。ここで作成するプロジェクトファイルは MVVM パターンにしたがって WPF アプリケーションを作成するときのテンプレート的なプロジェクトとなるので必ず修得しましょう。

3.1 WPF アプリケーションプロジェクトを作成する

Visual Studio を起動後、「ファイル」→「新規作成」→「プロジェクト」を選択すると、次のようなダイアログが開きます。このダイアログの左側のツリーメニューから「Visual C#」を選択すると、真ん中のリストに「WPF アプリケーション」という項目があるので、これを選択します。そして、ダイアログ下部でプロジェクト名、保存場所、ソリューション名を指定して OK ボタンを押します。

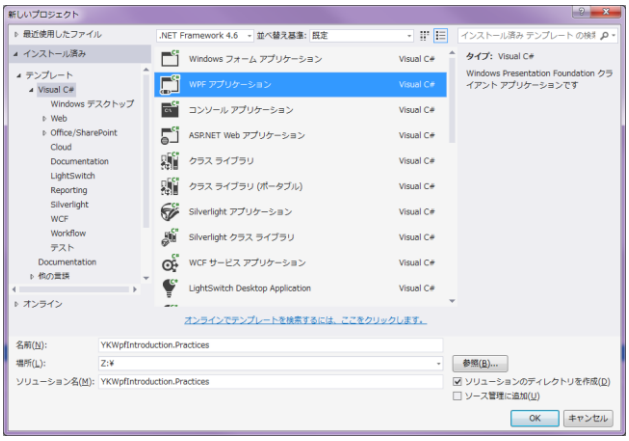


図 3.1：新しいプロジェクトを追加する

ここではソリューション名とプロジェクト名を YKWpfIntroduction.Practices として WPF アプリケーションのプロジェクトを作成します。プロジェクトを作成すると、次のような画面になります。

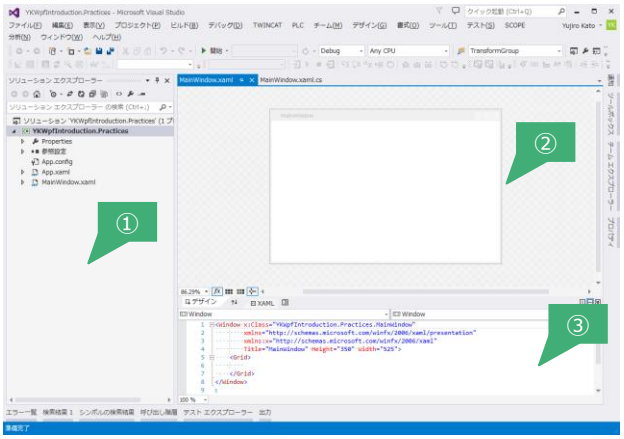


図 3.2：WPF アプリケーションプロジェクト作成時の画面

① ソリューションエクスプローラーではソリューションに含まれるファイル群がツリー形式で表示されます。編集したいソースファイルはここから選んでダブルクリックすると開くことができます。

② XAML デザイナーは、WPF の外観を定義する XAML（ザムル：Extensible Application Markup Language）ファイルを編集するときに表示されます。ツールボックスウィンドウからコントロールをドラッグ＆ドロップすることでコントロールを配置することもできますが、③ XAML エディターを使用して XAML コードを直接入力することでもコントロールを配置することができます。

せっかくですので、③ XAML デザイナーの 6 行目にコントロールを追加してみましょう。以下のコードのように TextBlock コントロールを配置してから F5 キーを押して実行してみましょう。

コード 3.1：WPF で Hello world.

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="350" Width="525">
5      <Grid>
6          <TextBlock Text="Hello world." />
7      </Grid>
8  </Window>
```

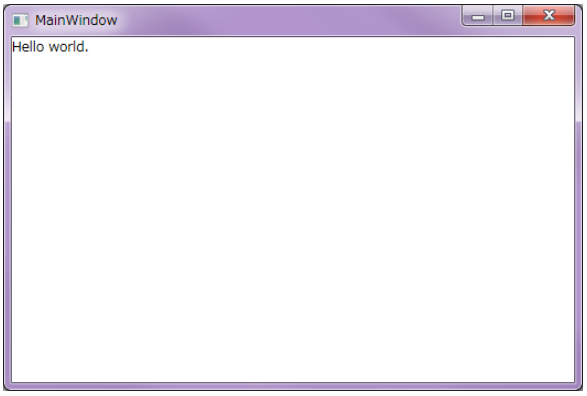


図 3.3："Hello world." が表示される

3.2　Main はどこに？

コンソールアプリケーションや VC++ の MFC アプリケーションは必ず Main() 関数があり、すべてのアプリケーションはこの Main() 関数から実行されます。WPF も例外なくこの Main() 関数から始まっています。ところが、ソリューションエクスプローラーに含まれているどのファイルを覗いても Main() 関数が見当たりません。実は、WPF アプリケーションの Main() 関数はコンパイラが自動生成しています。これを確認してみましょう。

ソリューションエクスプローラーでプロジェクトを右クリックし、「エクスプローラーでフォルダーを開く」メニューを選択すると、プロジェクトファイルがある場所を Windows のエクスプローラーで開くことができます。

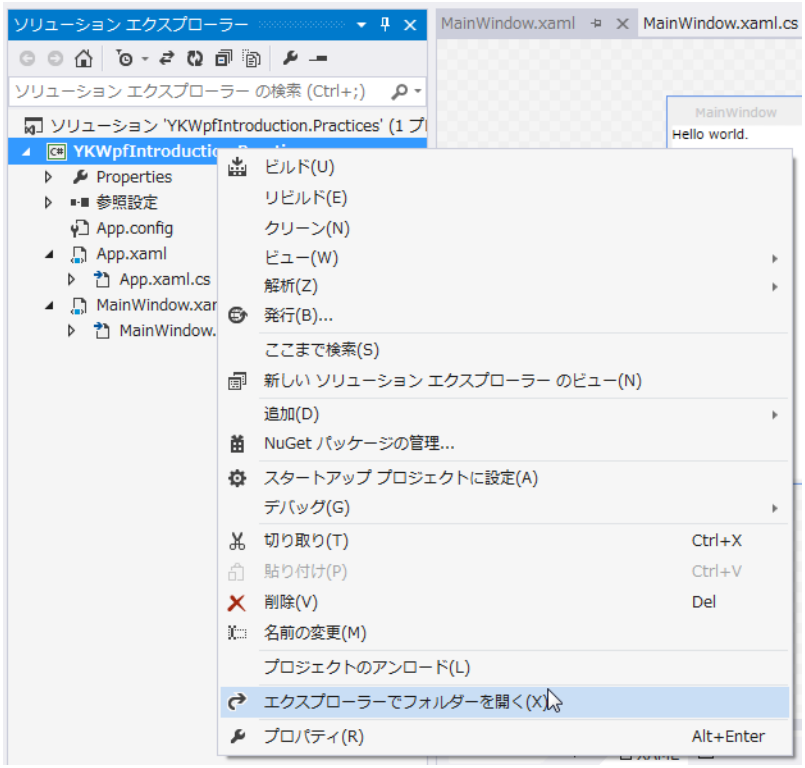


図 3.4：エクスプローラーで開くときは右クリックメニュー

一度ビルドすると、プロジェクトのある同一フォルダ内に obj フォルダが生成されています。obj¥Debug フォルダにビルドしたときのオブジェクトファイルが格納されています。この中に App.g.cs というファイルがあるのでこれを開いてみましょう。すると、下の方に次のような記述があります。

コード 3.2：オブジェクトファイル内で Main() 関数が自動生成されている

App.g.cs

```
56          /// <summary>
57          /// Application Entry Point.
58          /// </summary>
59          [System.STAThreadAttribute()]
60          [System.Diagnostics.DebuggerNonUserCodeAttribute()]
61          [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
62          public static void Main() {
63              YKWpfIntroduction.Practices.App app = new YKWpfIntroduction.Practices.App();
64              app.InitializeComponent();
65              app.Run();
66          }
```

62 行目に Main() 関数が定義されているのがわかると思います。また、63 行目で YKWpfIntroduction.Practices.App クラスをインスタンス化し、65 行目で Run() メソッドをコールしています。YKWpfIntroduction.Practices.App クラスというのは、実は作成した WPF アプリケーションのプロジェクトの中のファイルに含まれています。

ソリューションエクスプローラーに含まれている App.xaml.cs ファイルをダブルクリックしてください。

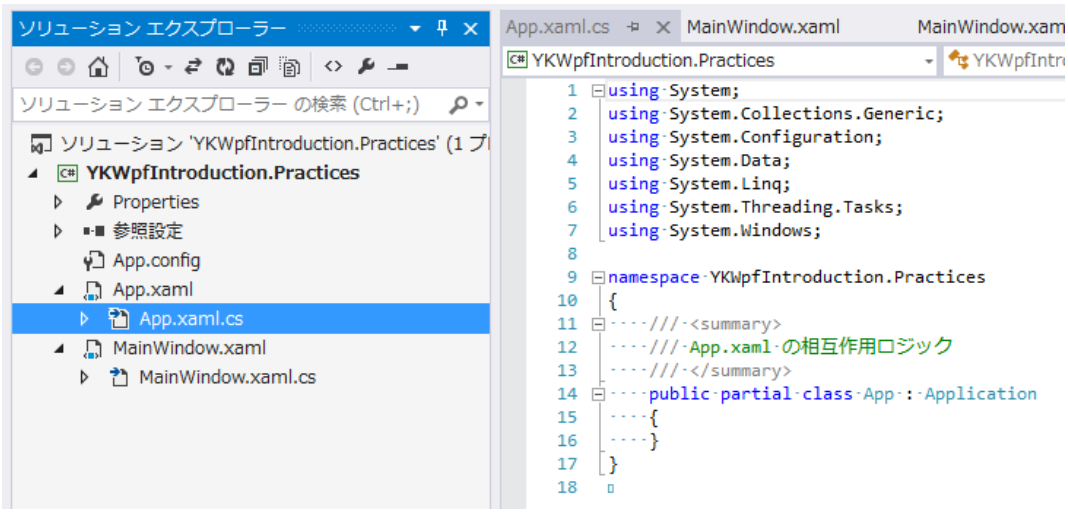


図 3.5：App.xaml のコードビハインド App.xaml.cs を確認

App.xaml.cs の 14 行目で、YKWpfIntroduction.Practices.App クラスが定義されています。コンパイラで自動生成された Main() 関数の中で、この App クラスが実体化し、自動的に Run() メソッドが呼ばれてこのアプリケーションが起動するようになっているわけです。

ところで、App クラスの Run() メソッドが呼ばれたからといって、なぜ MainWindow.xaml で定義したウィンドウが表示されるのでしょうか。実はこれは App.xaml の中で定義されています。App.xaml は次のようなコードになっています。

コード 3.3：StartupUri プロパティで起動時のウィンドウを指定している

App.xaml

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

よく見てみると、4 行目に StartupUri="MainWindow.xaml" という記述があるのがわかります。StartupUri プロパティは、アプリケーション起動時に自動的に表示させる UI を指定することができます。デフォルトの WPF アプリケーションプロジェクトはこのような設定がされているため、いきなりビルド、実行されると MainWindow が表示されるようになっています。

ワンポイント 3.1：partial 修飾子でクラス定義を分割

App.xaml は YKWpfIntroduction.Practices.App クラスを XAML で定義しており、App.xaml.cs は YKWpfIntroduction.Practices.App クラスを C# コードで定義しています。2 つのファイルで同名のクラスを定義しているので、二重定義しているように見えますが、実は C# コードのほうでは partial 修飾子を付けてクラスを定義しています。partial 修飾子を付けることで、クラスの定義を分割して記述できるようになります。

他にも MainWindow.xaml と MainWindow.xaml.cs なども partial 修飾子によって定義が分割されています。

ワンポイント 3.2：コードビハインド

MainWindow.xaml はウィンドウの外観を定義するための XAML コードが書かれているファイルです。しかし、例えばイベントハンドラなどは XAML では直接記述できないため、このようなロジカルな処理については C# コードに頼らざるを得ません。このような場合、MainWindow.xaml.cs で分割して定義されている MainWindow クラスに記述することができます。このとき、XAML で定義されたクラスの裏で記述されている C# コードのことをコードビハインドと呼びます。

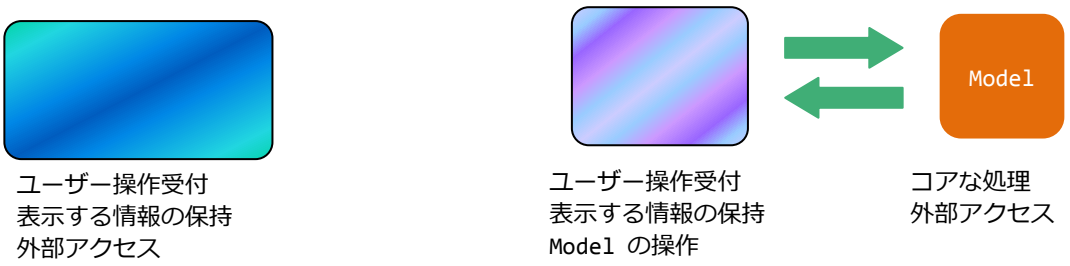
WinForms では外観定義の裏にコードを書くスタイルが一般的なため、WPF でもコードビハインドに処理を追加していくと思われがちですが、そうではありません。WPF は根本的に MVVM パターンをサポートしているフレームワークであるため、コードビハインドはあまり使用せず、データバインディングによるプロパティの同期をおこなうことで ViewModel あるいは Model で複雑な処理をおこなうことになります。

3.3　MVVM パターンとは

MVVM パターン（Model-View-ViewModel Pattern）とはソフトウェアアーキテクチャのひとつで、アプリケーションの内部構造をどのようにするかを決めるための指針となります。

アプリケーションの内部構造をどのようにするか考えずにコーディングしてしまうと、どこでどんな処理をしているのかわからない、いろんなオブジェクトが依存し合う、いわゆるスパゲティコードになってしまいます。しかし、ある程度経験のある方は自分なりのスタイルを持っており、操作系、表示系など機能別に分けられるようにコーディングする場合もあると思います。MVVM パターンもまさしくこのように機能別に分けて考えるやり方です。

図 3.6 に MVVM パターンの考え方を示します。同図 (a) は何も考えずに作ったアプリケーションを表しています。ユーザー操作の受け付けや表示する情報の保持、サーバなどの外部アクセスを一手に担っているため、非常に混沌としています。この中から GUI にまったく依存しない処理を抽出して Model と名付けます。このときの状態が同図 (b) となります。これだけでもかなりコードがすっきりしますが、ここからさらに外観に関係しないものを ViewModel として抽出します。このときの状態が同図 (c) となり、これが MVVM パターンの形となります。



(a) スパゲティ状態になっている　　(b) コアな処理を抽出

(c) 責務の細分化

図 3.6：MVVM パターンの考え方

View、ViewModel、Model それぞれの役割は下表のようになります。

表 3.1：View、ViewModel、Model の役割

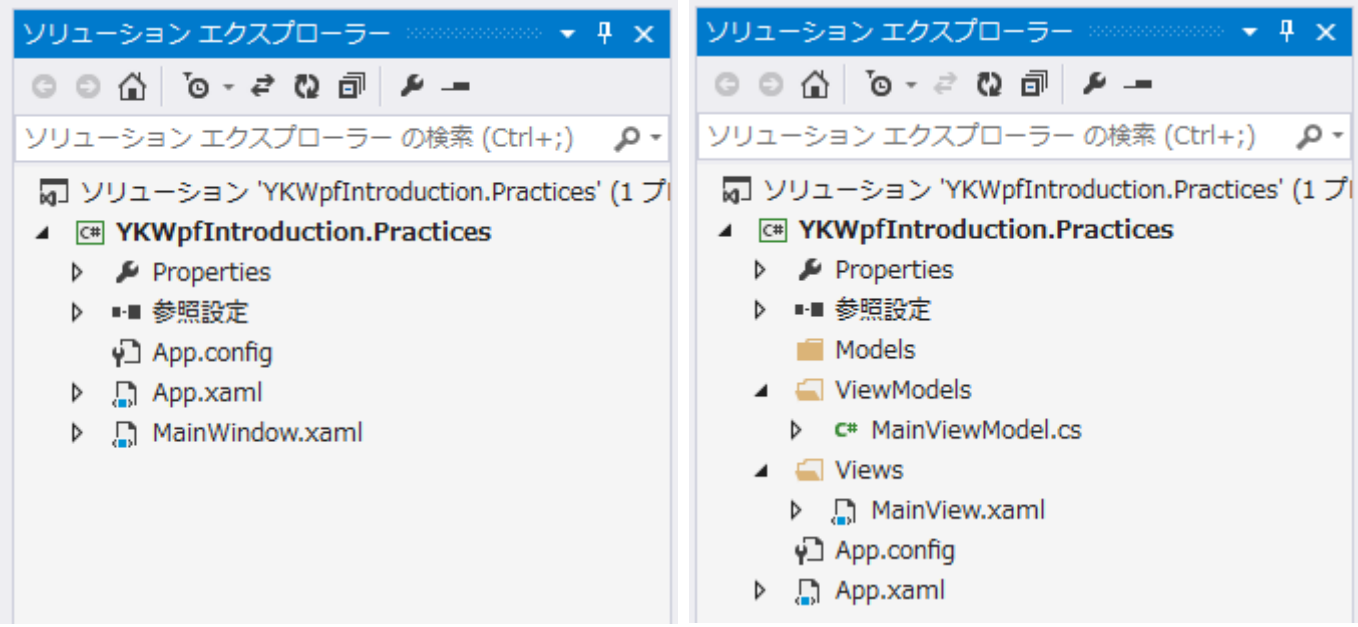
名称	役割	備考
View	情報の表示 ユーザー操作の受付	ViewModel に依存する
ViewModel	表示する情報の保持 表示する情報への変換 Model の操作	View がどのような情報を要求しているか知る必要があるが、 View のインスタンスを持つ必要はない。 View と Model の橋渡しではなく、Model の影となるもの。
Model	UI に依存しない処理 外部へのアクセス 内部的なエラー処理	View にも ViewModel にも依存しない

ここで注意しなければならないのは、MVVM パターンはあくまでもひとつの指針であるということです。必ずしもこのパターンに則った形でなければいけないということはありません。このパターンをひとつの理想形として、自分のアプリケーションに対する内部構造を機能的に設計できればそれで問題ありません。

3.4　MVVM パターンを意識した内部構造にしてみよう

必ずしも MVVM パターンにならなくてもいいとは言いましたが、始めからパターンから逸脱することはおすすめしません。というわけで、WPF アプリケーションプロジェクトのファイル構造を MVVM パターンにしてみましょう。

「3.1 WPF アプリケーションプロジェクトを作成する」で作成したプロジェクトでは、MainWindow.xaml を含む図 3.7 (a) のようなファイル構造でした。これを図 3.7 (b) のようなファイル構造にします。



(a) デフォルトの内部構造　　(b) 変更後の内部構造

図 3.7：MVVM パターンを意識した内部構造に変更する

まず、MainWindow.xaml は削除します。次にプロジェクトを右クリックして「追加」→「新しいフォルダー」を選択して「Models」、「ViewModels」、「Views」という名前のフォルダを作ります。そして、「ViewModels」フォルダに MainViewModel クラス、「Views」フォルダに MainView ウィンドウを追加します。クラスを追加するときは右クリックメニューの「追加」→「クラス」を、ウィンドウを追加するときは右クリックメニューの「追加」→「ウィンドウ」を選択します。「Models」フォルダの中身は空っぽのままです。なぜなら、まだどんなアプリケーションにするか決めていないからです。

これで見た目は MVVM パターンっぽくなりましたが、このままではコンパイルが通りません。前節でも説明したように、App クラスがデフォルトで MainWindow を参照しているからです。これを修正します。
まず、App.xaml に記述されていた StartupUri プロパティを削除します。削除後の App.xaml のコードは以下のようになります。

コード 3.4：StartupUri プロパティを削除する

App.xaml

```
1  <Application x:Class="YKWpfIntroduction.Practices.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>
```

そして、App.xaml.cs のコードビハインドで起動時の処理をオーバーライドします。

コード 3.5：OnStartup() メソッドをオーバーライドして起動時処理をおこなう

App.xaml.cs

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

```
 5      using YKWpfIntroduction.Practices.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              // ウィンドウをインスタンス化します
17              var w = new MainView();
18
19              // ウィンドウに対する ViewModel をインスタンス化します
20              var vm = new MainViewModel();
21
22              // ウィンドウに対する ViewModel をデータコンテキストに指定します
23              w.DataContext = vm;
24
25              // ウィンドウを表示します
26              w.Show();
27          }
28      }
29  }
```

OnStartup() メソッドはアプリケーション起動時に処理されるメソッドなのでこれをオーバーライドすることで起動時の処理を追加しています。この中で、17 行目で MainView ウィンドウをインスタンス化し、26 行目でこのウィンドウを表示しています。これだけでもウィンドウが表示されますが、その間で何やら処理をしています。

20 行目では、MainViewModel クラスのインスタンスを生成し、23 行目で MainView ウィンドウのデータコンテキストにこの MainViewModel クラスのインスタンスを指定しています。データコンテキストとは、オブジェクト内の要素がデータバインディングをおこなうときに参照するオブジェクトのことです。MainView ウィンドウ内でデータバインディングする相手として MainViewModel を指定していることになります。

これでファイル構造は MVVM パターンのようになりましたが、実はこれだけでは足りません。真に MVVM パターンとするには、INotifyPropertyChanged インターフェースや ICommand インターフェースを実装したクラスを使わなければなりません。これらについて以降で説明します。

ワンポイント 3.3：DataContext は伝播する

MainView ウィンドウに対するデータコンテキストに対して MainViewModel クラスを指定したので、MainView ウィンドウがデータバインディングする相手は MainViewModel クラスになりました。では、MainView ウィンドウが持つ Grid パネルや Button コントロールのデータコンテキストはどうなるのでしょうか。正解は MainView ウィンドウと同じ MainViewModel クラスとなります。データコンテキストは特に指定しない場合、親要素のデータコンテキストを参照するようになっています。したがって、MainView ウィンドウの子要素となるコントロールのデータコンテキストはすべて MainViewModel クラスとなります。

3.5 INotifyPropertyChanged インターフェースの実装

前節で、MainView ウィンドウのデータコンテキストが MainViewModel クラスとなったので、さっそく MainViewModel のプロパティを MainView ウィンドウに表示してみたいと思います。まず MainViewModel クラスに 2 つのプロパティを追加します。

コード 3.6：MainViewModel クラスにプロパティを追加

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      /// <summary>
4      /// MainView ウィンドウに対するデータコンテキストを表します。
5      /// </summary>
6      internal class MainViewModel
7      {
8          private string _upperString;
9          /// <summary>
10         /// すべて大文字に変換した文字列を取得します。
11         /// </summary>
12         public string UpperString
13         {
14             get { return this._upperString; }
15             private set
16             {
17                 if (this._upperString != value)
18                 {
19                     this._upperString = value;
20                 }
21             }
22         }
23
24         private string _inputString;
25         /// <summary>
26         /// 入力文字列を取得または設定します。
27         /// </summary>
28         public string InputString
29         {
30             get { return this._inputString; }
31             set
32             {
33                 if (this._inputString != value)
34                 {
35                     this._inputString = value;
36
37                     // 入力された文字列を大文字に変換します
38                     this.UpperString = this._inputString.ToUpper();
39
40                     // 出力ウィンドウに結果を表示します
41                     System.Diagnostics.Debug.WriteLine("UpperString=" + this.UpperString);
42                 }
43             }
44         }
45     }
46 }
```

MainViewModel クラスに UpperString プロパティと InputString プロパティを定義しています。UpperString プロパティは string 型で、get アクセサのみを公開しています。つまり外部からは取得できますが、設定することができない読取専用のプロパティです。InputString プロパティは string 型で、get アクセサも set アクセサも公開しています。また、set アクセサ内で設定された値を大文字に変換して自分の UpperString プロパティに設定しています。

30 行目で使用している System.Diagnostics.Debug クラスはデバッグ用のクラスで、WriteLine() メソッドによって Visual Studio の出力ウィンドウに指定した文字列を出力させることができます。Debug クラスに定義されているメソッ

ドはすべて DEBUG シンボルが定義されているときのみコンパイルされるようになっています。つまり、Release モードでコンパイルすると、Debug.WriteLine() メソッドなどはコールされないようになるので、デバッグコードとしてよく用いられます。

ワンポイント 3.4：System.Diagnostics.Debug クラスでデバッグコードを書くようにしよう

デバッグコードとして出力ウィンドウにメッセージを出力したいときは System.Diagnostics.Debug クラスを使いましょう。System.Console クラスの WriteLine() メソッドでも同様に出力ウィンドウにメッセージを出力させることができますが、こちらは Release モードでコンパイルしてもコードとして残ってしまうため、Release モードでもどこかにメッセージが出力されることになります。

コード上で "System.Diagnostics.Debug.WriteLine" と記述した後、"Debug" の部分にキーボードカーソルを置いてから F12 キーを押してみましょう。すると Debug クラスの定義を覗くことができます。開いた直後はすべてアウトラインが閉じているので、[+] をクリックしてアウトラインを開くと各メソッドの概要などのコメントが見えるようになります。その中で、メソッドの定義の真上の行に必ず "[Conditional("DEBUG")]" が付いていることが確認できます。これは、条件付きコンパイルシンボルに "DEBUG" が定義されているときのみコンパイルされるメソッドであることを意味します。デフォルトのプロジェクト設定では、Debug モードのときは必ず DEBUG シンボルが定義されるようになっています。

続いて MainView ウィンドウの XAML を次のように編集します。

コード 3.7：MainView ウィンドウでデータバインディングを設定

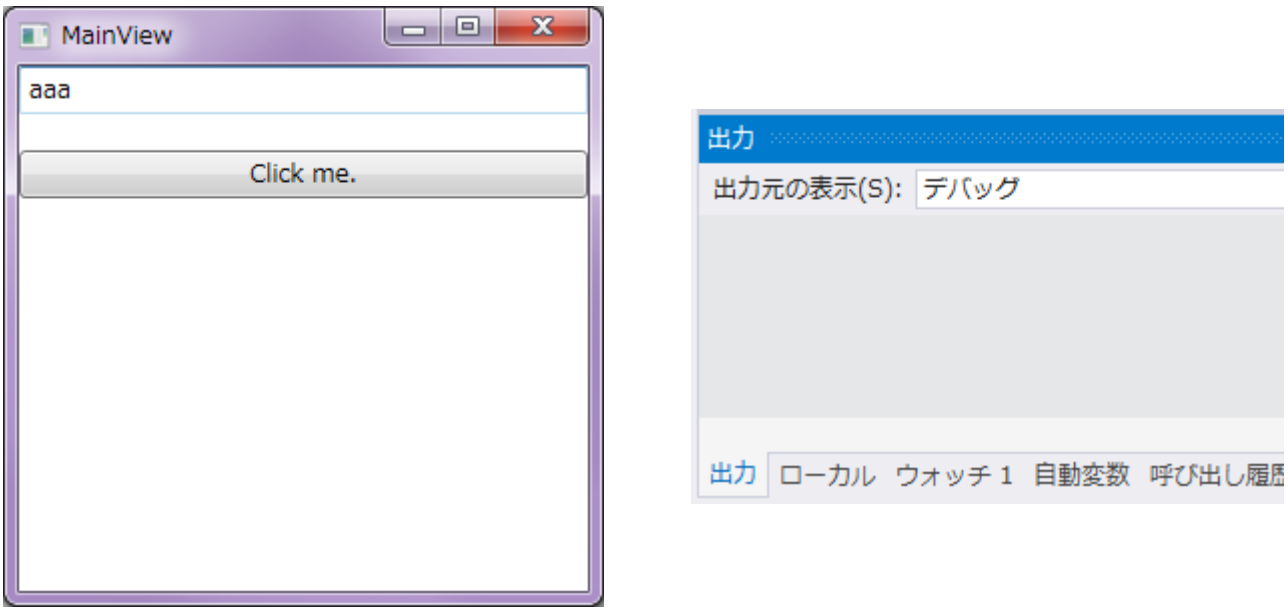
MainView.xaml

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

TextBox コントロールはユーザーにテキストを入力させるためのコントロールです。入力されるテキストは Text プロパティとして取得できます。今回はこの Text プロパティに対して MainViewModel クラスの InputString プロパティをデータバインディングで同期させます。つまり、TextBox コントロールで文字列が入力されると、 MainViewModel クラスの InputString プロパティが変更されるようになります。

TextBlock コントロールはユーザーにテキストを表示するためのコントロールです。表示するテキストは Text プロパティで指定できます。今回はこの Text プロパティに対して MainViewModel クラスの UpperString プロパティをデータバインディングで同期させます。

つまり、このサンプルアプリケーションは、ユーザーに入力された文字列を大文字に変換して表示する、という機能を実現しようとしています。それではさっそく実行してみましょう。実はこのままではうまくいきません。



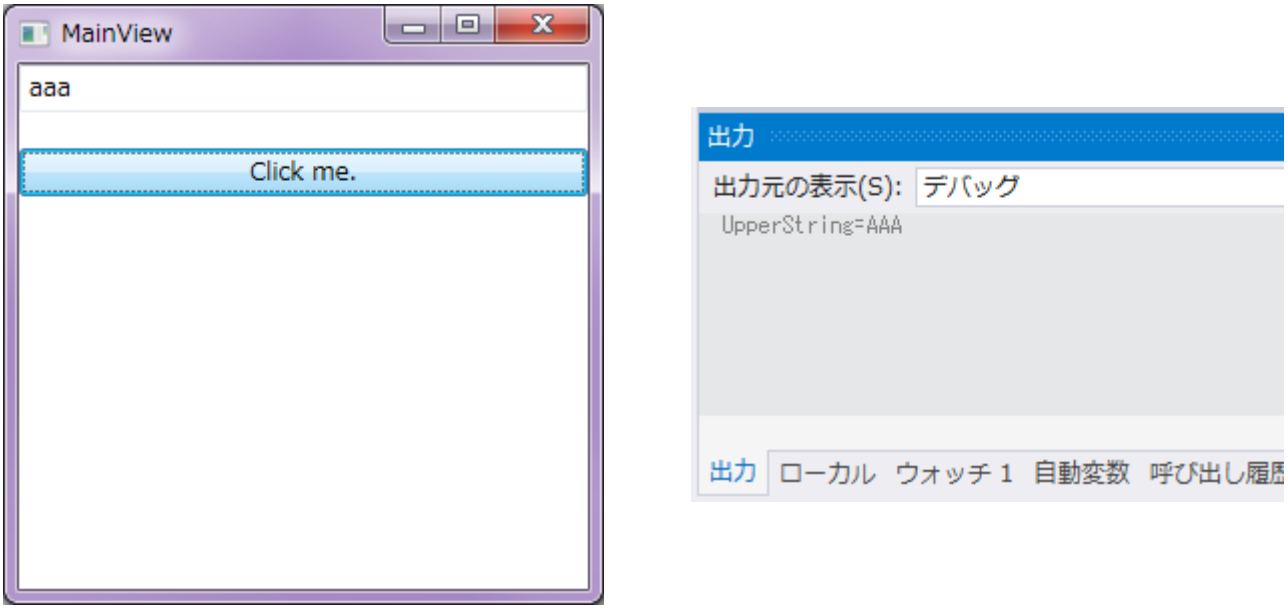
(a) テキストを入力できる　　(b) 出力ウィンドウに特に表示されない

図 3.8：サンプルコードを実行してもうまくいかない

TextBox コントロールに文字列を入力すると MainViewModel クラスの InputString プロパティが設定されるはずなので、MainViewModel クラスのソースの 41 行目に指定した Debug.WriteLine() メソッドが実行され、Visual Studio の出力ウィンドウに文字列が表示されるようになっているはずですが、これが表示されません。ということはこの行が実行されていないということです。

どういうことかというと、TextBox コントロールに文字列を入力してもその変更がただちにデータコンテキストへ通知されるわけではありません。TextBox コントロールの場合、文字列を入力した後、フォーカスを失ったときにその変更が通知されるようになっています。

このことを確認するために、TextBox コントロールに文字列を入力した後、Tab キーを押してフォーカスを Button コントロールへ移動させてみてください。その直後に出力ウィンドウにメッセージが表示されます。



(a) テキストを入力した後にフォーカスを移す　　(b) 出力ウィンドウに表示される

図 3.9：値が設定されるのに UI に反映されない

確かに set アクセサが実行され、デバッグコードが処理されたことが確認できました。しかし、TextBox コントロールと Button コントロールの間を確認してください。ここには MainViewModel クラスの UpperString プロパティを表示するための TextBlock コントロールがあるはずですが、文字列が何も表示されていません。実はこのサンプルコードのままではこれで正しい動作となります。

上記のサンプルのように、WPF では UI からプロパティが変更されたことはデータコンテキストに自動的に通知されますが、データコンテキストからプロパティが変更されたことは自動的には通知されません。つまり、MainViewModel 側から明示的にプロパティ変更通知をおこなう必要があります。これは INotifyPropertyChanged インターフェースによって実現できます。

それでは、コード 3.6 のコードに対して INotifyPropertyChanged インターフェースを実装しましょう。インターフェースを実装するときは、図 3.10 のように Visual Studio の機能を使うと実装漏れがなく確実な実装ができます。基本クラスに "INotifyPropertyChanged" と入力した後、マウスポインタを動かすと表示されるメニューから選択できます。

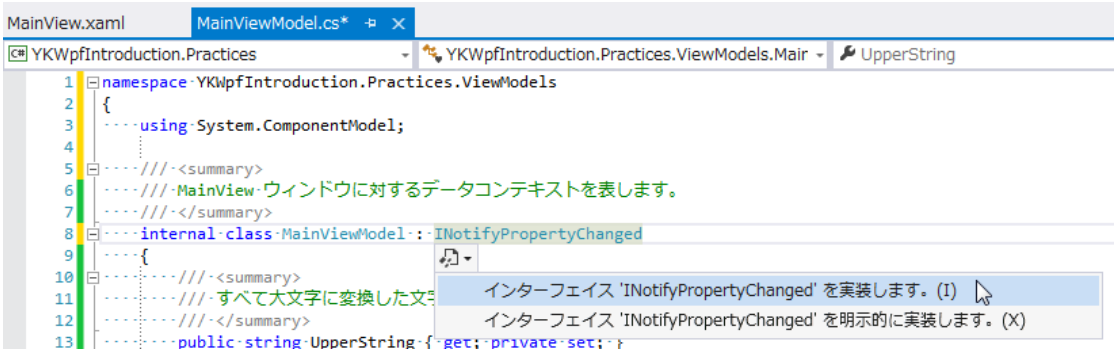


図 3.10：インターフェース名にキーボードカーソルを置くと表示されるメニューで実装する

コード 3.8：MainViewModel クラスに INotifyPropertyChanged インターフェースを実装する

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      using System.ComponentModel;
4
5      /// <summary>
6      /// MainView ウィンドウに対するデータコンテキストを表します。
7      /// </summary>
8      internal class MainViewModel : INotifyPropertyChanged
9      {
10         private string _upperString;
11         /// <summary>
12         /// すべて大文字に変換した文字列を取得します。
13         /// </summary>
14         public string UpperString
15         {
16             get { return this._upperString; }
17             private set
18             {
19                 if (this._upperString != value)
20                 {
21                     this._upperString = value;
22                     RaiseProeprtyChanged("UpperString");
23                 }
24             }
25         }
26
27         private string _inputString;
28         /// <summary>
29         /// 入力文字列を取得または設定します。
30         /// </summary>
31         public string InputString
32         {
33             get { return this._inputString; }
34             set
35             {
```

```
36                 if (this._inputString != value)
37                 {
38                     this._inputString = value;
39                     RaiseProeprtyChanged("InputString");
40 
41                     // 入力された文字列を大文字に変換します
42                     this.UpperString = this._inputString.ToUpper();
43 
44                     // 出力ウィンドウに結果を表示します
45                     System.Diagnostics.Debug.WriteLine("UpperString=" + this.UpperString);
46                 }
47             }
48         }
49 
50         #region INotifyPropertyChanged の実装
51         /// <summary>
52         /// プロパティに変更があった場合に発生します。
53         /// </summary>
54         public event PropertyChangedEventHandler PropertyChanged;
55 
56         /// <summary>
57         /// PropertyChanged イベントを発行します。
58         /// </summary>
59         /// <param name="propertyName">プロパティ値に変更があったプロパティ名を指定します。</param>
60         private void RaiseProeprtyChanged(string propertyName)
61         {
62             var h = this.PropertyChanged;
63             if (h != null) h(this, new PropertyChangedEventArgs(propertyName));
64         }
65         #endregion INotifyPropertyChanged の実装
66     }
67 }
```

INotifyPropertyChanged インターフェースは PropertyChanged イベントを持つインターフェースです。これをプロパティ値に変更があったタイミングで発行することで、プロパティ変更を UI 側へ通知することができます。このため、イベント発行をおこなう RaisePropertyChanged() メソッドを 60 行目で定義し、各プロパティ値が変更されたタイミング（22、39 行目）でこのメソッドを呼び出しています。

それでは、MainViewModel に INotifyPropertyChanged インターフェースを実装したので、もう一度実行してみましょう。データコンテキスト側から UI へプロパティ値の変更が通知されるようになったので、図 3.11 のように、テキスト入力後にフォーカスを移すと、TextBlock コントロールに大文字に変換された文字列が表示されるようになりました。

ワンポイント 3.5：#region ～ #endregion でアウトライン機能を使いこなせ

Visual Studio で C# コーディングをするとき、コード 3.8 の 50 行目と 65 行目のように #region ～ #endregion で括ると、括ったコードを折り畳むことができるようになります。縦に長いソースファイルは可読性が良くないため、このようなアウトライン機能を活用してコードを見やすくするための努力をしましょう。

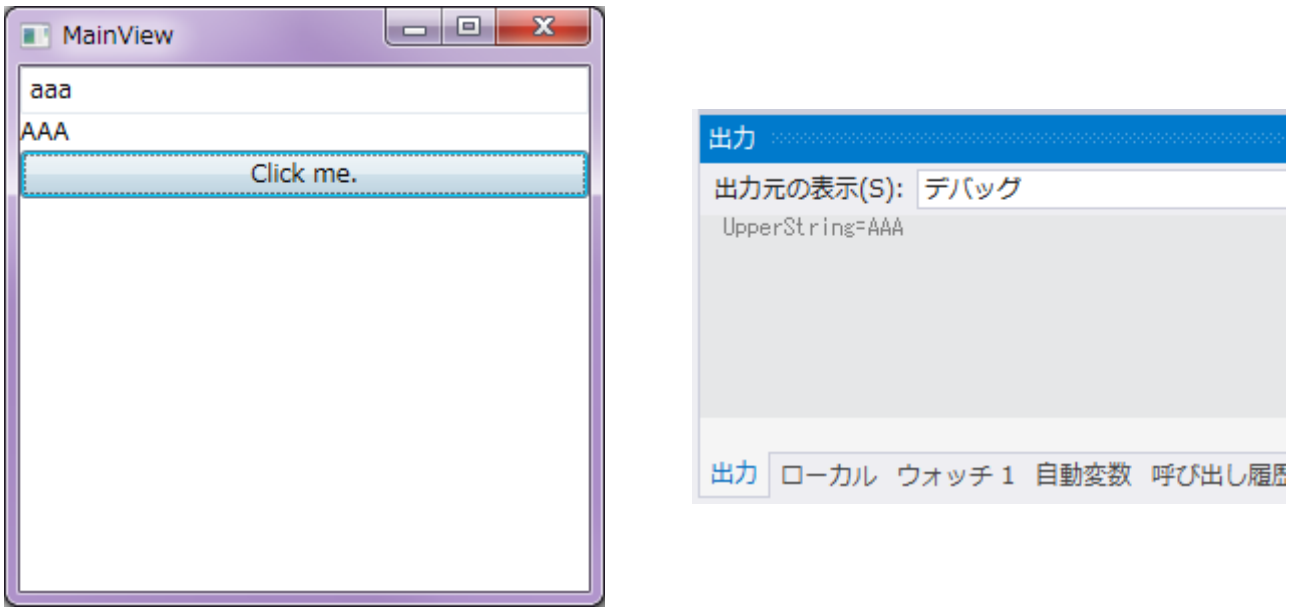
また、Visual Stuido には拡張機能を追加することができますが、「C# outline 2013」という拡張機能を追加すると、すべての "{" ～ "}" で囲まれた部分をアウトライン化してくれるようになります。するとメソッド単位で畳んだり、if 文のコードを畳んだりすることができるようになります。拡張機能は「ツール」→「拡張機能と更新プログラム」メニューで開くダイアログで、ツリーメニューから「オンライン」を選択して「C# outline 2013」を検索すると設定できます。

ワンポイント 3.6：データバインディングの同期タイミングは UpdateSourceTrigger で変更できる

TextBox コントロールの Text プロパティの同期タイミングはデフォルトではフォーカスを失ったときですが、これをプロパティ変更時に同期させることができます。XAML でデータバインディングの設定をおこなうとき、

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

とすると、テキストを入力した直後に同期されるため、フォーカスを移動させなくてもデータコンテキストに変更通知されるようになります。結果的に、テキストを入力した直後に表示が反映されるようになります。



(a) テキストを入力した後にフォーカスを移す　　(b) 出力ウィンドウに表示される

図 3.11：プロパティ変更が通知されて UI が反映されている

コード 3.8 では、RaisePropertyChanged() メソッドにプロパティ値に変更のあったプロパティ名を明示的に指定させるようにしていますが、これではプロパティ値変更通知の度にプロパティ名をキーボードで入力しなければなりません。入力引数は文字列であるため、Visual Studio の Intellisense 機能が働かず、タイプミスをする可能性があります。タイプミスした場合、コンパイルも問題なく通るため、なぜうまく動作しないか発見しにくいバグとなってしまいます。

この問題を改善するために、次のように RaisePropertyChanged(() メソッドを作りかえ、呼び出し側も変更します。

コード 3.9：RaisePropertyChanged() メソッドを作りかえた MainViewModel クラス

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      using System.ComponentModel;
4      using System.Runtime.CompilerServices;
5
6      /// <summary>
7      /// MainView ウィンドウに対するデータコンテキストを表します。
8      /// </summary>
9      internal class MainViewModel : INotifyPropertyChanged
10     {
11         private string _upperString;
12         /// <summary>
13         /// すべて大文字に変換した文字列を取得します。
14         /// </summary>
15         public string UpperString
16         {
17             get { return this._upperString; }
18             private set
19             {
20                 if (this._upperString != value)
21                 {
22                     this._upperString = value;
23                     RaiseProeprtyChanged();
24                 }
25             }
26         }
27
28         private string _inputString;
29         /// <summary>
30         /// 入力文字列を取得または設定します。
```

```
31          /// </summary>
32          public string InputString
33          {
34              get { return this._inputString; }
35              set
36              {
37                  if (this._inputString != value)
38                  {
39                      this._inputString = value;
40                      RaiseProeprtyChanged();
41
42                      // 入力された文字列を大文字に変換します
43                      this.UpperString = this._inputString.ToUpper();
44
45                      // 出力ウィンドウに結果を表示します
46                      System.Diagnostics.Debug.WriteLine("UpperString=" + this.UpperString);
47                  }
48              }
49          }
50
51          #region INotifyPropertyChanged の実装
52          /// <summary>
53          /// プロパティに変更があった場合に発生します。
54          /// </summary>
55          public event PropertyChangedEventHandler PropertyChanged;
56
57          /// <summary>
58          /// PropertyChanged イベントを発行します。
59          /// </summary>
60          /// <param name="propertyName">プロパティ値に変更があったプロパティ名を指定します。</param>
61          private void RaiseProeprtyChanged([CallerMemberName]string propertyName = null)
62          {
63              var h = this.PropertyChanged;
64              if (h != null) h(this, new PropertyChangedEventArgs(propertyName));
65          }
66          #endregion INotifyPropertyChanged の実装
67      }
68  }
```

RaisePropertyChanged() メソッドを定義する際、入力引数に CallerMemberName 属性を付加しています。この属性を付けた string 型の入力引数は、特に指定しない場合は呼び出し元のメソッド名またはプロパティ名となります。つまり、例えば 23 行目で入力引数を特に指定せずに RaisePropertyChanged() メソッドを呼び出しているのは UpperString プロパティであるため、61 行目の入力引数 propertyName の値は "UpperString" という文字列になります。また、40 行目で同様に呼び出していますが、この場合は呼び出し元が InputString プロパティであるため、61 行目の入力引数 propertyName の値は "InputString" という文字列になります。

もう少し RaisePropertyChanged() メソッドについて考えてみましょう。このメソッドはプロパティに変更があった場合には必ず呼ばれるメソッドになります。つまり、コード 3.9 の 20 行目のように現在の値と設定される値を比較し、異なったときに現在の値を更新して RaisePropertyChanged() メソッドを呼び出す、といった決まった流れを必ずコード化することになります。同じコードを何度も書くのは非効率なので、この部分をメソッド化することにしましょう。

コード 3.10：SetProperty() メソッドを追加した MainViewModel クラス

MainViewModel.cs

```
1   namespace YKWpfIntroduction.Practices.ViewModels
2   {
3       using System.ComponentModel;
4       using System.Runtime.CompilerServices;
5
6       /// <summary>
7       /// MainView ウィンドウに対するデータコンテキストを表します。
```

```
 8      /// </summary>
 9      internal class MainViewModel : INotifyPropertyChanged
10      {
11          private string _upperString;
12          /// <summary>
13          /// すべて大文字に変換した文字列を取得します。
14          /// </summary>
15          public string UpperString
16          {
17              get { return this._upperString; }
18              private set { SetProperty(ref this._upperString, value); }
19          }
20
21          private string _inputString;
22          /// <summary>
23          /// 入力文字列を取得または設定します。
24          /// </summary>
25          public string InputString
26          {
27              get { return this._inputString; }
28              set
29              {
30                  if (SetProperty(ref this._inputString, value))
31                  {
32                      // 入力された文字列を大文字に変換します
33                      this.UpperString = this._inputString.ToUpper();
34
35                      // 出力ウィンドウに結果を表示します
36                      System.Diagnostics.Debug.WriteLine("UpperString=" + this.UpperString);
37                  }
38              }
39          }
40
41          #region INotifyPropertyChanged の実装
42          /// <summary>
43          /// プロパティに変更があった場合に発生します。
44          /// </summary>
45          public event PropertyChangedEventHandler PropertyChanged;
46
47          /// <summary>
48          /// PropertyChanged イベントを発行します。
49          /// </summary>
50          /// <param name="propertyName">プロパティ値に変更があったプロパティ名を指定します。</param>
51          private void RaiseProeprtyChanged([CallerMemberName]string propertyName = null)
52          {
53              var h = this.PropertyChanged;
54              if (h != null) h(this, new PropertyChangedEventArgs(propertyName));
55          }
56
57          /// <summary>
58          /// プロパティ値を変更するヘルパです。
59          /// </summary>
60          /// <typeparam name="T">プロパティの型を表します。</typeparam>
61          /// <param name="target">変更するプロパティの実体を ref 指定します。</param>
62          /// <param name="value">変更後の値を指定します。</param>
63          /// <param name="propertyName">プロパティ名を指定します。</param>
64          /// <returns>プロパティ値に変更があった場合に true を返します。</returns>
65          private bool SetProperty<T>(ref T target, T value, [CallerMemberName]string propertyName =
    null)
66          {
```

```
67              if (Equals(target, value))
68                  return false;
69              target = value;
70              RaiseProeprtyChanged(propertyName);
71              return true;
72          }
73          #endregion INotifyPropertyChanged の実装
74      }
75  }
```

65 行目で新たに SetProperty() メソッドを追加し、プロパティ値変更にはこのメソッドを使用するようにしています。SetProperty() メソッドでは、値が異なっているか、異なっている場合値を更新し、RaiseProeprtyChanged() メソッドを呼び出す、という一連の処理をおこなっています。プロパティは様々な型が想定されるため、汎用的にするためにジェネリックメソッドを適用しています。ジェネリックメソッドとは、型が異なるだけで処理内容が同一なものを扱うときに重宝する C# 言語仕様の一つです。

このヘルパを使うと、18 行目のように単にプロパティ値を設定するだけのプロパティはたった 1 行で書けるようになります。また、変更があった場合に true を戻り値として返しているため、30 行目のように if 文のステートメントとしても使うことができます。さらに、RaisePropertyChanged() メソッドと同じく CallerMemberName 属性を使用しているため、プロパティ名を明示的に書かなくてもプロパティ値更新を正常におこなえます。

これまで MainViewModel クラスに INotifyPropertyChanged インターフェースを実装し、拡張してきましたが、このような機能は ViewModel に留まらず汎用的に使えるようにしたほうが便利なので、MainViewModel クラスから切り離し、NotificationObject 抽象クラスとして実装し、MainViewModel クラスは NotificationObject クラスからの派生クラスとしましょう。それぞれのコードは次のようになります。

コード 3.11：INotifyPropertyChanged インターフェースを実装した NotificationObject 抽象クラス

NotificationObject.cs

```
1   namespace YKWpfIntroduction.Practices
2   {
3       using System.ComponentModel;
4       using System.Runtime.CompilerServices;
5
6       internal abstract class NotificationObject : INotifyPropertyChanged
7       {
8           #region INotifyPropertyChanged の実装
9           /// <summary>
10          /// プロパティに変更があった場合に発生します。
11          /// </summary>
12          public event PropertyChangedEventHandler PropertyChanged;
13
14          /// <summary>
15          /// PropertyChanged イベントを発行します。
16          /// </summary>
17          /// <param name="propertyName">プロパティ値に変更があったプロパティ名を指定します。</param>
18          protected void RaiseProeprtyChanged([CallerMemberName]string propertyName = null)
19          {
20              var h = this.PropertyChanged;
21              if (h != null) h(this, new PropertyChangedEventArgs(propertyName));
22          }
23
24          /// <summary>
25          /// プロパティ値を変更するヘルパです。
26          /// </summary>
27          /// <typeparam name="T">プロパティの型を表します。</typeparam>
28          /// <param name="target">変更するプロパティの実体を ref 指定します。</param>
29          /// <param name="value">変更後の値を指定します。</param>
30          /// <param name="propertyName">プロパティ名を指定します。</param>
31          /// <returns>プロパティ値に変更があった場合に true を返します。</returns>
32          protected bool SetProperty<T>(ref T target, T value, [CallerMemberName]string propertyName =
    null)
```

```
33          {
34              if (Equals(target, value))
35                  return false;
36              target = value;
37              RaiseProeprtyChanged(propertyName);
38              return true;
39          }
40          #endregion INotifyPropertyChanged の実装
41      }
42  }
```

コード 3.12：NotificationObject 抽象クラスから派生するようにした MainViewModel クラス

MainViewModel.cs

```
1   namespace YKWpfIntroduction.Practices.ViewModels
2   {
3       /// <summary>
4       /// MainView ウィンドウに対するデータコンテキストを表します。
5       /// </summary>
6       internal class MainViewModel : NotificationObject
7       {
8           private string _upperString;
9           /// <summary>
10          /// すべて大文字に変換した文字列を取得します。
11          /// </summary>
12          public string UpperString
13          {
14              get { return this._upperString; }
15              private set { SetProperty(ref this._upperString, value); }
16          }
17
18          private string _inputString;
19          /// <summary>
20          /// 入力文字列を取得または設定します。
21          /// </summary>
22          public string InputString
23          {
24              get { return this._inputString; }
25              set
26              {
27                  if (SetProperty(ref this._inputString, value))
28                  {
29                      // 入力された文字列を大文字に変換します
30                      this.UpperString = this._inputString.ToUpper();
31
32                      // 出力ウィンドウに結果を表示します
33                      System.Diagnostics.Debug.WriteLine("UpperString=" + this.UpperString);
34                  }
35              }
36          }
37      }
38  }
```

3.6 ICommand インターフェースの実装

GUI アプリケーションでは、処理を開始するきっかけとして「ボタンを押す」というユーザー入力が非常によく使われています。WPF ではボタンを押したときの処理を ViewModel へ伝えるために ICommand インターフェースが用意されています。ICommand インターフェースを実装するためには、デリゲートを用いた処理の委譲を利用します。デリゲートについては「2.6 Action クラスと Func<TResult> クラスによるデリゲート」を参照してください。

ICommand インターフェースのメンバは下表のようになっています。

表 3.2：ICommand インターフェースのメンバ

メンバ	分類	説明
void Execute(object)	メソッド	コマンドが実行されたときの処理をおこないます。
bool CanExecute(object)	メソッド	コマンドが実行可能かどうかの判別処理をおこないます。
event EventHandler CanExecuteChanged	イベント	コマンドが実行可能かどうかの判別処理に関する状態が変更したことを UI に通知します。

この 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 3.13：ICommand インターフェースを実装した DelegateCommand クラス

MainViewModel.cs

```
 1 namespace YKWpfIntroduction.Practices
 2 {
 3     using System;
 4     using System.Windows.Input;
 5 
 6     internal class DelegateCommand : ICommand
 7     {
 8         /// <summary>
 9         /// コマンド実行時の処理内容を保持します。
10         /// </summary>
11         private Action<object> _execute;
12 
13         /// <summary>
14         /// コマンド実行可能判別の処理内容を保持します。
15         /// </summary>
16         private Func<object, bool> _canExecute;
17 
18         /// <summary>
19         /// 新しいインスタンスを生成します。
20         /// </summary>
21         /// <param name="execute">コマンド実行処理を指定します。</param>
22         public DelegateCommand(Action<object> execute)
23             : this(execute, null)
24         {
25         }
26 
27         /// <summary>
28         /// 新しいインスタンスを生成します。
29         /// </summary>
30         /// <param name="execute">コマンド実行処理を指定します。</param>
31         /// <param name="canExecute">コマンド実行可能判別処理を指定します。</param>
32         public DelegateCommand(Action<object> execute, Func<object, bool> canExecute)
33         {
34             this._execute = execute;
35             this._canExecute = canExecute;
36         }
37 
38         #region ICommand の実装
```

```
        /// <summary>
        /// コマンドが実行可能かどうかの判別処理をおこないます。
        /// </summary>
        /// <param name="parameter">判別処理に対するパラメータを指定します。</param>
        /// <returns>実行可能な場合に true を返します。</returns>
        public bool CanExecute(object parameter)
        {
            return (this._canExecute != null) ? this._canExecute(parameter) : true;
        }

        /// <summary>
        /// 実行可能かどうかの判別処理に関する状態が変更されたときに発生します。
        /// </summary>
        public event EventHandler CanExecuteChanged;

        /// <summary>
        /// CanExecuteChanged イベントを発行します。
        /// </summary>
        public void RaiseCanExecuteChanged()
        {
            var h = this.CanExecuteChanged;
            if (h != null) h(this, EventArgs.Empty);
        }

        /// <summary>
        /// コマンドが実行されたときの処理をおこないます。
        /// </summary>
        /// <param name="parameter">コマンドに対するパラメータを指定します。</param>
        public void Execute(object parameter)
        {
            if (this._execute != null)
                this._execute(parameter);
        }
        #endregion ICommand の実装
    }
}
```

11 行目や 16 行目にある Action<object> クラスや Func<object, bool> クラスはデリゲートと呼ばれ、メソッドを参照するためのクラスです。Action<object> クラスは入力引数に object、戻り値なしのメソッドに対するデリゲートを表しています。また、Func<object, bool> クラスは入力引数に object、戻り値に bool 型を持つメソッドに対するデリゲートを表しています。

ICommand インターフェースは、コマンドが実行されるときに Execute() メソッドが呼ばれます。DelegateCommand クラスでは、Execute() メソッドが呼ばれると、コンストラクタで指定された _execute が保持しているメソッドを実行するようにしています。つまり、DelegateCommand クラスを使う側の処理が委譲されています。

また、コマンドが実行される前に、そもそもコマンドを実行していいかどうかを評価する機能があります。それが CanExecute() メソッドで、このメソッドの戻り値が false の場合は Execute() メソッドが呼ばれることはありません。DelegateCommand クラスでは、CanExecute() メソッドの処理も外部から指定されるメソッドを実行するようにしています。

CanExecuteChanged イベントは、CanExecute() メソッドによる判別結果が変更されたことを UI へ通知するためのイベントです。ICommand インターフェースを実装したコマンドを UI にデータバインディングすると、CanExecute() メソッドの戻り値が false のとき、データバインディングした相手のコントロールが自動的に無効状態となり、ユーザーの入力を受け付けなくなるようになっています。

文章ばかりで説明しても分からないと思うので、DelegateCommand クラスを実際に使ってみましょう。前節で MainView ウィンドウのデータコンテキストに指定した MainViewModel クラスに ClearCommand プロパティを作成します。

コード 3.14：DelegateCommand の使用例

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
```

```
 3      /// <summary>
 4      /// MainView ウィンドウに対するデータコンテキストを表します。
 5      /// </summary>
 6      internal class MainViewModel : NotificationObject
 7      {
 8          private string _upperString;
 9          /// <summary>
10          /// すべて大文字に変換した文字列を取得します。
11          /// </summary>
12          public string UpperString
13          {
14              get { return this._upperString; }
15              private set { SetProperty(ref this._upperString, value); }
16          }
17
18          private string _inputString;
19          /// <summary>
20          /// 入力文字列を取得または設定します。
21          /// </summary>
22          public string InputString
23          {
24              get { return this._inputString; }
25              set
26              {
27                  if (SetProperty(ref this._inputString, value))
28                  {
29                      // 入力された文字列を大文字に変換します
30                      this.UpperString = this._inputString.ToUpper();
31
32                      // 出力ウィンドウに結果を表示します
33                      System.Diagnostics.Debug.WriteLine("UpperString=" + this.UpperString);
34                  }
35              }
36          }
37
38          private DelegateCommand _clearCommand;
39          /// <summary>
40          /// クリアコマンドを取得します。
41          /// </summary>
42          public DelegateCommand ClearCommand
43          {
44              get
45              {
46                  if (this._clearCommand == null)
47                  {
48                      this._clearCommand = new DelegateCommand(_ =>
49                      {
50                          this.InputString = "";
51                      });
52                  }
53                  return this._clearCommand;
54              }
55          }
56      }
57  }
```

42 行目に ClearCommand プロパティを定義しています。このプロパティは get アクセサのみを持つ読取専用プロパティで、実体は 38 行目のメンバ変数となっています。最初にアクセスされたとき、変数 _clearCommand は null であるため、46 行目が true となり、48 行目で DelegateCommand クラスのインスタンスが代入されます。以降、変数 _clearCommand は変更されることなく 53 行目でそのインスタンスが返されるようになります。

48 行目で DelegateCommand クラスのインスタンスが生成されています。入力引数がひとつだけなので、DelegateComand.CanExecute() メソッドは常に true を返すようになっています。DelegateCommand.Execute() メソッドで実行される処理はラムダ式で定義されており、その処理内容は 50 行目のように InputString プロパティを空にしています。この ClearCommand プロパティをボタンの動作として機能させるため、MainView ウィンドウを次のように編集します。

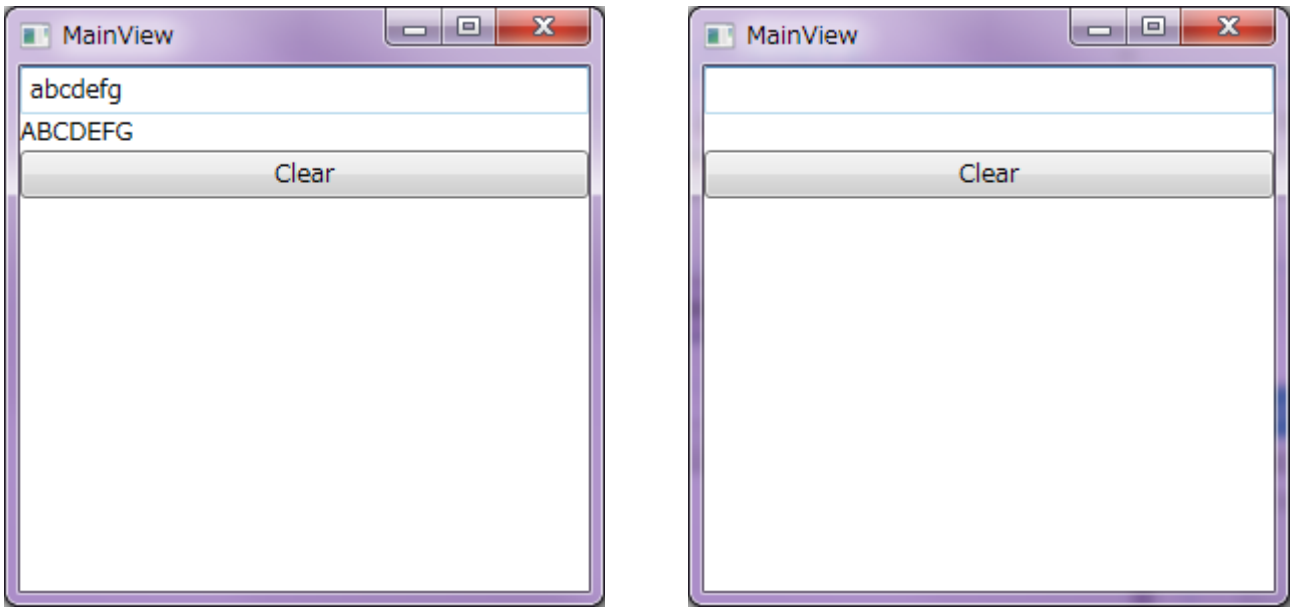
コード 3.15：Button コントロールに ClearCommand プロパティを同期させる

MainView.xaml

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

Button コントロールでボタンを配置することができます。ボタン上にテキストを表示する場合は Content プロパティに文字列を指定します。そして、ボタンを押したときのコマンドは Command プロパティに指定できます。今回は MainViewModel クラスの ClearCommand プロパティを指定しています。

このサンプルプログラムを実行した結果、図 3.12 のように、テキストを入力してから Clear ボタンを押すと、入力したテキストが空になります。



(a) テキストを入力した状態　　(b) Clear ボタンを押すと入力テキストが空になる

図 3.12：ClearCommand の動作確認

コード 3.14 では、DelegateCommand クラスのコンストラクタに入力引数をひとつしか与えていないため、DelegateCommand.CanExecute() メソッドは常に true を返すようになっていました。次のように CanExecute() メソッドの処理を指定することで、ボタンを指定した条件で無効化することができます。

コード 3.16：DelegateCommand に実行可能判別処理を指定する

MainViewModel.cs

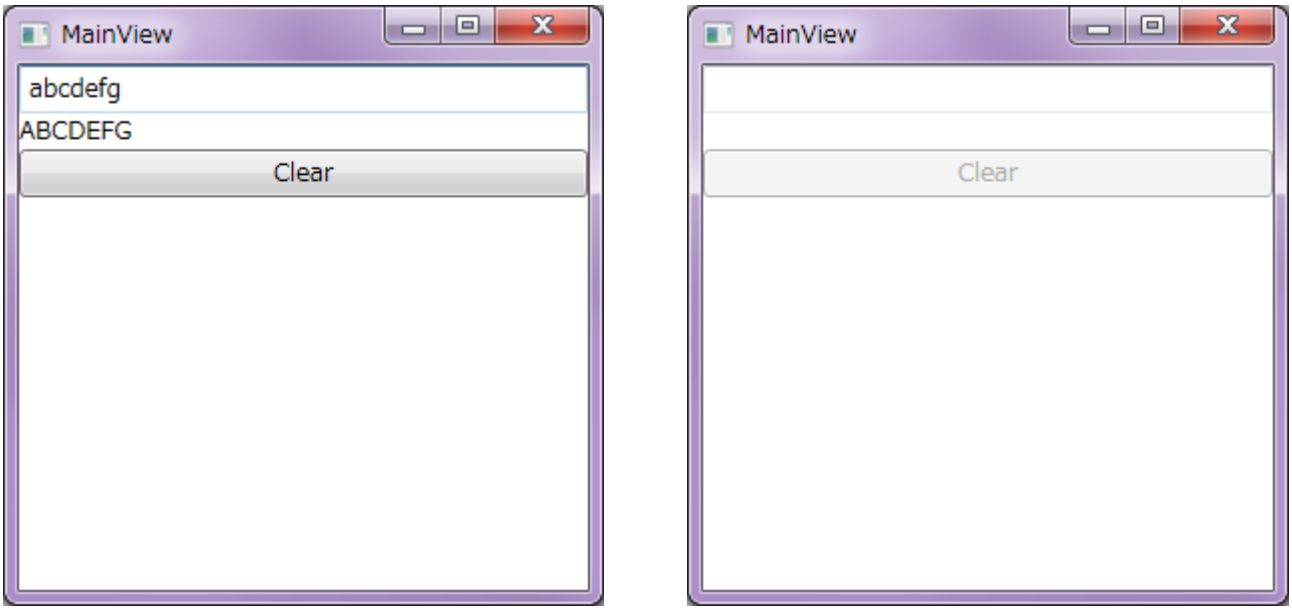
```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      /// <summary>
4      /// MainView ウィンドウに対するデータコンテキストを表します。
5      /// </summary>
6      internal class MainViewModel : NotificationObject
7      {
```

```
 8          private string _upperString;
 9          /// <summary>
10          /// すべて大文字に変換した文字列を取得します。
11          /// </summary>
12          public string UpperString
13          {
14              get { return this._upperString; }
15              private set { SetProperty(ref this._upperString, value); }
16          }
17
18          private string _inputString;
19          /// <summary>
20          /// 入力文字列を取得または設定します。
21          /// </summary>
22          public string InputString
23          {
24              get { return this._inputString; }
25              set
26              {
27                  if (SetProperty(ref this._inputString, value))
28                  {
29                      // 入力された文字列を大文字に変換します
30                      this.UpperString = this._inputString.ToUpper();
31                      // コマンドの実行可能判別結果に影響を与えているので変更通知をおこないます
32                      this.ClearCommand.RaiseCanExecuteChanged();
33
34                      // 出力ウィンドウに結果を表示します
35                      System.Diagnostics.Debug.WriteLine("UpperString=" + this.UpperString);
36                  }
37              }
38          }
39
40          private DelegateCommand _clearCommand;
41          /// <summary>
42          /// クリアコマンドを取得します。
43          /// </summary>
44          public DelegateCommand ClearCommand
45          {
46              get
47              {
48                  if (this._clearCommand == null)
49                  {
50                      this._clearCommand = new DelegateCommand(_ =>
51                      {
52                          this.InputString = "";
53                      },
54                      _ => !string.IsNullOrEmpty(this.InputString));
55                  }
56                  return this._clearCommand;
57              }
58          }
59      }
60  }
```

50 行目の DelegateCommand クラスのコンストラクタで、第 2 引数としてコマンドの実行可能判別処理を指定しています。ここでは 54 行目のように、InputString プロパティが文字列として空でないときに実行可能であるとしています。また、この実行可能判別処理は InputString プロパティが変更されたときに結果が変わるため、32 行目のように、InputString プロパティ変更時に ClearCommand.RaiseCanExecuteChanged() メソッドを呼び出してその変更通知をおこなっています。

このサンプルコードを実行すると、テキストが入力されていない状態のときは Clear ボタンが押せない状態に自動的に切

り替わるようになります。



(a) テキストを入力した状態　　(b) テキストが入力されていない状態

図 3.13：ClearCommand の有効性が自動的に切り替わる

余談ですが、コード 3.16 の 48 行目のように null かどうかを判別し、null なら同じ型で異なる値を返すような場合は null 合体演算子と呼ばれる "??" による記述をおこなうとよりシンプルなコードを書くことができます。コード 3.16 の 40 ～ 58 行目は次のように書き換えることができます。

コード 3.17：DelegateCommand に実行可能判別処理を指定する

MainViewModel.cs

```
40          private DelegateCommand _clearCommand;
41          /// <summary>
42          /// クリアコマンドを取得します。
43          /// </summary>
44          public DelegateCommand ClearCommand
45          {
46              get
47              {
48                  return this._clearCommand ?? (this._clearCommand = new DelegateCommand(
49                  _ => this.InputString = "",
50                  _ => !string.IsNullOrEmpty(this.InputString)));
51              }
52          }
```

"??" 演算子は左側のオペランドが null でなければそれ自体を表し、null の場合は右側のオペランドにある値を表します。ここでは変数 _clearCommand が null のとき、右側オペランドで初期化をおこなうと同時にその値そのものを返すように指定しています。DelegateCommand クラスを使って ICommand インターフェースを実装したクラスを公開するとき、そのコマンドはほとんどの場合動的に変更する必要がないため、コード 3.17 のように "??" 演算子を使った初期化を同時に記述する方法を用いることが一般的です。

3.7 まとめ

WPF はその根底から MVVM パターンをサポートする、Windows 向け GUI アプリケーションを開発するためのフレームワークです。MVVM パターンの要であるデータバインディング機能は View と ViewModel の間のプロパティを同期するための非常に強力で便利なツールです。しかし、ViewModel から View へプロパティの変更を通知する仕組みは開発者側で明示的におこなう必要があります。そのために .NET Framework では INotifyPropertyChanged インターフェースが用意されています。また、View からユーザー操作を ViewModel へ伝えるために ICommand インターフェースが用意されています。これらを実装したクラスを使用することで MVVM パターンのインフラを構築することができます。

本章では INotifyPropertyChanged インターフェースを実装した NotificationObject クラスと、ICommand インターフェースを実装した DelegateCommand クラスのサンプルコードを掲載し、これらによって MVVM パターンで WPF アプリケーションを作成する方法を簡単に説明しました。紹介したサンプルコードのように、ViewModel では View に公開したいプロパティを定義し、そのプロパティを変更するためのメソッドなどを持っています。View は ViewModel が公開しているプロパティをデータバインディング機能で同期させながら、どのようなコントロールでユーザーに表示するかに専念できます。本章では登場しませんでしたが、本来、アプリケーションのコアとなる処理を Model として記述するべきです。本章のサンプルコードでは文字列を大文字に変換する処理がコアな処理となりますが、あまりに機能が小さいため Model を省略していました。このように機能の大小やその他の理由によっては完全に MVVM パターンに当てはめる必要はなく、それぞれのアプリケーションに合ったスタイルで開発すれば良いでしょう。

4 割り算アプリで学ぶ基礎

本章では割り算をするアプリケーションを作成することで MVVM パターンにしたがったアプリケーション開発の基礎を修得します。

4.1 準備

ここでは第 3 章で説明した MVVM パターンを意識した内部構造となるように WPF プロジェクトを新たに作成することを前提として進めていきます。INotifyPropertyChanged インターフェースを実装した NotificationObject クラスとこれを基本クラスとした MainViewModel クラス、ICommand インターフェースを実装した DelegateCommand クラスを作成しておきましょう。また、App.xaml.cs で起動時の処理をオーバーライドし、MainView ウィンドウのデータコンテキストに MainViewModel クラスを指定した状態で MainView ウィンドウを表示するようにしましょう。

準備ができた状態のソリューションエクスプローラーは図 4.1 のようになります。

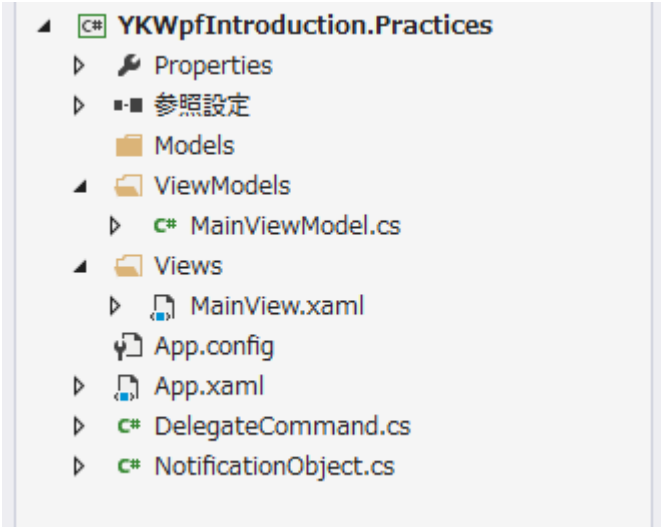


図 4.1：準備できた状態のソリューションエクスプローラー

4.2 UI を作成する

次のような UI を、本章を通して使用する基本的な UI とします。

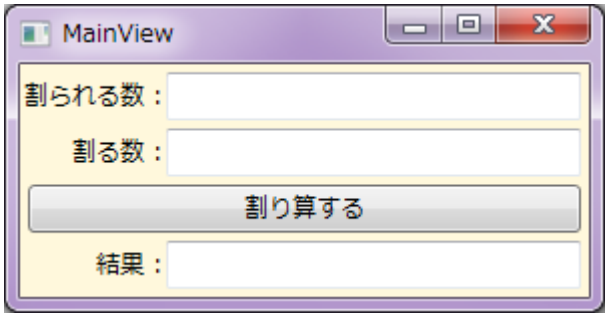


図 4.2：割り算アプリの外観

まずはこの外観を作成していきましょう。まずはコントロールを配置するためのパネルとして Grid パネルを設定します。

コード 4.1：Grid パネルでマス目を作る

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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          Background="Cornsilk">
6      <Grid>
7          <Grid.RowDefinitions>
8              <RowDefinition />
9              <RowDefinition />
10             <RowDefinition />
11             <RowDefinition />
12         </Grid.RowDefinitions>
13         <Grid.ColumnDefinitions>
14             <ColumnDefinition />
15             <ColumnDefinition />
16         </Grid.ColumnDefinitions>
17
       </Grid>
   </Window>
```

Grid パネルを使って領域を格子状に区切ります。RowDefinitions プロパティに RowDefinition クラスを並べた数だけ行数を増やすことができます。列数は ColumnDefinitions プロパティに ColumnDefinition クラスを並べます。ここでは 4 行 2 列になるように設定しています。XAML デザイナー上では図 4.3 のように領域が区切られている様子が表示されます。

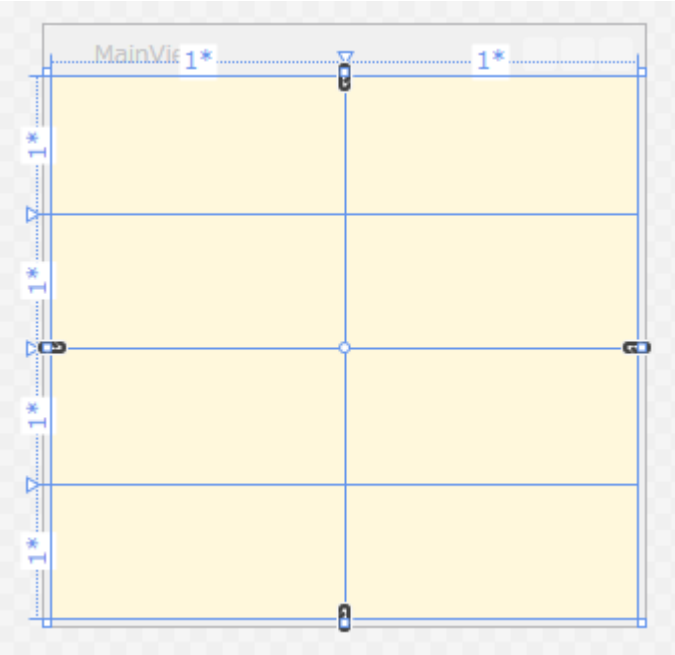


図 4.3：Grid パネルで区切られた領域

それでは、図 4.2 のようにテキストやボタンを配置しましょう。Grid パネルの何行何列にコントロールを配置するかは Grid.Row 添付プロパティと Grid.Column 添付プロパティを使用します。また、複数の行をまたぐように配置するときは Grid.RowSpan 添付プロパティでまたぐ行数を指定します。

コード 4.2：Grid パネルにコントロールを配置する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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          Background="Cornsilk">
6      <Grid>
7          <Grid.RowDefinitions>
8              <RowDefinition />
9              <RowDefinition />
10             <RowDefinition />
11             <RowDefinition />
12         </Grid.RowDefinitions>
13         <Grid.ColumnDefinitions>
14             <ColumnDefinition />
15             <ColumnDefinition />
16         </Grid.ColumnDefinitions>
17
18         <TextBlock Grid.Row="0" Grid.Column="0" Text="割られる数 :" TextAlignment="Right"
   VerticalAlignment="Center" />
19         <TextBox Grid.Row="0" Grid.Column="1" Margin="2" />
20         <TextBlock Grid.Row="1" Grid.Column="0" Text="割る数 :" TextAlignment="Right"
   VerticalAlignment="Center" />
21         <TextBox Grid.Row="1" Grid.Column="1" Margin="2" />
22
23         <Button  Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Content="割り算する" Margin="2" />
24         <TextBlock Grid.Row="3" Grid.Column="0" Text="結果 :" TextAlignment="Right"
   VerticalAlignment="Center" />
25         <TextBox Grid.Row="3" Grid.Column="1" IsReadOnly="True" Margin="2" />
26     </Grid>
27 </Window>
```

図 4.4：コントロールを配置

このまま実行すると、図 4.4 のようにコントロールが配置されます。しかし、配置される位置は正しいですが、コントロールが大きくなっていて不格好です。Grid パネルは特に何も指定しない場合、与えられた領域を等分割で区切るパネルなので、Window いっぱいに広がっている領域を等分割しています。今回は高さを等分割ではなく、配置するコントロールの高さに自動的に合うようにしましょう。

Grid パネルで区切る領域のサイズを指定するときは、RowDefinition クラスの Height プロパティと ColumnDefintion クラスの Width プロパティを使用します。それぞれに "Auto" を指定すると、配置されるコントロールの高さや幅に自動的に合わせてくれるようになります。

コード 4.3：行の高さと列の幅を自動設定にする

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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"
        Background="Cornsilk">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <TextBlock Grid.Row="0" Grid.Column="0" Text="割られる数 :" TextAlignment="Right" VerticalAlignment="Center" />
        <TextBox Grid.Row="0" Grid.Column="1" Margin="2" />
        <TextBlock Grid.Row="1" Grid.Column="0" Text="割る数 :" TextAlignment="Right" VerticalAlignment="Center" />
        <TextBox Grid.Row="1" Grid.Column="1" Margin="2" />

        <Button  Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Content="割り算する" Margin="2" />
        <TextBlock Grid.Row="3" Grid.Column="0" Text="結果 :" TextAlignment="Right" VerticalAlignment="Center" />
        <TextBox Grid.Row="3" Grid.Column="1" IsReadOnly="True" Margin="2" />
    </Grid>
```

```
27 </Window>
```

図 4.5：行の高さと列の幅が自動調整されている

このままでも構いませんが、ウィンドウの高さが余分でスペースが無駄に空いてしまっているので、Window コントロールの SizeToContent プロパティで高さを自動調整するようにしましょう。

コード 4.4：完成

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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          Width="300"
6          SizeToContent="Height"
7          Background="Cornsilk">
8      <Grid Margin="2">
9          <Grid.RowDefinitions>
10             <RowDefinition Height="Auto" />
11             <RowDefinition Height="Auto" />
12             <RowDefinition Height="Auto" />
13             <RowDefinition Height="Auto" />
14         </Grid.RowDefinitions>
15         <Grid.ColumnDefinitions>
16             <ColumnDefinition Width="Auto" />
17             <ColumnDefinition />
18         </Grid.ColumnDefinitions>
19
20         <TextBlock Grid.Row="0" Grid.Column="0" Text="割られる数 :" TextAlignment="Right"
   VerticalAlignment="Center" />
21         <TextBox Grid.Row="0" Grid.Column="1" Margin="2" />
22         <TextBlock Grid.Row="1" Grid.Column="0" Text="割る数 :" TextAlignment="Right"
   VerticalAlignment="Center" />
23         <TextBox Grid.Row="1" Grid.Column="1" Margin="2" />
24
25         <Button  Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Content="割り算する" Margin="2" />
26         <TextBlock Grid.Row="3" Grid.Column="0" Text="結果 :" TextAlignment="Right"
   VerticalAlignment="Center" />
27         <TextBox Grid.Row="3" Grid.Column="1" IsReadOnly="True" Margin="2" />
28     </Grid>
29 </Window>
```

Window コントロールの Height プロパティを削除し、SizeToContent プロパティに "Height" を指定することでウィンドウの高さをコンテンツの高さに自動調整してくれます。

以上で割り算アプリの外観のベースが完成しました。次節以降では、この UI をアレンジしながら進めてます。

4.3 MainViewModel クラスのプロパティと同期する

前節で作成した UI を使って、MainViewModel クラスに追加するプロパティと同期するように設定しましょう。

まず、割られる数と割る数と計算結果を表示するための TextBox コントロールの Text プロパティを MainViewModel のプロパティと同期するようにデータバインディングすることを考えます。MainViewModel には次のようなプロパティを追加します。

コード 4.5：各プロパティを追加した MainViewModel クラス

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      /// <summary>
4      /// MainView ウィンドウに対するデータコンテキストを表します。
5      /// </summary>
6      internal class MainViewModel : NotificationObject
7      {
8          private string _lhs;
9          /// <summary>
10         /// 割られる数に指定される文字列を取得または設定します。
11         /// </summary>
12         public string Lhs
13         {
14             get { return this._lhs; }
15             set { SetProperty(ref this._lhs, value); }
16         }
17
18         private string _rhs;
19         /// <summary>
20         /// 割る数に指定しされる文字列を取得または設定します。
21         /// </summary>
22         public string Rhs
23         {
24             get { return this._rhs; }
25             set { SetProperty(ref this._rhs, value); }
26         }
27
28         private string _result;
29         /// <summary>
30         /// 計算結果を文字列として取得します。
31         /// </summary>
32         public string Result
33         {
34             get { return this._result; }
35             private set { SetProperty(ref this._result, value); }
36         }
37     }
38 }
```

コメントにある通り、割られる数に Lhs、割る数に Rhs、計算結果に Result プロパティがそれぞれ同期するようにします。実際に XAML のほうでデータバインディングすると次のようになります。

コード 4.6：MainViewModel クラスの各プロパティと同期するように設定する

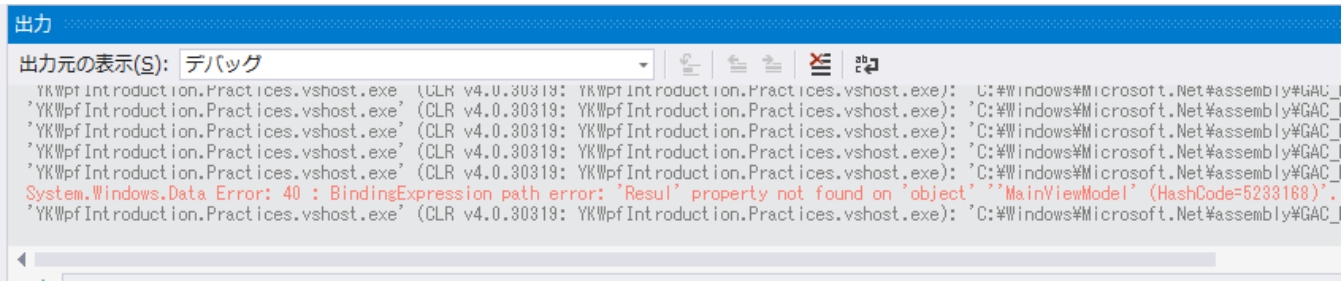
MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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          Width="300"
6          SizeToContent="Height"
```

```
 7         Background="Cornsilk">
 8     <Grid>
 9         <Grid.RowDefinitions>
10             <RowDefinition Height="Auto" />
11             <RowDefinition Height="Auto" />
12             <RowDefinition Height="Auto" />
13             <RowDefinition Height="Auto" />
14         </Grid.RowDefinitions>
15         <Grid.ColumnDefinitions>
16             <ColumnDefinition Width="Auto" />
17             <ColumnDefinition />
18         </Grid.ColumnDefinitions>
19
20         <TextBlock Grid.Row="0" Grid.Column="0" Text="割られる数 :" TextAlignment="Right" VerticalAlignment="Center" />
21         <TextBox Grid.Row="0" Grid.Column="1" Text="{Binding Lhs}" Margin="2" />
22         <TextBlock Grid.Row="1" Grid.Column="0" Text="割る数 :" TextAlignment="Right" VerticalAlignment="Center" />
23         <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Rhs}" Margin="2" />
24
25         <Button  Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Content="割り算する" Margin="2" />
26         <TextBlock Grid.Row="3" Grid.Column="0" Text="結果 :" TextAlignment="Right" VerticalAlignment="Center" />
27         <TextBox Grid.Row="3" Grid.Column="1" Text="{Binding Result, Mode=OneWay}" IsReadOnly="True" Margin="2" />
28     </Grid>
29 </Window>
```

21、23、27 行目にある TextBox コントロールの Text プロパティにそれぞれデータバインディング機能によって MainViewModel クラスが持つプロパティと同期するように指定しています。また、27 行目の計算結果を表示するための TextBox コントロールは IsReadOnly プロパティに true を指定することで読取専用としているため、Text プロパティを変更することが許されません。したがって、データバインディング機能による同期設定をおこなうとき、Mode に "OneWay" を設定する必要があります。これは、データコンテキストからの変更にのみ同期し、UI からの変更には同期しないという設定になります。特に指定しない場合は "TwoWay" というモードで、双方向で同期するようになっています。"OneWay" とは逆に、UI からの変更にのみ同期する "OneWayToSource" というモードもあります。

このコードでそのまま実行しても見た目は前節と変わりません。しかし、もしデータバインディングの設定でプロパティ名を間違ったまま実行すると、Visual Studio の出力ウィンドウに "BindingExpression path error" というエラーメッセージが表示されるので、こういったメッセージが表示されないことを確認しておいても良いでしょう。



```
 'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshost.exe): 'C:¥Windows¥Microsoft.Net¥assembly¥GAC_
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshost.exe): 'C:¥Windows¥Microsoft.Net¥assembly¥GAC_
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshost.exe): 'C:¥Windows¥Microsoft.Net¥assembly¥GAC_
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshost.exe): 'C:¥Windows¥Microsoft.Net¥assembly¥GAC_
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshost.exe): 'C:¥Windows¥Microsoft.Net¥assembly¥GAC_
System.Windows.Data Error: 40 : BindingExpression path error: 'Resul' property not found on 'object' ''MainViewModel' (HashCode=5233168)'.
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshost.exe): 'C:¥Windows¥Microsoft.Net¥assembly¥GAC_
```

図 4.6：プロパティ名を間違うとエラーメッセージが表示される

さて、実際に割り算をするにはそのきっかけとなるコマンドが必要です。ここでは「割り算する」ボタンを押すことで割り算を実行し、計算結果を表示するようにします。そのために、MainViewModel クラスに DelegateCommand クラスのプロパティを準備します。

コード 4.7：割り算コマンドを追加

MainViewModel.cs

```
namespace YKWpfIntroduction.Practices.ViewModels
{
    /// <summary>
    /// MainView ウィンドウに対するデータコンテキストを表します。
```

```
 5      /// </summary>
 6      internal class MainViewModel : NotificationObject
 7      {
 8          private string _lhs;
 9          /// <summary>
10          /// 割られる数に指定される文字列を取得または設定します。
11          /// </summary>
12          public string Lhs
13          {
14              get { return this._lhs; }
15              set { SetProperty(ref this._lhs, value); }
16          }
17
18          private string _rhs;
19          /// <summary>
20          /// 割る数に指定しされる文字列を取得または設定します。
21          /// </summary>
22          public string Rhs
23          {
24              get { return this._rhs; }
25              set { SetProperty(ref this._rhs, value); }
26          }
27
28          private string _result;
29          /// <summary>
30          /// 計算結果を文字列として取得します。
31          /// </summary>
32          public string Result
33          {
34              get { return this._result; }
35              private set { SetProperty(ref this._result, value); }
36          }
37
38          private DelegateCommand _divCommand;
39          /// <summary>
40          /// 割り算コマンドを取得します。
41          /// </summary>
42          public DelegateCommand DivCommand
43          {
44              get
45              {
46                  return this._divCommand ?? (this._divCommand = new DelegateCommand(
47                  _ =>
48                  {
49                      OnDivision();
50                  }));
51              }
52          }
53
54          /// <summary>
55          /// 割り算を実行します。
56          /// </summary>
57          private void OnDivision()
58          {
59              throw new System.NotImplementedException();
60          }
61      }
62  }
```

42 行目に割り算を実行するためのコマンドを用意し、実行されたときに処理されるメソッドを 57 行目に用意しています。

まだ割り算の処理を実装していないので、もし実行された場合は未実装例外 NotImplementedException を発生させるようにしています。

ワンポイント 4.1：メソッドなどの自動生成機能を利用する

Visual Studio では、未定義のメソッドやプロパティを自動的に生成する機能があります。例えばコード 4.7 では新たに OnDivision() メソッドを追加しています。このとき、通常は 57 行目のメソッドの定義を書いてから 49 行目の呼び出しを記述しますが、あえて 57 行目の定義をする前に唐突に 49 行目で OnDivision() メソッドを呼び出そうとしてみてください。すると、Intellisense 機能によって "OnDivision" に下線が表示されてしまいますが、"OnDivision" の文字列の上にキーボードカーソルがある状態でマウスカーソルを近づけると、文字列周辺にアイコンが表示されます。このアイコンをクリックすると、図 4.7 のように定義されていないものへの対処方法が表示されます。今回は未定義のメソッドを呼び出そうとしているコードへの対処法なので、「'OnDivision' のメソッドスタブを生成します」というメニューが表示されています。これをクリックすると、コード 4.7 の 57 行目のように OnDivision() メソッドが自動的に追加されます。

Visual Studio で C# コーディングをおこなうときは、この機能を良く活用します。わざわざメソッドやプロパティを定義してからコードを書くのではなく、コードを書いている中で未定義のメソッドやプロパティをこの機能で追加していくスタイルは慣れると効率が良くなります。

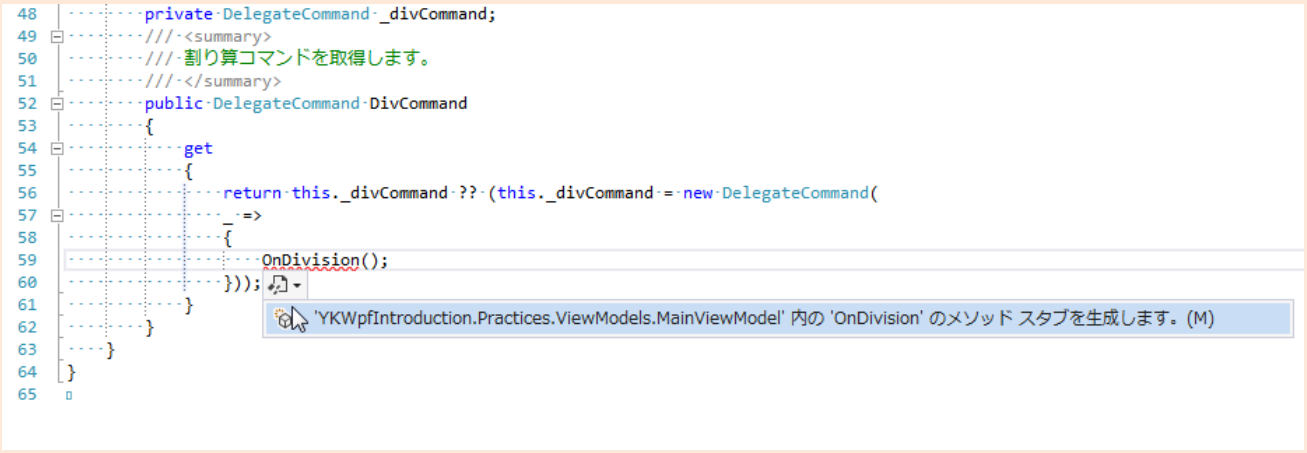


図 4.7：メソッドを自動生成するメニューが表示される

用意した割り算コマンドを UI と紐付けるために、MainView ウィンドウの XAML で Button コントロールの Command プロパティを次のように編集しましょう。

コード 4.8：割り算コマンドをデータバインディング機能で同期する

MainView.xaml

```
25          <Button  Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Content="割り算する"
Command="{Binding DivCommand}" Margin="2" />
```

ここまでのコードでいったん実行してみましょう。見た目は図 4.2 と変わりませんが、「割り算する」ボタンを押すと、コード 4.7 の 63 行目で停止することを確認します。ここで停止するということは、ボタンと DivCommand がきちんと紐付いているということです。

4.4 割り算をおこなうクラスを作成する

今回のアプリケーションは「割り算をおこなう」アプリケーションです。この「割り算をおこなう」処理は UI とは切り離して考えることができます。したがって、割り算をおこなうためのクラスを用意し、これを Model として扱うようにします。

割り算をおこなうためのクラスとして Calculator クラスを作成し、次のように定義しましょう。

コード 4.9：Calculator クラスの定義

Calculator.cs

```
1  namespace YKWpfIntroduction.Practices.Models
2  {
3      internal class Calculator
4      {
5          /// <summary>
6          /// 被演算項を取得または設定します。
7          /// </summary>
8          public double Lhs { get; set; }
9
10         /// <summary>
11         /// 演算項を取得または設定します。
12         /// </summary>
13         public double Rhs { get; set; }
14
15         /// <summary>
16         /// 計算結果を取得します。
17         /// </summary>
18         public double Result { get; private set; }
19
20         /// <summary>
21         /// 割り算をおこないます。
22         /// </summary>
23         public void ExecuteDiv()
24         {
25             this.Result = this.Lhs / this.Rhs;
26         }
27     }
28 }
```

割られる数を Lhs、割る数を Rhs、計算結果を Result プロパティとし、ExecuteDiv() メソッドを実行すると現在の状態での計算結果を Result プロパティに代入しています。ただし、MainViewModel クラスとは違い、それぞれのプロパティは string 型ではなく double 型となっています。このクラスでは実際に数値を計算することが目的なので、各プロパティは double 型で持つべきだからです。

ExecuteDiv() メソッドはこのクラスの内部状態を外部から操作させるためのメソッドです。割り算をおこなえばいいので "public double ExecuteDiv(double x, double y)" というようなメソッドを考えがちですが、それは同一クラス内で内部状態を変更するときの private メソッドとして扱うべきで、ここでは外部から内部状態を操作させるためのメソッドを定義するため、このように入力引数も戻り値もないメソッドを定義しています。

ワンポイント 4.2：Model の公開メソッドは内部状態を変更するためのもの

ExecuteDiv() メソッドは入力引数や戻り値を持っていません。これは ViewModel など外部からアクセスされることを想定しているからです。

Model を外部から操作する場合、「現在の状態で処理をおこなう」ことが基本的な考え方であり、入力引数はあくまでもそれを補助するためのパラメータに過ぎません。また、公開メソッドはそのクラスの内部状態を変更するための操作であって、そのメソッドが戻り値を持つということは基本的にはあり得ません。メソッドによって内部状態を変更し、その結果は変更されるべき状態から確認できるからです。

4.5 ViewModel から Model を操作する

前節で作成した Calculator クラスを MainViewModel クラスから操作するようにしましょう。

コード 4.10：Calculator クラスで割り算をおこなう

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      using YKWpfIntroduction.Practices.Models;
4
5      /// <summary>
6      /// MainView ウィンドウに対するデータコンテキストを表します。
7      /// </summary>
8      internal class MainViewModel : NotificationObject
9      {
10         /// <summary>
11         /// 新しいインスタンスを生成します。
12         /// </summary>
13         public MainViewModel()
14         {
15             this._calc = new Calculator();
16         }
17
18         private string _lhs;
19         /// <summary>
20         /// 割られる数に指定される文字列を取得または設定します。
21         /// </summary>
22         public string Lhs
23         {
24             get { return this._lhs; }
25             set { SetProperty(ref this._lhs, value); }
26         }
27
28         private string _rhs;
29         /// <summary>
30         /// 割る数に指定しされる文字列を取得または設定します。
31         /// </summary>
32         public string Rhs
33         {
34             get { return this._rhs; }
35             set { SetProperty(ref this._rhs, value); }
36         }
37
38         private string _result;
39         /// <summary>
40         /// 計算結果を文字列として取得します。
41         /// </summary>
42         public string Result
43         {
44             get { return this._result; }
45             private set { SetProperty(ref this._result, value); }
46         }
47
48         private DelegateCommand _divCommand;
49         /// <summary>
50         /// 割り算コマンドを取得します。
51         /// </summary>
52         public DelegateCommand DivCommand
53         {
54             get
55             {
```

```
56                  return this._divCommand ?? (this._divCommand = new DelegateCommand(
57                  _ =>
58                  {
59                      OnDivision();
60                  }));
61              }
62          }
63
64          /// <summary>
65          /// 割り算を実行します。
66          /// </summary>
67          private void OnDivision()
68          {
69              this._calc.ExecuteDiv();
70              this.Result = this._calc.Result.ToString();
71          }
72
73          /// <summary>
74          /// 計算をおこなうオブジェクト
75          /// </summary>
76          private Calculator _calc;
77      }
78  }
```

Model となる Calculator クラスを 76 行目で定義した private フィールドで保持するようにし、13 行目のコンストラクタ内で初期化をおこなっています。

そして、67 行目の割り算が実行されるメソッド内で、実際に Calculator クラスで計算を実行させています。また、割り算を実行した後、70 行目でその計算結果を MainViewModel クラスの Result プロパティに代入しています。このように、Model が持っている状態を View へ公開するためにその値または型を変換・保持することが ViewModel の役割となります。

それでは一度実行してみましょう。アプリケーション起動後に「割り算する」ボタンを押してみてください。

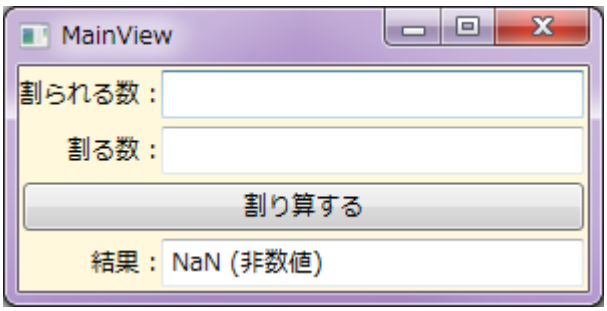


図 4.8：割り算結果が Result プロパティに反映されている

Calculator クラスは new でインスタンス化しただけなので、その内部状態は Lhs == 0、Rhs == 0 という状態となっています。そのまま割り算を実行しているため、0 ÷ 0 の計算をおこない、結果 "NaN" となっています。

それでは、MainViewModel クラスが持っている割られる数と割る数の文字列を数値に変換し、これを用いて割り算するように OnDivision() メソッドを次のように書き換えましょう。

コード 4.11：string 型を double 型に変換する

MainViewModel.cs

```
64          /// <summary>
65          /// 割り算を実行します。
66          /// </summary>
67          private void OnDivision()
68          {
69              this._calc.Lhs = double.Parse(this.Lhs);
70              this._calc.Rhs = double.Parse(this.Rhs);
71              this._calc.ExecuteDiv();
72              this.Result = this._calc.Result.ToString();
```

```
73          }
```

文字列を数値に変換するときは、目的とする数値型が持っている Parse() メソッドを使います。今回は double 型に変換したいので、double.Parse() メソッドです。これを使って割られる数と割る数の文字列を数値に変換し、Calculator クラスの各プロパティに代入し、それから割り算を実行するようにしています。

もう一度実行してみましょう。割られる数と割る数にそれぞれ数値を入力してから「割り算する」ボタンを押すと、割り算の結果がきちんと表示されるでしょうか。

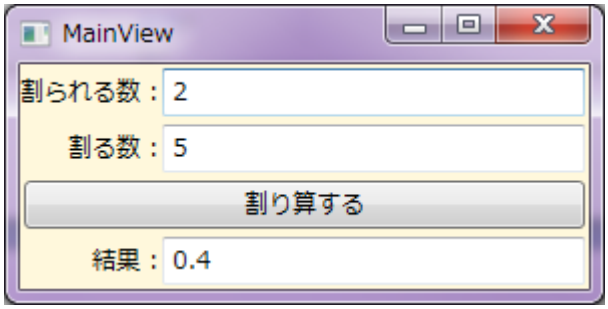


図 4.9：割り算が実行されるようになる

数値がきちんと入力されていれば図 4.9 のように計算結果が正常に表示されるようになりました。しかし、どちらかを入力しなかったり、数値以外の書式を入力したりした状態で「割り算する」ボタンを押すと、double.Parse() メソッドで FormatException という例外が発生してしまいます。つまり、double 型の数値に変換できない書式の文字列が指定されてしまったため、エラーが発生しています。

double.Parse() メソッドの入力引数は、きちんと数値に変換できる書式であることを保証した文字列を与えるようにしないと、このように例外が発生してしまいます。入力される文字列が数値に変換できない場合があることを考慮するときは double.TryParse() メソッドを使いましょう。修正したコードは次のようになります。

コード 4.12：TryParse() メソッドで string 型を数値に変換する

MainViewModel.cs

```
64          /// <summary>
65          /// 割り算を実行します。
66          /// </summary>
67          private void OnDivision()
68          {
69              var lhs = 0.0;
70              var rhs = 0.0;
71              if (!double.TryParse(this.Lhs, out lhs))
72              {
73                  return;
74              }
75              if (!double.TryParse(this.Rhs, out rhs))
76              {
77                  return;
78              }
79              this._calc.Lhs = lhs;
80              this._calc.Rhs = rhs;
81              this._calc.ExecuteDiv();
82              this.Result = this._calc.Result.ToString();
83          }
```

double.TryParse() メソッドは、第 1 引数に変換元の文字列、第 2 引数に変換後の値を代入する変数を指定します。変換後の値はメソッド内で更新された内容を呼び出し元にも反映させるため、初期化済みの変数を out キーワードを付けて指定する必要があります。もし数値に変換出来た場合は戻り値に true が返ってきますが、変換できなかった場合は false が返ってきます。上記のサンプルでは変換できなかった場合は return するようにし、以降の処理をおこなわないようにしています。

これでもう一度実行してみましょう。すると、割られる数や割る数に数値以外の書式の文字列を入力した状態で「割り算する」ボタンを押しても何も処理されず、計算結果には以前の結果が残ったままとなります。

ところで、数値以外のものが入力されたときは「割り算する」ボタンが無効となればエラーが発生することはありません。「割り算する」ボタンには割り算コマンドである DivCommand プロパティが紐付けられているので、このコマンドの実行可能判別処理を追加しましょう。

コード 4.13：DivCommand プロパティに実行可能判別処理を追加する

MainViewModel.cs

```
 1  namespace YKWpfIntroduction.Practices.ViewModels
 2  {
 3      using YKWpfIntroduction.Practices.Models;
 4
 5      /// <summary>
 6      /// MainView ウィンドウに対するデータコンテキストを表します。
 7      /// </summary>
 8      internal class MainViewModel : NotificationObject
 9      {
10          /// <summary>
11          /// 新しいインスタンスを生成します。
12          /// </summary>
13          public MainViewModel()
14          {
15              this._calc = new Calculator();
16          }
17
18          private string _lhs;
19          /// <summary>
20          /// 割られる数に指定される文字列を取得または設定します。
21          /// </summary>
22          public string Lhs
23          {
24              get { return this._lhs; }
25              set
26              {
27                  if (SetProperty(ref this._lhs, value))
28                  {
29                      this.DivCommand.RaiseCanExecuteChanged();
30                  }
31              }
32          }
33
34          private string _rhs;
35          /// <summary>
36          /// 割る数に指定しされる文字列を取得または設定します。
37          /// </summary>
38          public string Rhs
39          {
40              get { return this._rhs; }
41              set
42              {
43                  if (SetProperty(ref this._rhs, value))
44                  {
45                      this.DivCommand.RaiseCanExecuteChanged();
46                  }
47              }
48          }
49
50          private string _result;
51          /// <summary>
52          /// 計算結果を文字列として取得します。
53          /// </summary>
```

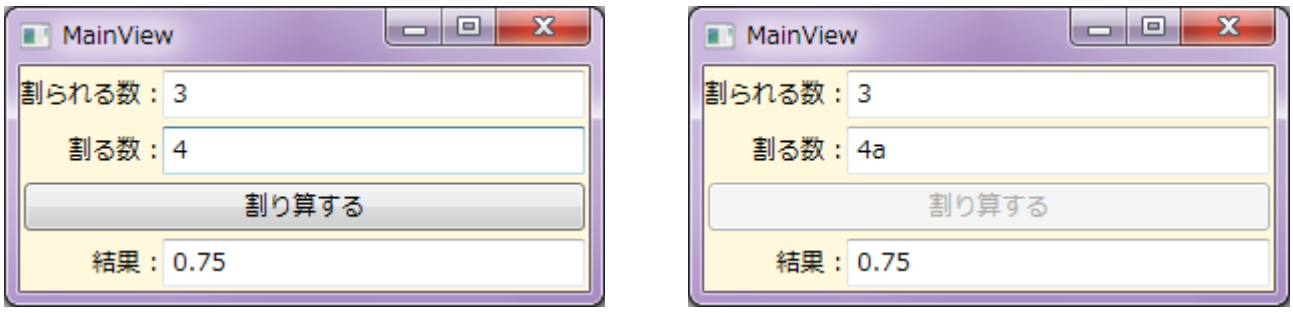
```
54          public string Result
55          {
56              get { return this._result; }
57              private set { SetProperty(ref this._result, value); }
58          }
59
60          private DelegateCommand _divCommand;
61          /// <summary>
62          /// 割り算コマンドを取得します。
63          /// </summary>
64          public DelegateCommand DivCommand
65          {
66              get
67              {
68                  return this._divCommand ?? (this._divCommand = new DelegateCommand(
69                  _ =>
70                  {
71                      OnDivision();
72                  },
73                  _ =>
74                  {
75                      var dummy = 0.0;
76                      if (!double.TryParse(this.Lhs, out dummy))
77                      {
78                          return false;
79                      }
80                      if (!double.TryParse(this.Rhs, out dummy))
81                      {
82                          return false;
83                      }
84                      return true;
85                  }));
86              }
87          }
88
89          /// <summary>
90          /// 割り算を実行します。
91          /// </summary>
92          private void OnDivision()
93          {
94              this._calc.Lhs = double.Parse(this.Lhs);
95              this._calc.Rhs = double.Parse(this.Rhs);
96              this._calc.ExecuteDiv();
97              this.Result = this._calc.Result.ToString();
98          }
99
100         /// <summary>
101         /// 計算をおこなうオブジェクト
102         /// </summary>
103         private Calculator _calc;
104     }
105 }
```

DivCommand プロパティは DelegateCommand クラスで、「3.6 ICommand インターフェースの実装」で説明したように、このクラスは実行可能判別処理 CanExecute() メソッドを持っています。実行可能判別処理で実行不可となった場合、紐付けられているコントロールが自動的に無効になるようになります。コード 4.13 では実行可能判別処理として、75 ～ 84 行目のように、割られる数の文字列と割る数の文字列が double 型に変換できるかどうかを検証し、変換できれば実行可能、変換できなければ実行不可とする処理を追加しています。また、この実行可能判別処理は Lhs プロパティまたは Rhs プロパティが変更されたときに結果が変わる可能性があるため、29 行目と 45 行目のそれぞれのプロパティが変更されるタイミングで DivCommand.RaiseCanExecuteChanged() メソッドを呼び出し、実行可能判別の結果が変更されることを通知して

います。

この実行可能判別処理を追加することで、OnDivision() メソッドが実行されるときは必ず数値に変換できる文字列となっていることが保証されるため、double.TryParse() メソッドではなく、double.Parse() メソッドにしています。ただし、これはあくまでもサンプルアプリケーションなので Parse() にあえて戻していますが、実際のアプリケーションでは想定外の呼び出しにも対応できるように TryParse() メソッドを使うことを強く推奨します。

上記の修正をしてからアプリケーションを実行してみましょう。すると、割られる数や割る数に数値を入力すると「割り算する」ボタンが有効化し、数値以外を入力すると無効化します。



(a) きちんと数値を入力した場合　(b) 数値以外を入力した場合

図 4.10：ボタンの有効性が自動的に切り替わる

しかし、実際に触ってみるとわかると思いますが、ボタンの有効性が切り替わるタイミングが TextBox コントロールからフォーカスを移動させたときとなっています。これはデータバインディング機能で UI からデータコンテキストへ変更を通知するタイミングがデフォルトで LostFocus、すなわちフォーカスを失ったときになっているためです。文字列を入力している最中にボタンの有効性を切り替えたい、すなわちデータコンテキストへ変更を通知させたいときは、次のようにデータバインディングを指定するときに UpdateSourceTrigger を指定します。

コード 4.14：変更通知をプロパティ変更時に指定する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView"
        Width="300"
        SizeToContent="Height"
        Background="Cornsilk">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <TextBlock Grid.Row="0" Grid.Column="0" Text="割られる数 :" TextAlignment="Right" VerticalAlignment="Center" />
        <TextBox Grid.Row="0" Grid.Column="1" Text="{Binding Lhs, UpdateSourceTrigger=PropertyChanged}" Margin="2" />
        <TextBlock Grid.Row="1" Grid.Column="0" Text="割る数 :" TextAlignment="Right" VerticalAlignment="Center" />
        <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Rhs, UpdateSourceTrigger=PropertyChanged}" Margin="2" />

        <Button  Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Content="割り算する" Command="{Binding DivCommand}" Margin="2" />
```

```
26          <TextBlock Grid.Row="3" Grid.Column="0" Text="結果 :" TextAlignment="Right"
    VerticalAlignment="Center" />
27          <TextBox Grid.Row="3" Grid.Column="1" Text="{Binding Result, Mode=OneWay}" IsReadOnly="True"
    Margin="2" />
28      </Grid>
29  </Window>
```

21 行目と 23 行目で、割られる数および割る数に対する文字列にデータバインディングを指定するときに、UpdateSourceTrigger に PropertyChanged を指定しています。これは、データバインディングするプロパティ、ここでは TextBox コントロールの Text プロパティが変更されたときにデータコンテキストへ変更を通知するように指定していることになります。

このように UpdateSourceTrigger に PropertyChnaged を指定すると、割られる数や割る数を入力している最中に「割り算する」ボタンの有効性が自動的に切り替わるようになります。

4.6 まとめ

本章で作成した割り算アプリを通して、MVVM パターンにしたがってどのようにアプリケーションを開発するかが少しずつ見えてきたのではないでしょうか。

View に該当するウィンドウでは、とにかくユーザーに対してどのように見せるかを考えてコントロールを配置し、必要な情報は ViewModel のプロパティと同期させます。

ViewModel に該当するクラスでは、View と同期するためのプロパティを定義し、Model に該当するクラスのインスタンスを保持します。そして View からの操作指示を受け取るコマンドを定義し、その中で Model を操作します。操作した結果もまた View へ公開しているプロパティへ適切に変換して受け渡します。

Model に該当するクラスでは、View や ViewModel に依存することなくただ淡々と処理をおこないます。そのための内部状態を保持し、現在の状態における操作を ViewModel に公開するようにします。

MVVM パターンにしたがってアプリケーション開発をおこなうことは責務の細分化という観点から見ればとても理に適った方法ですが、あまりにも縛られてしまうと自由にコーディングできなかったり、無駄な処理が増えてオーバーヘッドが増大したりすることもあります。MVVM パターンはひとつの理想形として捉えるくらいで良いでしょう。

5 メニューとステータスバーで学ぶ基礎

本章ではメニューとステータスバーを備えた、よく見るアプリケーションをどのように構築していくかを修得します。第 3 章で紹介したように MVVM パターンを意識した内部構造となるように WPF アプリケーションプロジェクトを新たに作成してから進めてください。

5.1 メニューとステータスバー

アプリケーションにはよく「ファイル」や「ヘルプ」などのメニューがウィンドウ上部に表示されています。WPF ではこれを Menu コントロールで簡単に配置することができます。また、アプリケーションの状態を表すためのステータスバーがウィンドウ下部に表示されていることもあります。こちらは StatusBar コントロールを使います。これらを配置したサンプルは次のようになります。本章ではこのようなアプリケーションを作成していきます。

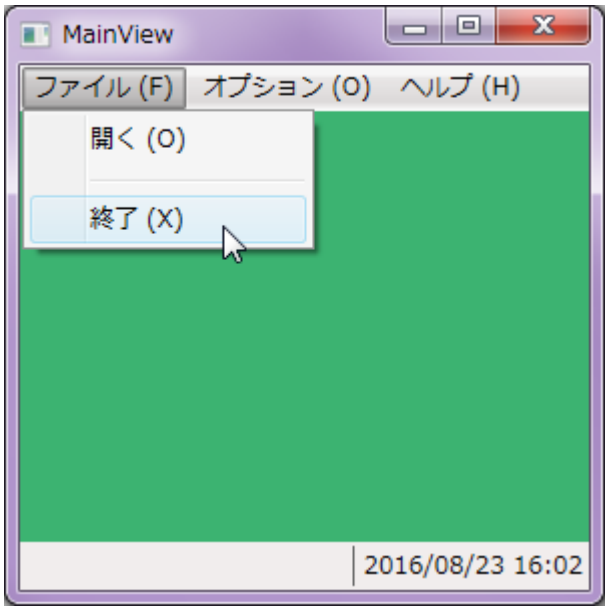


図 5.1：メニューとステータスバーを備えたアプリの外観

5.2 UI を作成する

まずは図 5.1 の外観を作成しましょう。完成形をよく見ると、メニューが上部に張り付いてステータスバーが下部に貼り付き、残りの領域がコンテンツ領域となっています。こういう配置には DockPanel パネルが最適です。まず DockPanel パネルの基本的な挙動を確認するために次のように XAML を編集してみましょう。

コード 5.1：DockPanel パネルに Button コントロールを配置

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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="上" />
7          <Button DockPanel.Dock="Left" Content="左" />
8          <Button Content="残りの領域" />
9      </DockPanel>
10 </Window>
```

DockPanel パネルでは、子要素の配置を DockPanel.Dock 添付プロパティで指定します。上記のコードでは「上」ボタンを "Top"、「左」ボタンを "Left" として配置し、特に配置を指定しないボタンを最後に配置しています。すると図 5.2 のように、DockPanel.Dock 添付プロパティを指定されたコントロールはその方向に張り付くように配置され、最後のボタンは残りの領域を占有するように配置されました。

図 5.2：コントロールが順番に指定位置に張り付く

ここで、コード 5.1 の 6 行目と 7 行目を入れ替え、次のようなコードにして見てください。

コード 5.2：DockPanel パネル内の Button コントロールの順序を変更する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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="Left" Content="左" />
7          <Button DockPanel.Dock="Top" Content="上" />
8          <Button Content="残りの領域" />
9      </DockPanel>
10 </Window>
```

すると図 5.3 のように「左」ボタンがウィンドウ上端まで到達するように配置されていることが分かります。これは、XAML に記述した順番通りにコントロールを配置するからです。コード 5.1 では「上」ボタンを最初に配置しているため、ウィンドウいっぱいの領域に対して上端に張り付くように配置されています。これに対してコード 5.2 では、「左」ボタンが最初に配置され、その次に「上」ボタンが配置されているため、「上」ボタンは「左」ボタンが配置された後の残った領域の上端に配置されるようになっています。

図 5.3：順番を入れ替えると配置が変わる

このように、DockPanel パネルはコントロールを順番に端に配置するものです。このようなパネルを使って Menu コントロールと StatusBar コントロールを配置しましょう。

コード 5.3：メニューとステータスバーを持つアプリの外観の基本形

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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          <Menu DockPanel.Dock="Top">
7          </Menu>
8
9          <StatusBar DockPanel.Dock="Bottom">
10         </StatusBar>
11
12         <Grid Background="MediumSeaGreen">
13         </Grid>
14     </DockPanel>
15 </Window>
```

このような外観を定義して実行すると次のようにメニューとステータスバーが存在しないような UI になります。これは Menu コントロールや StatusBar コントロールに対して子要素をひとつも配置していないからです。それぞれの子要素の配置方法については以降で説明します。

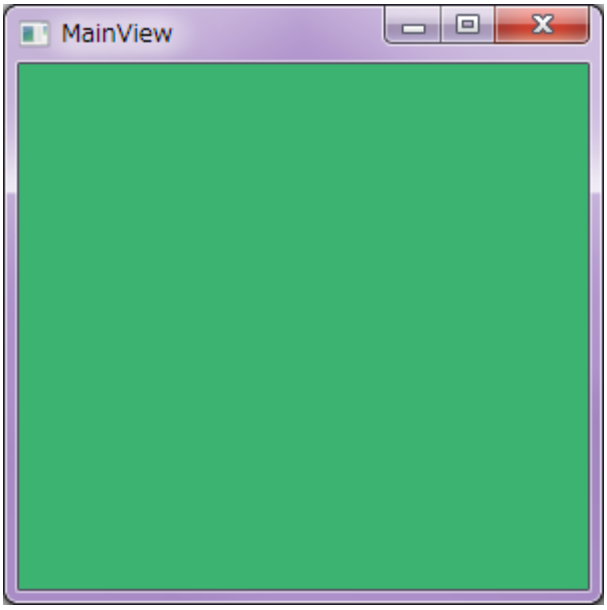


図 5.4：メニューとステータスバーを持っているはずの UI

5.3 「開く」メニューからファイルを開く

よくファイルを開くために図 5.5 のようなコモンダイアログが使用されます。WPF からもファイルを開いたり保存したりするときにこのコモンダイアログが使用できればファイルアクセスが非常に簡単になります。WPF でこのようなコモンダイアログを呼び出すには、Microsoft.Win32.OpenFileDialog クラスや Microsoft.Win32.SaveFileDialog クラスをインスタンス化し、それぞれ ShowDialog() メソッドを呼び出す必要があります。

ところで、ダイアログを表示するという行為は MVVM パターンでは View の管轄となります。そこで、ここでは View 側のコードでダイアログを開き、その結果を ViewModel 側で受け取る仕組みについて紹介します。

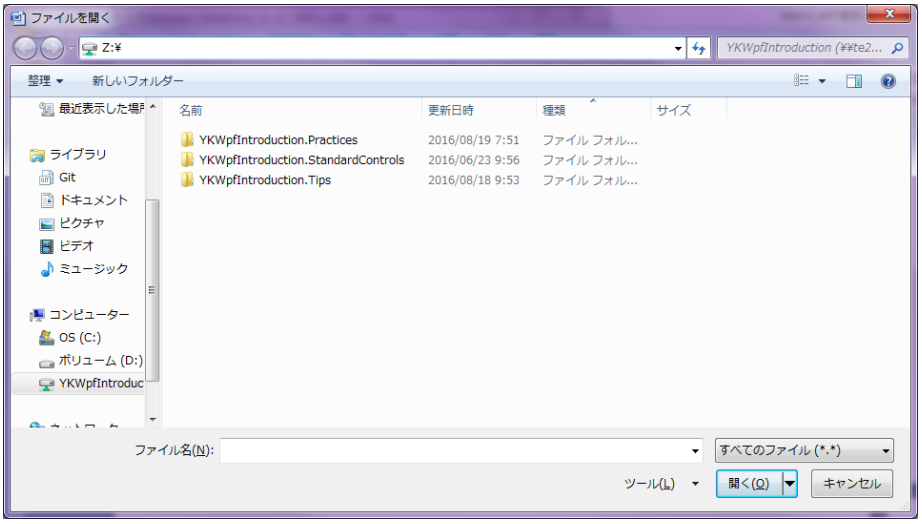


図 5.5：ファイルを開くためのコモンダイアログ

5.3.1 「ファイル」メニューと「開く」メニューの作り方

Menu コントロールにメニューを追加するときは MenuItem コントロールを子要素に配置します。例えば「ファイル」メニューと「ヘルプ」メニューを置く場合は次のように並べて記述します。

コード 5.4：メニュー項目を追加する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="ファイル (_F)" />
            <MenuItem Header="ヘルプ (_H)" />
        </Menu>

        <StatusBar DockPanel.Dock="Bottom">
        </StatusBar>

        <Grid Background="MediumSeaGreen">
        </Grid>
    </DockPanel>
</Window>
```

図 5.6：メニューの追加

メニューの項目名は Header プロパティに指定します。このとき、"(_F)" のようにアルファベットの手前にアンダースコア "_" を記述することで、キーボード操作するときのショートカットキーを指定することができます。このサンプルでは「ファイル」メニューに対して "F" キーを割り当てているため、Alt+F キーを押すと「ファイル」メニューが選択されるようになります。

「ファイル」メニューのサブメニューとして「開く」メニューを追加するときは、MenuItem コントロールを入れ子にして記述します。

コード 5.5：サブメニュー項目を追加する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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">
    <DockPanel>
```

```
6         <Menu DockPanel.Dock="Top">
7             <MenuItem Header="ファイル (_F)">
8                 <MenuItem Header="開く(_O)" />
9             </MenuItem>
10            <MenuItem Header="ヘルプ (_H)" />
11        </Menu>
12
13        <StatusBar DockPanel.Dock="Bottom">
14        </StatusBar>
15
16        <Grid Background="MediumSeaGreen">
17        </Grid>
18    </DockPanel>
19</Window>
```

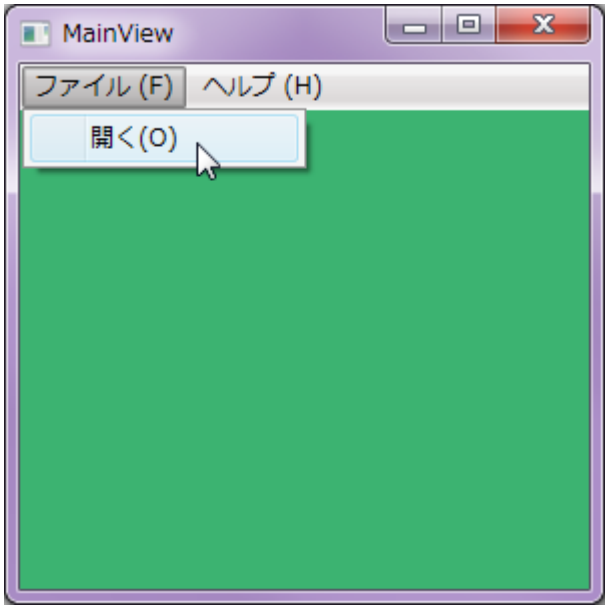


図 5.7：サブメニューの追加

「開く」メニューを選択したときに処理をおこなうときは、コマンド経由で ViewModel 側でおこないます。やり方は Button コントロールと同様に、Command プロパティに DelegateCommand クラスをデータバインディングさせます。

コード 5.6：ファイルを開くコマンドを持つ MainViewModel クラス

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      /// <summary>
4      /// MainView ウィンドウに対するデータコンテキストを表します。
5      /// </summary>
6      internal class MainViewModel : NotificationObject
7      {
8          private DelegateCommand _openFileCommand;
9          /// <summary>
10         /// ファイルを開くコマンドを取得します。
11         /// </summary>
12         public DelegateCommand OpenFileCommand
13         {
14             get
15             {
16                 return this._openFileCommand ?? (this._openFileCommand = new DelegateCommand(
17                 _ =>
18                 {
19                     System.Diagnostics.Debug.WriteLine("ファイルを開きます。");
```

```
20                  }));
21              }
22          }
23      }
24  }
```

コード 5.7：ファイルを開くコマンドをデータバインディングで紐付ける

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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          <Menu DockPanel.Dock="Top">
7              <MenuItem Header="ファイル (_F)">
8                  <MenuItem Header="開く(_O)" Command="{Binding OpenFileCommand}" />
9              </MenuItem>
10             <MenuItem Header="ヘルプ (_H)" />
11         </Menu>
12
13         <StatusBar DockPanel.Dock="Bottom">
14         </StatusBar>
15
16         <Grid Background="MediumSeaGreen">
17         </Grid>
18     </DockPanel>
19 </Window>
```

次のように出力ウィンドウにメッセージが表示されれば成功です。

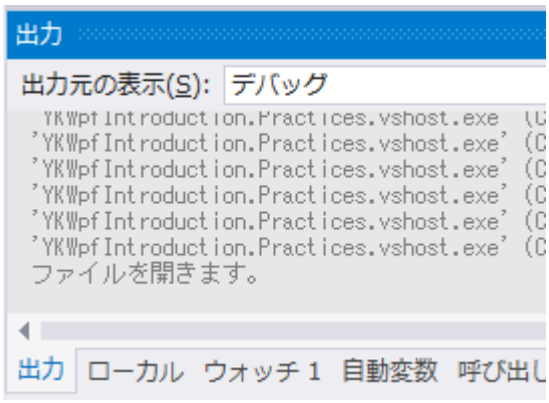


図 5.8：メニューを選択するとコマンドが実行されている

本節の冒頭でも述べましたが、ダイアログを開くという仕事は View 側の役割なので、MainViewModel 側から直接 OpenFileDialog クラスの ShowDialog() メソッドを呼ぶようなことはしません。ここでは添付ビヘイビアによって View 側にコールバックメソッドを渡し、View 側でダイアログを開いた後、コールバックメソッドによって ViewModel 側で開いたファイルに関する処理をおこなうようにします。

添付ビヘイビアを説明する前に、添付プロパティについて次節で説明します。

5.3.2 添付プロパティの作成

これまでに Canvas.Left や Grid.Row などの添付プロパティが登場してきました。これらは、自分自身に与えるプロパティではなく、異なるコントロールに対して添付するためのプロパティであり、Canvas パネルや Grid パネルはこれらの添付プロパティを読み込んで子要素の配置をおこなっていました。

つまり、添付プロパティとは既存のコントロールにプロパティを添付することです。このような添付プロパティを自作することができます。ここでは MenuItem コントロールに ViewModel のコールバックメソッドをデータバインディングできるように Action<bool, string> 型の添付プロパティを作成してみましょう。

まず、CommonDialogBehavior クラスを作成します。View に関する操作なので、ソリューションエクスプローラーの Views フォルダに Behaviors フォルダを作成し、そこにファイルを追加します。このクラスを使って添付プロパティを説明しますが、その後これを利用して添付ビヘイビアとするため、フォルダ名を Behaviors としています。

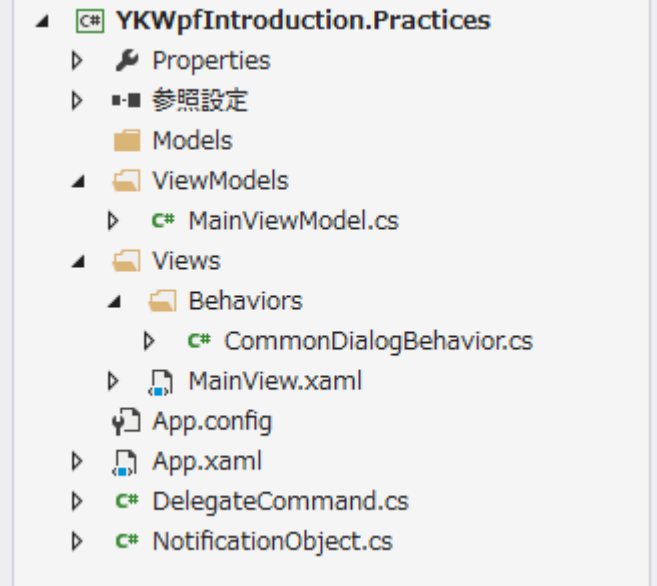


図 5.9：ビヘイビアを記述するためのクラスを追加する

添付プロパティを定義するには、DependencyProperty クラスを用いて次のように記述します。

コード 5.8：Callback 添付プロパティを持つ CommonDialogBehavior クラス

CommonDialogBehavior.cs

```
1  namespace YKWpfIntroduction.Practices.Views.Behaviors
2  {
3      using System;
4      using System.Windows;
5
6      /// <summary>
7      /// コモンダイアログに関するビヘイビアを表します。
8      /// </summary>
9      internal class CommonDialogBehavior
10     {
11         #region Callback 添付プロパティ
12         /// <summary>
13         /// Action&lt;bool, string&gt; 型の Callback 添付プロパティを定義します。
14         /// </summary>
15         public static readonly DependencyProperty CallbackProperty =
   DependencyProperty.RegisterAttached("Callback", typeof(Action<bool, string>),
   typeof(CommonDialogBehavior), new PropertyMetadata(null));
16
17         /// <summary>
18         /// Callback 添付プロパティを取得します。
19         /// </summary>
20         /// <param name="target">対象とする DependencyObject を指定します。</param>
```

```
21          /// <returns>取得した値を返します。</returns>
22          public static Action<bool, string> GetCallback(DependencyObject target)
23          {
24              return (Action<bool, string>)target.GetValue(CallbackProperty);
25          }
26
27          /// <summary>
28          /// Callback 添付プロパティを設定します。
29          /// </summary>
30          /// <param name="target">対象とする DependencyObject を指定します。</param>
31          /// <param name="value">設定する値を指定します。</param>
32          public static void SetCallback(DependencyObject target, Action<bool, string> value)
33          {
34              target.SetValue(CallbackProperty, value);
35          }
36          #endregion Callback 添付プロパティ
37      }
38  }
```

まず 15 行目で CallbackProperty という名前の変数を定義しています。この変数が添付プロパティを表しており、DependencyProperty.RegisterAttached() メソッドでどのような添付プロパティかを定義しています。

DependencyProperty.RegisterAttached() メソッドの第 1 引数は添付プロパティの名前を指定します。第 2 引数は添付プロパティの型、第 3 引数はこの添付プロパティを所有するクラスの型を指定します。第 4 引数には添付プロパティに対するメタ情報を指定します。ここでは ProeprtyMetadata クラスを使用して、既定値が null であるというメタ情報を指定しています。

さらに、添付プロパティを定義する場合、Get○○() メソッドと Set○○() メソッドを必ず定義する必要があります。○○の部分は必ず添付プロパティ名と同一でなければなりません。ここでは添付プロパティ名を "Callback" としているため、GetCallback() メソッドと SetCallback() メソッドになっています。

一方で、Action<bool, string> 型のプロパティを MainViewModel に準備します。これは後ほどダイアログのコールバックメソッドとして利用するため、DialogCallback という名前で次のように定義します。

コード 5.9：MainViewModel クラスに DialogCallback プロパティを追加する

MainViewModel.cs

```
1   namespace YKWpfIntroduction.Practices.ViewModels
2   {
3       using System;
4
5       /// <summary>
6       /// MainView ウィンドウに対するデータコンテキストを表します。
7       /// </summary>
8       internal class MainViewModel : NotificationObject
9       {
10          private DelegateCommand _openFileCommand;
11          /// <summary>
12          /// ファイルを開くコマンドを取得します。
13          /// </summary>
14          public DelegateCommand OpenFileCommand
15          {
16              get
17              {
18                  return this._openFileCommand ?? (this._openFileCommand = new DelegateCommand(
19                  _ =>
20                  {
21                      System.Diagnostics.Debug.WriteLine("ファイルを開きます。");
22                  }));
23              }
24          }
25
26          private Action<bool, string> _dialogCallback;
```

```
27          /// <summary>
28          /// ダイアログに対するコールバックを取得します。
29          /// </summary>
30          public Action<bool, string> DialogCallback
31          {
32              get { return this._dialogCallback; }
33              private set { SetProperty(ref this._dialogCallback, value); }
34          }
35      }
36  }
```

準備が整ったので、「開く」メニューを表す MenuItem コントロールに用意した添付プロパティを実装しましょう。

コード 5.10：作成した添付プロパティを使用する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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:YKWpfIntroduction.Practices.Views.Behaviors"
5          Title="MainView" Height="300" Width="300">
6      <DockPanel>
7          <Menu DockPanel.Dock="Top">
8              <MenuItem Header="ファイル (_F)">
9                  <MenuItem Header="開く(_O)"
10                           Command="{Binding OpenFileCommand}"
11                           b:CommonDialogBehavior.Callback="{Binding DialogCallback}"
12                           />
13             </MenuItem>
14             <MenuItem Header="ヘルプ (_H)" />
15         </Menu>
16
17         <StatusBar DockPanel.Dock="Bottom">
18         </StatusBar>
19
20         <Grid Background="MediumSeaGreen">
21         </Grid>
22     </DockPanel>
23 </Window>
```

11 行目で CommonDialogBehavior.Callback 添付プロパティに対して MainViewModel クラスの DialogCallback プロパティをデータバインディングしています。ただし、CommonDialogBehavior クラスは .NET Framework 標準のクラスではないため、アクセスするために名前空間のエイリアスを定義する必要があります。これが 4 行目の "xmlns:b" から始まる記述で、YKWpfIntroduction.Practices.Views.Behaviors 名前空間のエイリアスを "b" として定義しています。こうすることで "b:" で記述すると YKWpfIntroduction.Practices.Views.Behaviors 名前空間に属するクラスにアクセスできるようになります。11 行目でも "b:CommonDialogBehavior" というように記述しています。

この時点でアプリケーションを実行しても特に動作は変わりません。というのは MainViewModel クラスの DialogCallback プロパティは何も指定していないので中身は null ですし、そもそも作成した添付プロパティを参照する相手がいないからです。

次にこの添付プロパティを利用してコントロールの振る舞いを決定する添付ビヘイビアを作成します。

5.3.3　添付ビヘイビアの作成

添付ビヘイビアとは、添付プロパティを利用してそのコントロールの振る舞いを決定するためのものです。ここでは、Callback 添付プロパティによって決められた振る舞いをするように添付ビヘイビアを作成します。

前節で作成した CommonDialogBehavior クラスに Callback 添付プロパティの値が変更されたときの処理を次のように追加します。

コード 5.11：Callback 添付プロパティの変更イベントハンドラで振る舞いを決定する

CommonDialogBehavior.cs

```
 1  namespace YKWpfIntroduction.Practices.Views.Behaviors
 2  {
 3      using System;
 4      using System.Windows;
 5
 6      /// <summary>
 7      /// コモンダイアログに関するビヘイビアを表します。
 8      /// </summary>
 9      internal class CommonDialogBehavior
10      {
11          #region Callback 添付プロパティ
12          /// <summary>
13          /// Action&lt;bool, string&gt; 型の Callback 添付プロパティを定義します。
14          /// </summary>
15          public static readonly DependencyProperty CallbackProperty =
    DependencyProperty.RegisterAttached("Callback", typeof(Action<bool, string>),
    typeof(CommonDialogBehavior), new PropertyMetadata(null, OnCallbackPropertyChanged));
16
17          /// <summary>
18          /// Callback 添付プロパティを取得します。
19          /// </summary>
20          /// <param name="target">対象とする DependencyObject を指定します。</param>
21          /// <returns>取得した値を返します。</returns>
22          public static Action<bool, string> GetCallback(DependencyObject target)
23          {
24              return (Action<bool, string>)target.GetValue(CallbackProperty);
25          }
26
27          /// <summary>
28          /// Callback 添付プロパティを設定します。
29          /// </summary>
30          /// <param name="target">対象とする DependencyObject を指定します。</param>
31          /// <param name="value">設定する値を指定します。</param>
32          public static void SetCallback(DependencyObject target, Action<bool, string> value)
33          {
34              target.SetValue(CallbackProperty, value);
35          }
36          #endregion Callback 添付プロパティ
37
38          /// <summary>
39          /// Callback 添付プロパティ変更イベントハンドラ
40          /// </summary>
41          /// <param name="sender">イベント発行元</param>
42          /// <param name="e">イベント引数</param>
43          private static void OnCallbackPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
44          {
45              var callback = GetCallback(sender);
46              if (callback != null)
47              {
48                  callback(true, "ファイルパス");
49              }
```

```
50          }
51      }
52  }
```

15 行目で定義した Callback 添付プロパティに対するメタ情報に OnCallbackPropertyChanged イベントハンドラを追加しています。これは、Callback 添付プロパティの値が変更されたときに呼ばれるメソッドを指定しています。このメソッドは 43 行目で定義しています。このメソッドの中で、イベント発行元から GetCallback() メソッドによって Callback 添付プロパティを取得しています。イベント発行元とは、Callback 添付プロパティが変更されたコントロールとなります。そして、取得した Callback 添付プロパティが null でなければ実行するようにしています。

ワンポイント 5.1：イベントハンドラも自動生成機能で作成できる

ワンポイント 4.1 でも紹介したように、Visual Studio には未定義のメソッドやプロパティを自動生成する機能があります。イベントハンドラも例外なく自動生成できるので、例えば添付プロパティの変更イベントハンドラもこの機能で生成しましょう。コード 5.11 の 43 行目に Callback 添付プロパティの変更イベントハンドラを定義していますが、このメソッドを自分で入力する前に、15 行目の PropertyMetadata クラスのコンストラクタの入力引数に対して "OnCallbackPropertyChanged" と入力しましょう。すると、ワンポイント 4.1 のようにメニューが表示され、これを選択すると 43 行目のようなメソッドが自動的に生成されます。

自動生成機能を使うことで、そのイベントハンドラがどのような入力引数だったのかを覚えていなくても簡単にイベントハンドラを作ることができるので、積極的に利用しましょう。

続いて、MainViewModel クラスで DialogCallback プロパティが変更されるように修正します。

コード 5.12：DialogCallback プロパティが変更されるように修正

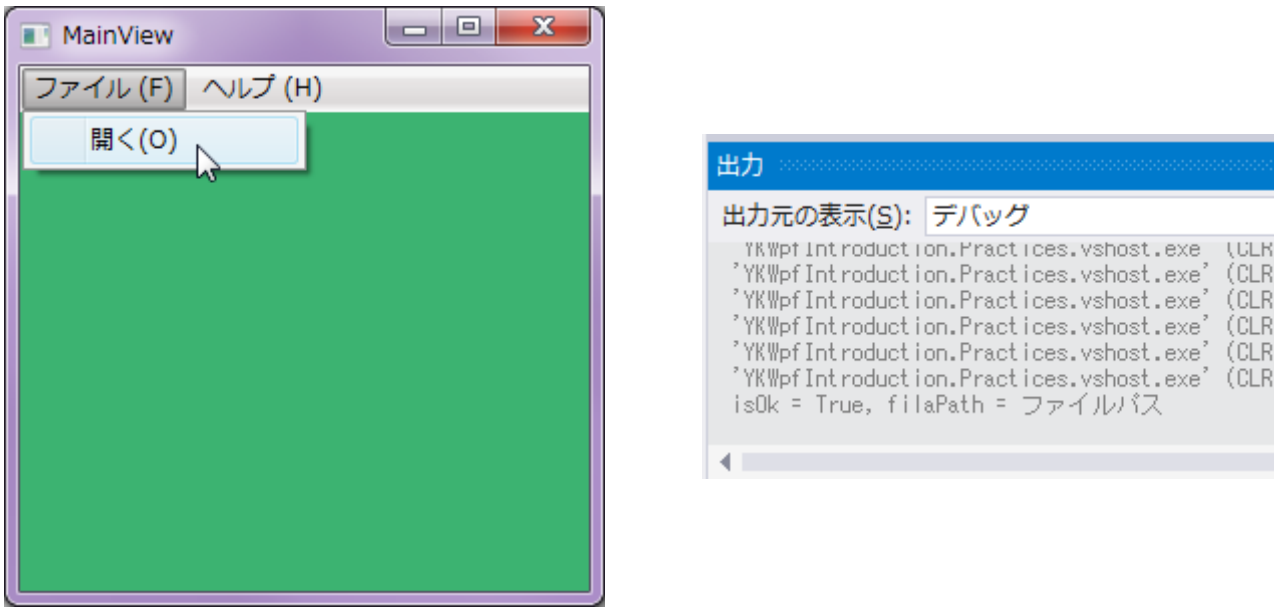
MainViewModel.cs

```
 1  namespace YKWpfIntroduction.Practices.ViewModels
 2  {
 3      using System;
 4
 5      /// <summary>
 6      /// MainView ウィンドウに対するデータコンテキストを表します。
 7      /// </summary>
 8      internal class MainViewModel : NotificationObject
 9      {
10          private DelegateCommand _openFileCommand;
11          /// <summary>
12          /// ファイルを開くコマンドを取得します。
13          /// </summary>
14          public DelegateCommand OpenFileCommand
15          {
16              get
17              {
18                  return this._openFileCommand ?? (this._openFileCommand = new DelegateCommand(
19                  _ =>
20                  {
21                      this.DialogCallback = OnDialogCallback;
22                  }));
23              }
24          }
25
26          private Action<bool, string> _dialogCallback;
27          /// <summary>
28          /// ダイアログに対するコールバックを取得します。
29          /// </summary>
30          public Action<bool, string> DialogCallback
31          {
32              get { return this._dialogCallback; }
33              private set { SetProperty(ref this._dialogCallback, value); }
34          }
35
36          /// <summary>
```

```
37          /// ダイアログに対するコールバック処理をおこないます。
38          /// </summary>
39          /// <param name="isOk">ダイアログの結果を指定します。</param>
40          /// <param name="filePath">ファイルのフルパスを指定します。</param>
41          private void OnDialogCallback(bool isOk, string filePath)
42          {
43              this.DialogCallback = null;
44              System.Diagnostics.Debug.WriteLine("コールバック処理をおこないます。");
45          }
46      }
47  }
```

21 行目で DialogCallback プロパティを変更しています。ここの処理は「開く」メニューを選択したときに実行されます。また、指定した OnDialogCallback() メソッドでは DialogCallback プロパティに null を指定しています。コールバック処理されたときに DialogCallback プロパティを null に戻すようにしています。

それではアプリケーションを実行してみましょう。図 5.10 のように、「開く」メニューを選択すると出力ウィンドウにメッセージが表示されるようになりました。



(a) 「開く」メニューを選択する　　(b) メッセージが表示される

図 5.10：添付ビヘイビアによるコールバック処理の確認

動作について箇条書きで整理します。

1. 「開く」メニューが選択されると MainViewModel クラスの OpenFileCommand プロパティが実行される
2. MainViewModel クラスの OpenFileCommand プロパティが実行され、DialogCallback プロパティが変更される
3. MainViewModel クラスの DialogCallback プロパティと CommonDialogBehavior クラスの Callback 添付プロパティがデータバインディングによって紐付けられているため、CommonDialogBehavior クラスの Callback 添付プロパティが変更される
4. CommonDialogBehavior クラスの OnCallbackPropertyChanged イベントハンドラが処理される
5. OnCallbackPropertyChanged イベントハンドラで指定されたコールバックメソッドが呼び出される
6. MainViewModel クラスの OnDialogCallback() メソッドが処理され、DialogCallback プロパティが null に変更されるので、CommonDialogBehavior クラスの OnCallbackPropertyChanged イベントハンドラが処理されるが、Callback 添付プロパティは null に変更されているため、特に何も処理されずに OnCallbackPropertyChanged が終了する
7. 引き続き MainViewModel クラスの OnDialogCallback() メソッドの処理が継続して終了する

5.3.4 ファイルを開くダイアログを表示する

冒頭でも説明したように、ファイルを開くためのコモンダイアログは Microsoft.Win32.OpenFileDialog クラスの ShowDialog() メソッドを呼び出す必要があります。前節で作成した添付ビヘイビアでこの処理をおこないます。

コード 5.13：Callback 添付プロパティの変更イベントハンドラでダイアログを開く

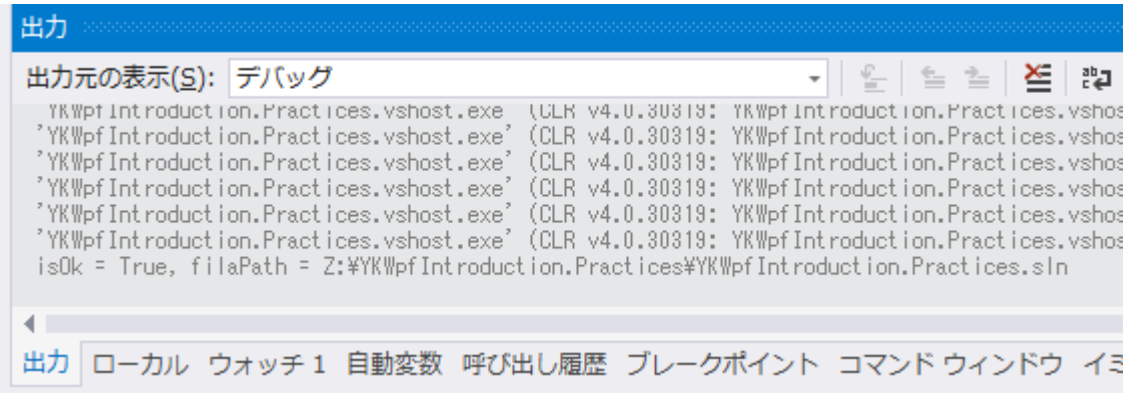
CommonDialogBehavior.cs

```
 1  namespace YKWpfIntroduction.Practices.Views.Behaviors
 2  {
 3      using Microsoft.Win32;
 4      using System;
 5      using System.Windows;
 6
 7      /// <summary>
 8      /// コモンダイアログに関するビヘイビアを表します。
 9      /// </summary>
10      internal class CommonDialogBehavior
11      {
12          #region Callback 添付プロパティ
13          /// <summary>
14          /// Action&lt;bool, string&gt; 型の Callback 添付プロパティを定義します。
15          /// </summary>
16          public static readonly DependencyProperty CallbackProperty =
    DependencyProperty.RegisterAttached("Callback", typeof(Action<bool, string>),
    typeof(CommonDialogBehavior), new PropertyMetadata(null, OnCallbackPropertyChanged));
17
18          /// <summary>
19          /// Callback 添付プロパティを取得します。
20          /// </summary>
21          /// <param name="target">対象とする DependencyObject を指定します。</param>
22          /// <returns>取得した値を返します。</returns>
23          public static Action<bool, string> GetCallback(DependencyObject target)
24          {
25              return (Action<bool, string>)target.GetValue(CallbackProperty);
26          }
27
28          /// <summary>
29          /// Callback 添付プロパティを設定します。
30          /// </summary>
31          /// <param name="target">対象とする DependencyObject を指定します。</param>
32          /// <param name="value">設定する値を指定します。</param>
33          public static void SetCallback(DependencyObject target, Action<bool, string> value)
34          {
35              target.SetValue(CallbackProperty, value);
36          }
37          #endregion Callback 添付プロパティ
38
39          /// <summary>
40          /// Callback 添付プロパティ変更イベントハンドラ
41          /// </summary>
42          /// <param name="sender">イベント発行元</param>
43          /// <param name="e">イベント引数</param>
44          private static void OnCallbackPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
45          {
46              var callback = GetCallback(sender);
47              if (callback != null)
48              {
49                  var dlg = new OpenFileDialog()
50                  {
51                      Title = "ファイルを開きましょう",
```

```
52                     Filter = "画像ファイル (*.bmp; *.jpg; *.png)|*.bmp;*.jpg;*.png|すべてのファイル (*.*)|*.*",
53                     Multiselect = false,
54                 };
55                 var owner = Window.GetWindow(sender);
56                 var result = dlg.ShowDialog(owner);
57                 callback(result.Value, dlg.FileName);
58             }
59         }
60     }
61 }
```

3 行目で Microsoft.Win32 名前空間を使用することを宣言し、後は Callback 添付プロパティの変更イベントハンドラ内だけを変更しています。ここで OpenFileDialog() をインスタンス化し、必要なプロパティを設定した後に ShowDialog() メソッドを呼び出しています。その結果と指定されたファイルのフルパスをコールバック処理の入力引数に与えています。ShowDialog() メソッドは入力引数なしでも呼び出すことができますが、上記のようにオーナーウィンドウを指定することもできます。オーナーウィンドウを指定すると、そのウィンドウの傍にダイアログが表示されるようになります。

アプリケーションを実行し、「開く」メニューを選択してみましょう。ファイルを開くためのダイアログが開き、適当なファイルを開くと Visual Studio の出力ウィンドウに選択したファイルのフルパスが表示されるようになりました。



出力元の表示(S): デバッグ

```
 YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshos
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshos
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshos
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshos
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshos
'YKWpfIntroduction.Practices.vshost.exe' (CLR v4.0.30319: YKWpfIntroduction.Practices.vshos
isOk = True, filaPath = Z:¥YKWpfIntroduction.Practices¥YKWpfIntroduction.Practices.sln
```

図 5.11：選択したファイルのフルパスが ViewModel 側に伝わっている

5.3.5 CommonDialogBehavior クラスを汎用的にする

前節のサンプルでは、CommonDialogBehavior クラス内部でコモンダイアログに対するタイトルやファイルフィルタなどの設定を固定値として与えていましたが、これも添付プロパティとして定義することで、XAML から指定することができるようになります。

Title、Filter、Multiselect プロパティを添付プロパティとして定義するようにした CommonDialogBehavior クラスは次のようになります。

コード 5.14：ダイアログのタイトルなども外部から設定できるようにした CommonDialogBehavior クラス

CommonDialogBehavior.cs

```
namespace YKWpfIntroduction.Practices.Views.Behaviors
{
    using Microsoft.Win32;
    using System;
    using System.Windows;

    /// <summary>
    /// コモンダイアログに関するビヘイビアを表します。
    /// </summary>
    internal class CommonDialogBehavior
    {
        #region Callback 添付プロパティ
        /// <summary>
        /// Action&lt;bool, string&gt; 型の Callback 添付プロパティを定義します。
        /// </summary>
        public static readonly DependencyProperty CallbackProperty =
DependencyProperty.RegisterAttached("Callback", typeof(Action<bool, string>),
typeof(CommonDialogBehavior), new PropertyMetadata(null, OnCallbackPropertyChanged));

        /// <summary>
        /// Callback 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 Action<bool, string> GetCallback(DependencyObject target)
        {
            return (Action<bool, string>)target.GetValue(CallbackProperty);
        }

        /// <summary>
        /// Callback 添付プロパティを設定します。
        /// </summary>
        /// <param name="target">対象とする DependencyObject を指定します。</param>
        /// <param name="value">設定する値を指定します。</param>
        public static void SetCallback(DependencyObject target, Action<bool, string> value)
        {
            target.SetValue(CallbackProperty, value);
        }
        #endregion Callback 添付プロパティ

        #region Title 添付プロパティ
        /// <summary>
        /// string 型の Title 添付プロパティを定義します。
        /// </summary>
        public static readonly DependencyProperty TitleProperty =
DependencyProperty.RegisterAttached("Title", typeof(string), typeof(CommonDialogBehavior), new
PropertyMetadata("ファイルを開く"));

        /// <summary>
        /// Title 添付プロパティを取得します。
        /// </summary>
```

```
48          /// <param name="target">対象とする DependencyObject を指定します。</param>
49          /// <returns>取得した値を返します。</returns>
50          public static string GetTitle(DependencyObject target)
51          {
52              return (string)target.GetValue(TitleProperty);
53          }
54
55          /// <summary>
56          /// Title 添付プロパティを設定します。
57          /// </summary>
58          /// <param name="target">対象とする DependencyObject を指定します。</param>
59          /// <param name="value">設定する値を指定します。</param>
60          public static void SetTitle(DependencyObject target, string value)
61          {
62              target.SetValue(TitleProperty, value);
63          }
64          #endregion Title 添付プロパティ
65
66          #region Filter 添付プロパティ
67          /// <summary>
68          /// string 型の Filter 添付プロパティを定義します。
69          /// </summary>
70          public static readonly DependencyProperty FilterProperty =
    DependencyProperty.RegisterAttached("Filter", typeof(string), typeof(CommonDialogBehavior), new
    PropertyMetadata("すべてのファイル (*.*)|*.*"));
71
72          /// <summary>
73          /// Filter 添付プロパティを取得します。
74          /// </summary>
75          /// <param name="target">対象とする DependencyObject を指定します。</param>
76          /// <returns>取得した値を返します。</returns>
77          public static string GetFilter(DependencyObject target)
78          {
79              return (string)target.GetValue(FilterProperty);
80          }
81
82          /// <summary>
83          /// Filter 添付プロパティを設定します。
84          /// </summary>
85          /// <param name="target">対象とする DependencyObject を指定します。</param>
86          /// <param name="value">設定する値を指定します。</param>
87          public static void SetFilter(DependencyObject target, string value)
88          {
89              target.SetValue(FilterProperty, value);
90          }
91          #endregion Filter 添付プロパティ
92
93          #region Multiselect 添付プロパティ
94          /// <summary>
95          /// bool 型の Multiselect 添付プロパティを定義します。
96          /// </summary>
97          public static readonly DependencyProperty MultiselectProperty =
    DependencyProperty.RegisterAttached("Multiselect", typeof(bool), typeof(CommonDialogBehavior), new
    PropertyMetadata(true));
98
99          /// <summary>
100         /// Multiselect 添付プロパティを取得します。
101         /// </summary>
102         /// <param name="target">対象とする DependencyObject を指定します。</param>
103         /// <returns>取得した値を返します。</returns>
```

```
104         public static bool GetMultiselect(DependencyObject target)
105         {
106             return (bool)target.GetValue(MultiselectProperty);
107         }
108 
109         /// <summary>
110         /// Multiselect 添付プロパティを設定します。
111         /// </summary>
112         /// <param name="target">対象とする DependencyObject を指定します。</param>
113         /// <param name="value">設定する値を指定します。</param>
114         public static void SetMultiselect(DependencyObject target, bool value)
115         {
116             target.SetValue(MultiselectProperty, value);
117         }
118         #endregion Multiselect 添付プロパティ
119 
120         /// <summary>
121         /// Callback 添付プロパティ変更イベントハンドラ
122         /// </summary>
123         /// <param name="sender">イベント発行元</param>
124         /// <param name="e">イベント引数</param>
125         private static void OnCallbackPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
126         {
127             var callback = GetCallback(sender);
128             if (callback != null)
129             {
130                 var dlg = new OpenFileDialog()
131                 {
132                     Title = GetTitle(sender),
133                     Filter = GetFilter(sender),
134                     Multiselect = GetMultiselect(sender),
135                 };
136                 var owner = Window.GetWindow(sender);
137                 var result = dlg.ShowDialog(owner);
138                 callback(result.Value, dlg.FileName);
139             }
140         }
141     }
142 }
```

この添付ビヘイビアは XAML では次のように使用できます。

コード 5.15：作成した添付プロパティを使用する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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:YKWpfIntroduction.Practices.Views.Behaviors"
5          Title="MainView" Height="300" Width="300">
6      <DockPanel>
7          <Menu DockPanel.Dock="Top">
8              <MenuItem Header="ファイル (_F)">
9                  <MenuItem Header="開く(_O)"
10                           Command="{Binding OpenFileCommand}"
11                           b:CommonDialogBehavior.Title="ファイルを開く"
12                           b:CommonDialogBehavior.Filter="画像ファイル (*.bmp; *.jpg;
    *.png)|*.bmp;*.jpg;*.png|すべてのファイル (*.*)|*.*"
13                           b:CommonDialogBehavior.Multiselect="False"
14                           b:CommonDialogBehavior.Callback="{Binding DialogCallback}"
```

```
15                          />
16              </MenuItem>
17              <MenuItem Header="ヘルプ (_H)" />
18          </Menu>
19
20          <StatusBar DockPanel.Dock="Bottom">
21          </StatusBar>
22
23          <Grid Background="MediumSeaGreen">
24          </Grid>
25      </DockPanel>
26  </Window>
```

添付ビヘイビアは様々なコントロールに使い回せるというメリットがあるため、できるだけ汎用的に作成したほうが便利です。ここではファイルを開く機能しかありませんが、ファイルを保存するためのコモンダイアログもほとんど同様のプロパティで設定できるため、ファイルを開くモードか保存するモードかを選択させるような添付プロパティを追加し、モードによって OpenFileDialog クラスを使用するか SaveFileDialog クラスを使用するかを分岐させるようにするとなお便利な添付ビヘイビアになるでしょう。

5.4 「終了」メニューでアプリケーションを終了させる

アプリケーションをいきなり終了されてしまうとこれまでの作業内容が保存されなかったり、ログを保存できなかったりいろいろ不便が生じます。アプリケーションを終了するときは内部でなんらかの処理をおこない、その上で終了するようにしたほうが良いでしょう。ここでは、ユーザーからアプリケーションを終了したいという要求を受け取ったときにどのように処理するかについて説明します。

5.4.1 アプリケーションの終了方法はメニューだけではない

まず始めに、アプリケーションを終了する方法はひとつだけではないことに注意しなければいけません。これから作る「終了」メニューで終了させることはもちろんですが、ウィンドウに元々付いている「x」ボタンを押すことでも終了できます。また、Alt+Space キーや、ウィンドウ左上隅のアイコンをクリックすることで表示されるシステムメニューの「閉じる」メニューもしくは Alt+F4 キーによって終了させることもできます。さらに、タスクバーに表示されているウィンドウを右クリックすると「ウィンドウを閉じる」というメニューが選択できます。アプリケーション終了時に処理をおこなう場合は、これらすべてのパターンを網羅しなければ意味がないことに注意しましょう。

5.4.2 「終了」メニューでアプリケーションを終了させる

まずはメニューから終了する方法です。「終了」メニューが選択されたときに何か処理をおこなうということで、MenuItem コントロールの Command プロパティに MainViewModel クラスのコマンドプロパティを紐付けましょう。ここでは ExitCommand プロパティを定義し、これを紐付けます。

コード 5.16：アプリケーションを終了するための ExitCommand プロパティを追加

MainViewModel.cs

```
 1  namespace YKWpfIntroduction.Practices.ViewModels
 2  {
 3      using System;
 4
 5      /// <summary>
 6      /// MainView ウィンドウに対するデータコンテキストを表します。
 7      /// </summary>
 8      internal class MainViewModel : NotificationObject
 9      {
10          ファイルを開く
48
49          #region アプリケーションを終了する
50          private DelegateCommand _exitCommand;
51          /// <summary>
52          /// アプリケーション終了コマンドを取得します。
53          /// </summary>
54          public DelegateCommand ExitCommand
55          {
56              get
57              {
58                  return this._exitCommand ?? (this._exitCommand = new DelegateCommand(
59                  _ =>
60                  {
61                      OnExit();
62                  }));
63              }
64          }
65
66          /// <summary>
67          /// アプリケーションを終了します。
68          /// </summary>
69          private void OnExit()
70          {
71              App.Current.Shutdown();
72          }
73          #endregion アプリケーションを終了する
74      }
```

```
75  }
```

コード 5.17：「終了」メニューに ExitCommand プロパティを紐付ける

MainView.xaml

```
1   <Window x:Class="YKWpfIntroduction.Practices.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:YKWpfIntroduction.Practices.Views.Behaviors"
5           Title="MainView" Height="300" Width="300">
6       <DockPanel>
7           <Menu DockPanel.Dock="Top">
8               <MenuItem Header="ファイル (_F)">
9                   <MenuItem Header="開く(_O)"...
16                  <Separator />
17                  <MenuItem Header="終了 (_X)" Command="{Binding ExitCommand}" />
18              </MenuItem>
19              <MenuItem Header="ヘルプ (_H)" />
20          </Menu>
21
22          <StatusBar DockPanel.Dock="Bottom">
23          </StatusBar>
24
25          <Grid Background="MediumSeaGreen">
26          </Grid>
27      </DockPanel>
28  </Window>
```

C# コード上からアプリケーションを終了させるときは Application クラスの Shutdown() メソッドを使います。コード 5.16 のように、App.Current プロパティで現在の Application クラスを取得し、これに対する Shutdown() メソッドを呼び出します。

もし終了前に何か処理をしたい場合はコード 5.16 の 69 行目の OnExit() メソッドに処理を追加すればいいし、そもそも終了できないようにする場合は、ExitCommand プロパティの実行可能判別処理を追加して false を返すようにすると、「終了」メニューが無効化して選択できなくなります。

5.4.3 「x」ボタンでアプリケーションを終了される場合

「x」ボタンでアプリケーションを終了されてしまう場合、前節のように直接コマンドで処理することができません。そこで、添付ビヘイビアで Window.Closing イベントを利用し、このイベントハンドラで ViewModel 側のコールバック処理をおこなうようにします。添付ビヘイビアについては「5.3.3 添付ビヘイビアの作成」で説明しているため、詳細は省略します。

添付ビヘイビアとして WindowClosingBehavior クラスを次のように定義します。

コード 5.18：Window.Closing イベントを利用した WindowClosingBehavior 添付ビヘイビア

WindowClosingBehavior.cs

```
namespace YKWpfIntroduction.Practices.Views.Behaviors
{
    using System;
    using System.ComponentModel;
    using System.Windows;

    /// <summary>
    /// Window を閉じるときのビヘイビアを表します。
    /// </summary>
    internal class WindowClosingBehavior
    {
        #region Callback 添付プロパティ
        /// <summary>
        /// Func&lt;bool&gt; 型の Callback 添付プロパティを定義します。
        /// </summary>
        public static readonly DependencyProperty CallbackProperty =
DependencyProperty.RegisterAttached("Callback", typeof(Func<bool>), typeof(WindowClosingBehavior),
new PropertyMetadata(null, OnIsEnabledPropertyChagned));

        /// <summary>
        /// Callback 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 Func<bool> GetCallback(DependencyObject target)
        {
            return (Func<bool>)target.GetValue(CallbackProperty);
        }

        /// <summary>
        /// Callback 添付プロパティを設定します。
        /// </summary>
        /// <param name="target">対象とする DependencyObject を指定します。</param>
        /// <param name="value">設定する値を指定します。</param>
        public static void SetCallback(DependencyObject target, Func<bool> value)
        {
            target.SetValue(CallbackProperty, value);
        }
        #endregion Callback 添付プロパティ

        /// <summary>
        /// Callback 添付プロパティ変更イベントハンドラ
        /// </summary>
        /// <param name="sender">イベント発行元</param>
        /// <param name="e">イベント引数</param>
        private static void OnIsEnabledPropertyChagned(DependencyObject sender,
DependencyPropertyChangedEventArgs e)
        {
            var w = sender as Window;
            if (w != null)
            {
```

```
49                  var callback = GetCallback(w);
50                  if ((callback != null) && (e.OldValue == null))
51                  {
52                      w.Closing += OnClosing;
53                  }
54                  else if (callback == null)
55                  {
56                      w.Closing -= OnClosing;
57                  }
58              }
59          }
60
61          /// <summary>
62          /// Closing イベントハンドラ
63          /// </summary>
64          /// <param name="sender">イベント発行元</param>
65          /// <param name="e">イベント引数</param>
66          private static void OnClosing(object sender, CancelEventArgs e)
67          {
68              var callback = GetCallback(sender as DependencyObject);
69              if (callback != null)
70              {
71                  // コールバック処理の結果が false のときキャンセルする
72                  e.Cancel = !callback();
73              }
74          }
75      }
76  }
```

ViewModel 側から Func<bool> 型のプロパティとしてコールバックメソッドをもらうことを想定し、このメソッドが null でないとき、Window.Closing イベントを捕捉し、そのコールバック処理の結果に応じてアプリケーションを終了したりしなかったりするようにしています。

例えば ViewModel 側を次のように変更します。

コード 5.19：アプリケーション終了に条件を付ける

WindowClosingBehavior.cs

```
1   namespace YKWpfIntroduction.Practices.ViewModels
2   {
3       using System;
4
5       /// <summary>
6       /// MainView ウィンドウに対するデータコンテキストを表します。
7       /// </summary>
8       internal class MainViewModel : NotificationObject
9       {
10          ファイルを開く
48
49          #region アプリケーションを終了する
50          public Func<bool> ClosingCallback
51          {
52              get { return OnExit; }
53          }
54
55          private DelegateCommand _exitCommand;
56          /// <summary>
57          /// アプリケーション終了コマンドを取得します。
58          /// </summary>
59          public DelegateCommand ExitCommand
60          {
```

```
61              get
62              {
63                  return this._exitCommand ?? (this._exitCommand = new DelegateCommand(
64                  _ =>
65                  {
66                      OnExit();
67                  }));
68              }
69          }
70
71          /// <summary>
72          /// アプリケーションを終了します。
73          /// </summary>
74          private bool OnExit()
75          {
76              if (this._counter < 3)
77              {
78                  this._counter++;
79                  return false;
80              }
81
82              App.Current.Shutdown();
83              return true;
84          }
85
86          private int _counter;
87          #endregion アプリケーションを終了する
88      }
89  }
```

86 行目で定義した変数 _counter で終了コマンドが投げられた回数を数え、これが 4 回以上になったときに初めて 82 行目の Shutdown() メソッドが呼ばれるようにしています。3 回以下のとき、OnExit() メソッドが false を返しているため、ClosingCallback プロパティから呼ばれた場合、その戻り値が false となります。これは、WindowClosingBehavior クラスからコールバック処理として呼び出されることを想定しており、コード 5.18 の 69 行目にあるように、戻り値が false の場合は、反転して e.Calncel プロパティが true となります。Window.Closing イベントハンドラでは、このように e.Cancel プロパティを true にすることでウィンドウが閉じる処理を中断させることができます。

WindowClosingBehavior 添付ビヘイビアを実際に使ってみましょう。MainView ウィンドウの XAML は次のようになります。

コード 5.20：WindowClosingBehavior 添付ビヘイビアを Window に適用する

MainView.xaml

```
1   <Window x:Class="YKWpfIntroduction.Practices.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:YKWpfIntroduction.Practices.Views.Behaviors"
5           Title="MainView" Height="300" Width="300"
6           b:WindowClosingBehavior.Callback="{Binding ClosingCallback}">
7       <DockPanel>
8           <Menu DockPanel.Dock="Top">
9               <MenuItem Header="ファイル (_F)">
10                  <MenuItem Header="開く(_O)"...>/>
17                  <Separator />
18                  <MenuItem Header="終了 (_X)" Command="{Binding ExitCommand}" />
19              </MenuItem>
20              <MenuItem Header="ヘルプ (_H)" />
21          </Menu>
22
23          <StatusBar DockPanel.Dock="Bottom">
```

```
24          </StatusBar>
25
26          <Grid Background="MediumSeaGreen">
27          </Grid>
28      </DockPanel>
29  </Window>
```

アプリケーションを実行すると、「終了」メニューやウィンドウの「x」ボタンからアプリケーションを終了しようとしても、3 回まで終了せず、4 回目で終了するようになっています。

5.4.4 システムメニューからアプリケーションを終了される場合

実はシステムメニューからアプリケーションが終了される場合も Window.Closing イベントが発生します。したがって前節の WindowClosingBehavior 添付ビヘイビアで対応できます。

一方、そもそもシステムメニュー自体を表示させないようにしたり、Alt+F4 キーで終了できないようにしたりもできます。しかし、この方法は P/Invoke と呼ばれるプラットフォーム呼び出しという機能を使わなければいけないため、ここでは割愛します。

5.4.5 タスクバーの右クリックメニューからアプリケーションを終了される場合

こちらも Window.Closing イベントが発生するため、WindowClosingBehavior 添付ビヘイビアで対応できます。
また、Window.ShowInTaskbar プロパティを false にすることで、タスクバーに表示させないようにもできるため、そのような方法を取ることもできます。

5.5 「バージョン情報」メニューでバージョン情報を表示させる

5.5.1 子ウィンドウを表示させる

子ウィンドウを開くときは、その Window クラスの Show() メソッドあるいは ShowDialog() メソッドを呼び出す必要があります。したがって Window クラスのインスタンスを知っていなければいけません。つまり子ウィンドウを開く作業をおこなうのは View の役割となります。しかし、現在のアプリケーションの状態から子ウィンドウを開いていいかどうかを判断するのは ViewModel や Model しか知り得ません。したがって、コマンドによって ViewModel に指示を仰いでから、添付ビヘイビアを経由して View 側で子ウィンドウを生成することを考えます。

子ウィンドウを開くということは、別のウィンドウが必要となります。また、そのウィンドウに対するデータコンテキストも必要となります。ここではバージョン情報を表示するためのウィンドウなので、ウィンドウを VersionView クラス、そのデータコンテキストを VersionViewModel クラスという名前でソリューションエクスプローラーにそれぞれ追加します。

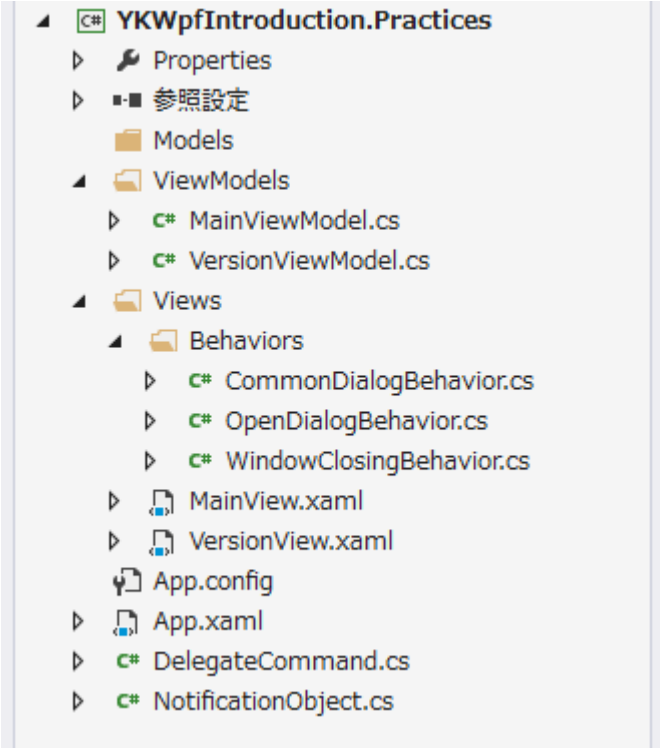


図 5.12：VersionView ウィンドウと VersionViewModel クラスをそれぞれ追加

ダイアログを開くための OpenDialogBehavior 添付ビヘイビアをコード 5.21 のように定義します。

コード 5.21：ダイアログを開くための OpenDialogBehavior 添付ビヘイビア

MainView.xaml

```
1  namespace YKWpfIntroduction.Practices.Views.Behaviors
2  {
3      using System;
4      using System.Windows;
5
6      /// <summary>
7      /// ダイアログを開くためのビヘイビアを表します。
8      /// </summary>
9      internal class OpenDialogBehavior
10     {
11         #region DataContext 添付プロパティ
12         /// <summary>
13         /// object 型の DataContext 添付プロパティを定義します。
14         /// </summary>
15         public static readonly DependencyProperty DataContextProperty =
   DependencyProperty.RegisterAttached("DataContext", typeof(object), typeof(OpenDialogBehavior), new
```

```
    PropertyMetadata(null));
16
17          /// <summary>
18          /// DataContext 添付プロパティを取得します。
19          /// </summary>
20          /// <param name="target">対象とする DependencyObject を指定します。</param>
21          /// <returns>取得した値を返します。</returns>
22          public static object GetDataContext(DependencyObject target)
23          {
24              return target.GetValue(DataContextProperty);
25          }
26
27          /// <summary>
28          /// DataContext 添付プロパティを設定します。
29          /// </summary>
30          /// <param name="target">対象とする DependencyObject を指定します。</param>
31          /// <param name="value">設定する値を指定します。</param>
32          public static void SetDataContext(DependencyObject target, object value)
33          {
34              target.SetValue(DataContextProperty, value);
35          }
36          #endregion DataContext 添付プロパティ
37
38          #region WindowType 添付プロパティ
39          /// <summary>
40          /// Type 型の WindowType 添付プロパティを定義します。
41          /// </summary>
42          public static readonly DependencyProperty WindowTypeProperty =
    DependencyProperty.RegisterAttached("WindowType", typeof(Type), typeof(OpenDialogBehavior), new
    PropertyMetadata(null));
43
44          /// <summary>
45          /// WindowType 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 Type GetWindowType(DependencyObject target)
50          {
51              return (Type)target.GetValue(WindowTypeProperty);
52          }
53
54          /// <summary>
55          /// WindowType 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 SetWindowType(DependencyObject target, Type value)
60          {
61              target.SetValue(WindowTypeProperty, value);
62          }
63          #endregion WindowType 添付プロパティ
64
65          #region Callback 添付プロパティ
66          /// <summary>
67          /// Action&lt;bool&gt; 型の Callback 添付プロパティを定義します。
68          /// </summary>
69          public static readonly DependencyProperty CallbackProperty =
    DependencyProperty.RegisterAttached("Callback", typeof(Action<bool>), typeof(OpenDialogBehavior),
    new PropertyMetadata(null, OnCallbackPropertyChanged));
70
```

```
71          /// <summary>
72          /// Callback 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 Action<bool> GetCallback(DependencyObject target)
77          {
78              return (Action<bool>)target.GetValue(CallbackProperty);
79          }
80
81          /// <summary>
82          /// Callback 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 SetCallback(DependencyObject target, Action<bool> value)
87          {
88              target.SetValue(CallbackProperty, value);
89          }
90          #endregion Callback 添付プロパティ
91
92          /// <summary>
93          /// Callback 添付プロパティ変更イベントハンドラ
94          /// </summary>
95          /// <param name="sender">イベント発行元</param>
96          /// <param name="e">イベント引数</param>
97          private static void OnCallbackPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
98          {
99              var callback = GetCallback(sender);
100             if (callback != null)
101             {
102                 var type = GetWindowType(sender);
103                 var obj = type.InvokeMember(null, System.Reflection.BindingFlags.CreateInstance,
    null, null, null);
104                 var child = obj as Window;
105                 if (child != null)
106                 {
107                     child.DataContext = GetDataContext(sender);
108                     var result = child.ShowDialog();
109                     callback(result.Value);
110                 }
111             }
112         }
113     }
114 }
```

11 行目から DataContext、38 行目から WindowType、65 行目から Callback 添付プロパティを定義しています。 97 行目に Callback 添付プロパティの変更イベントハンドラが定義されています。この OpenDialogBehavior 添付ビヘイビアは Callback 添付プロパティが変更されたときにある振る舞いをするものになっています。

その振る舞いこそ、ダイアログを表示するという処理になります。102 行目で与えられた WindowType 添付プロパティを取得し、その型情報を利用して 103 行目でインスタンスを生成しています。ここで生成したインスタンスは object 型にボックス化されているため、104 行目で Window クラスにキャストすることでアンボックス化しています。つまり、WindowType 添付プロパティは表示したいウィンドウの型情報が与えられることを想定しています。そして、DataContext 添付プロパティで与えられたオブジェクトをそのウィンドウの DataContext プロパティに設定し、108 行目で ShowDialog() メソッドを呼び出すことでウィンドウをダイアログとして表示しています。表示されたウィンドウを閉じると 109 行目に制御が戻り、与えられた Callback 添付プロパティのコールバック処理をおこないます。

このように定義された OpenDialogBehavior 添付ビヘイビアを使ってみましょう。まず MainView ウィンドウの XAML を次のように変更します。

コード 5.22：OpenDialogBehavior 添付ビヘイビアを MenuItem コントロールに適用する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:b="clr-namespace:YKWpfIntroduction.Practices.Views.Behaviors"
        xmlns:vw="clr-namespace:YKWpfIntroduction.Practices.Views"
        Title="MainView" Height="300" Width="300"
        b:WindowClosingBehavior.Callback="{Binding ClosingCallback}">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="ファイル (_F)">
                <MenuItem Header="開く(_O)"...>
                <Separator />
                <MenuItem Header="終了 (_X)" Command="{Binding ExitCommand}" />
            </MenuItem>
            <MenuItem Header="ヘルプ (_H)">
                <MenuItem Header="バージョン情報 (_V)"
                          Command="{Binding VersionDialogCommand}"
                          b:OpenDialogBehavior.WindowType="{x:Type vw:VersionView}"
                          b:OpenDialogBehavior.DataContext="{Binding VersionViewModel}"
                          b:OpenDialogBehavior.Callback="{Binding VersionDialogCallback}"/>
            </MenuItem>
        </Menu>

        <StatusBar DockPanel.Dock="Bottom">
        </StatusBar>

        <Grid Background="MediumSeaGreen">
        </Grid>
    </DockPanel>
</Window>
```

「ヘルプ」メニューに「バージョン情報」メニューを追加するために MenuItem コントロールを入れ子にしています。そして「バージョン情報」メニューを表す MenuItem コントロールに OpenDialogBehavior 添付ビヘイビアの各添付プロパティを設定しています。

OpenDialogBehavior.WindowType 添付プロパティには VersionView クラスの型情報を指定しています。このとき、名前空間のエイリアス vw を 5 行目のように定義しておく必要があります。OpenDialogBehavior.DataContext 添付プロパティには VersionView ウィンドウのデータコンテキストとして VersionViewModel クラスを指定しますが、View 側は ViewModel のインスタンスを持ちませんし、型情報も知り得ないため、MainViewModel から情報をもらうようにしています。OpenDialogBehavior.Callback 添付プロパティには表示したウィンドウを閉じた後のコールバック処理を MainViewModel から指定するようにしています。

今設定した添付プロパティに対する値に、VersionViewModel プロパティと VersionDialogCallback プロパティが出てきましたが、これはまだ MainViewModel クラスに定義していないプロパティなので、これらを次のように定義します。

コード 5.23：MainViewModel クラスに必要なプロパティを追加する

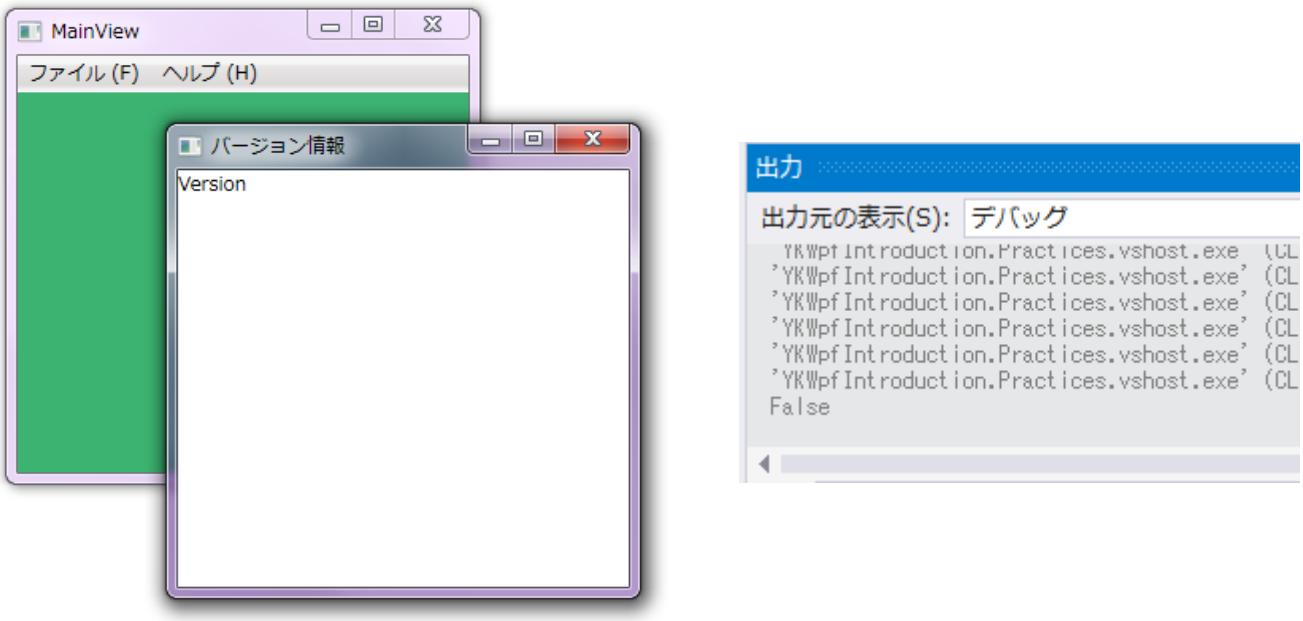
MainViewModel.cs

```
namespace YKWpfIntroduction.Practices.ViewModels
{
    using System;

    /// <summary>
    /// MainView ウィンドウに対するデータコンテキストを表します。
    /// </summary>
    internal class MainViewModel : NotificationObject
    {
        ファイルを開く
```

```
49          アプリケーションを終了する
88
89          #region バージョン情報を表示する
90          private VersionViewModel _versionViewModel = new VersionViewModel();
91          /// <summary>
92          /// VersionView ウィンドウに対するデータコンテキストを取得します。
93          /// </summary>
94          public VersionViewModel VersionViewModel
95          {
96              get { return this._versionViewModel; }
97          }
98
99          private DelegateCommand _versionDialogCommand;
100         /// <summary>
101         /// バージョン情報表示コマンドを取得します。
102         /// </summary>
103         public DelegateCommand VersionDialogCommand
104         {
105             get
106             {
107                 return this._versionDialogCommand ?? (this._versionDialogCommand = new
    DelegateCommand(
108                 _ =>
109                 {
110                     this.VersionDialogCallback = OnVersionDialog;
111                 }));
112             }
113         }
114
115         private Action<bool> _versionDialogCallback;
116         /// <summary>
117         /// バージョン情報表示コールバックを取得します。
118         /// </summary>
119         public Action<bool> VersionDialogCallback
120         {
121             get { return this._versionDialogCallback; }
122             private set { SetProperty(ref this._versionDialogCallback, value); }
123         }
124
125         /// <summary>
126         /// バージョン情報表示コールバック処理をおこないます。
127         /// </summary>
128         /// <param name="result"></param>
129         private void OnVersionDialog(bool result)
130         {
131             this.VersionDialogCallback = null;
132             System.Diagnostics.Debug.WriteLine(result);
133         }
134         #endregion バージョン情報を表示する
135     }
136 }
```

ここまでのコードで実行すると、「バージョン情報」メニューを選択すると VersionView ウィンドウがダイアログとして表示されるようになります。また、VersionView ウィンドウを閉じると MainViewModel クラスのコールバックメソッドが実行されるため、Visual Studio の出力ウィンドウにメッセージが表示されます。



(a) 「バージョン情報」メニューでダイアログが開く　　(b) コールバックメソッドがきちんと動作している

図 5.13：OpenDialogBehavior 添付ビヘイビアの動作確認

5.5.2　バージョン情報などをコードから取得する

前節ではバージョン情報ウィンドウを表示しましたが、肝心の中身はまだ表示していません。ここでは、C# コードからバージョン情報を取得し、VersionView ウィンドウに情報を渡すための準備をします。

ところで、アプリケーションのバージョン情報はどこに記述されているかと言うと、AssemblyInfo.cs というファイルの中で定義されています。このファイルはソリューションエクスプローラーの "Properties" のツリーを展開すると表示されます。

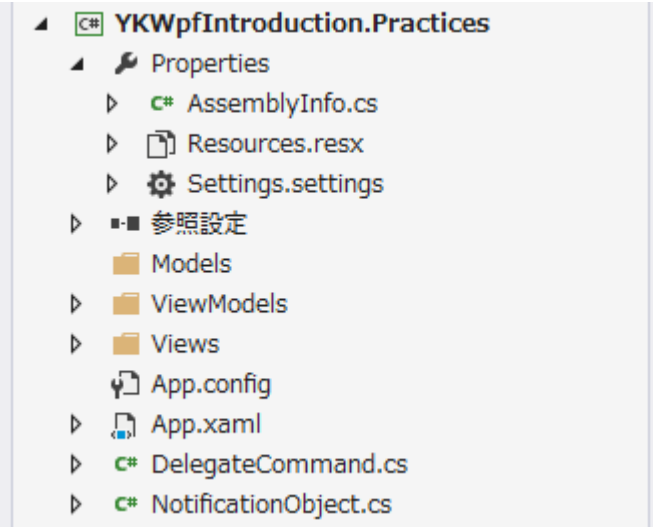


図 5.14：Properties の下に AssemblyInfo.cs ファイルがある

このファイルを開くと次のようなコードになっています。

コード 5.24：MainViewModel クラスに必要なプロパティを追加する

AsemmblyInfo.cs

```
1  using System.Reflection;
2  using System.Resources;
3  using System.Runtime.CompilerServices;
4  using System.Runtime.InteropServices;
5  using System.Windows;
```

```
6
7  // アセンブリに関する一般情報は以下の属性セットをとおして制御されます。
8  // アセンブリに関連付けられている情報を変更するには、
9  // これらの属性値を変更してください。
10 [assembly: AssemblyTitle("YKWpfIntroduction.Practices")]
11 [assembly: AssemblyDescription("")]
12 [assembly: AssemblyConfiguration("")]
13 [assembly: AssemblyCompany("Microsoft")]
14 [assembly: AssemblyProduct("YKWpfIntroduction.Practices")]
15 [assembly: AssemblyCopyright("Copyright © Microsoft 2016")]
16 [assembly: AssemblyTrademark("")]
17 [assembly: AssemblyCulture("")]
18
19 // ComVisible を false に設定すると、その型はこのアセンブリ内で COM コンポーネントから
20 // 参照不可能になります。COM からこのアセンブリ内の型にアクセスする場合は、
21 // その型の ComVisible 属性を true に設定してください。
22 [assembly: ComVisible(false)]
23
24 //ローカライズ可能なアプリケーションのビルドを開始するには、
25 //.csproj ファイルの <UICulture>CultureYouAreCodingWith</UICulture> を
26 //<PropertyGroup> 内部で設定します。たとえば、
27 //ソース ファイルで英語を使用している場合、<UICulture> を en-US に設定します。次に、
28 //下の NeutralResourceLanguage 属性のコメントを解除します。下の行の "en-US" を
29 //プロジェクト ファイルの UICulture 設定と一致するよう更新します。
30
31 //[assembly: NeutralResourcesLanguage("en-US", UltimateResourceFallbackLocation.Satellite)]
32
33
34 [assembly: ThemeInfo(
35     ResourceDictionaryLocation.None, //テーマ固有のリソース ディクショナリが置かれている場所
36     //(リソースがページ、
37     //またはアプリケーション リソース ディクショナリに見つからない場合に使用されます)
38     ResourceDictionaryLocation.SourceAssembly //汎用リソース ディクショナリが置かれている場所
39     //(リソースがページ、
40     //アプリケーション、またはいずれのテーマ固有のリソース ディクショナリにも見つからない場合に使用されま
   す)
41 )]
42
43
44 // アセンブリのバージョン情報は、以下の 4 つの値で構成されています:
45 //
46 //      Major Version
47 //      Minor Version
48 //      Build Number
49 //      Revision
50 //
51 // すべての値を指定するか、下のように '*' を使ってビルドおよびリビジョン番号を
52 // 既定値にすることができます:
53 // [assembly: AssemblyVersion("1.0.*")]
54 [assembly: AssemblyVersion("1.0.0.0")]
55 [assembly: AssemblyFileVersion("1.0.0.0")]
```

ファイル名が表しているように、ここではアセンブリに関する情報を定義しています。10 ～ 17 行目には一般情報が並んでおり、開発するアプリケーションの名称や著作権情報などが設定できます。また、バージョン情報は一番下の AssemblyVersion 属性または AssemblyFileVersion 属性で設定します。一般情報には以下のようなものがあります。

表 5.1：アセンブリに対する一般情報を表す属性

属性名	名称	説明
AssemblyTitle	説明	アプリケーションの概要を設定します。
AssemblyDescription	コメント	アプリケーションの詳細な説明を設定します。

AssemblyConfiguration	構成情報	"Debug" や "Release" といった構成情報を設定します。
AssemblyCompany	会社名	会社名を設定します。
AssemblyProduct	製品名	アプリケーションの正式名称を設定します。
AssemblyCopyright	著作権	著作権情報を設定します。
AssemblyTrademark	商標	アプリケーション名などを商標登録している場合にその旨を記述します。
AssemblyCulture	言語	"ja-JP" や "en-US" などのカルチャを設定します。リソースのみを含むサテライトアセンブリだけを指定できます。メインアセンブリの言語にする場合は "" としてニュートラル言語を設定します。
AssemblyVersion	アセンブリバージョン	.NET アセンブリ用のバージョン番号を設定します。.NET Framework のバージョン管理機能で使われます。この属性が指定されていない場合は AssemblyFileVersion 属性が使用されます。
AssemblyFileVersion	ファイルバージョン	Win32 ファイルシステム用のバージョン番号を設定します。この属性が指定されていない場合は AssemblyVersion 属性が使用されます。

バージョン情報は AssemblyVersion 属性だけあればいいので、その他の一般情報も含めて次のように変更します。

コード 5.25：MainViewModel クラスに必要なプロパティを追加する

AsemmblyInfo.cs

```
1  using System.Reflection;
2  using System.Resources;
3  using System.Runtime.CompilerServices;
4  using System.Runtime.InteropServices;
5  using System.Windows;
6
7  // アセンブリに関する一般情報は以下の属性セットをとおして制御されます。
8  // アセンブリに関連付けられている情報を変更するには、
9  // これらの属性値を変更してください。
10 [assembly: AssemblyTitle("WPF 実践")]
11 [assembly: AssemblyDescription("実践を通して WPF によるアプリケーション開発技術を習得するためのサンプル
   アプリケーションです。")]
12 #if DEBUG
13 [assembly: AssemblyConfiguration("Debug")]
14 #else
15 [assembly: AssemblyConfiguration("Release")]
16 #endif
17 [assembly: AssemblyCompany("YKSoftware")]
18 [assembly: AssemblyProduct("YKWpfIntroduction.Practices")]
19 [assembly: AssemblyCopyright("Copyright © 2016 YKSoftware")]
20 [assembly: AssemblyTrademark("")]
21 [assembly: AssemblyCulture("")]
22
23 // ComVisible を false に設定すると、その型はこのアセンブリ内で COM コンポーネントから
24 // 参照不可能になります。COM からこのアセンブリ内の型にアクセスする場合は、
25 // その型の ComVisible 属性を true に設定してください。
26 [assembly: ComVisible(false)]
27
28 //ローカライズ可能なアプリケーションのビルドを開始するには、
29 //.csproj ファイルの <UICulture>CultureYouAreCodingWith</UICulture> を
30 //<PropertyGroup> 内部で設定します。たとえば、
31 //ソース ファイルで英語を使用している場合、<UICulture> を en-US に設定します。次に、
32 //下の NeutralResourceLanguage 属性のコメントを解除します。下の行の "en-US" を
33 //プロジェクト ファイルの UICulture 設定と一致するよう更新します。
34
35 //[assembly: NeutralResourcesLanguage("en-US", UltimateResourceFallbackLocation.Satellite)]
36
37
```

```
38 [assembly: ThemeInfo(
39     ResourceDictionaryLocation.None, //テーマ固有のリソース ディクショナリが置かれている場所
40     //(リソースがページ、
41     //またはアプリケーション リソース ディクショナリに見つからない場合に使用されます)
42     ResourceDictionaryLocation.SourceAssembly //汎用リソース ディクショナリが置かれている場所
43     //(リソースがページ、
44     //アプリケーション、またはいずれのテーマ固有のリソース ディクショナリにも見つからない場合に使用されます)
45 )]
46
47
48 // アセンブリのバージョン情報は、以下の 4 つの値で構成されています:
49 //
50 //      Major Version
51 //      Minor Version
52 //      Build Number
53 //      Revision
54 //
55 // すべての値を指定するか、下のように '*' を使ってビルドおよびリビジョン番号を
56 // 既定値にすることができます:
57 // [assembly: AssemblyVersion("1.0.*")]
58 [assembly: AssemblyVersion("1.0.0.0")]
59 //[assembly: AssemblyFileVersion("1.0.0.0")]
```

ここで定義した一般情報を取得するクラスとして ProductInfo クラスを Model として新たに作成し、次のように定義します。

コード 5.26：バージョン情報などを取得するための ProductInfo クラスの定義

ProductInfo.cs

```
1  namespace YKWpfIntroduction.Practices.Models
2  {
3      using System;
4      using System.Reflection;
5
6      /// <summary>
7      /// プロダクト情報を取得するための静的なクラスを表します。
8      /// </summary>
9      public static class ProductInfo
10     {
11         /// <summary>
12         /// 自分のアセンブリを保持します。
13         /// </summary>
14         private static Assembly assembly = Assembly.GetExecutingAssembly();
15
16         private static string title;
17         /// <summary>
18         /// アプリケーションの名前を取得します。
19         /// </summary>
20         public static string Title
21         {
22             get { return title ?? (title =
    ((AssemblyTitleAttribute)Attribute.GetCustomAttribute(assembly,
    typeof(AssemblyTitleAttribute))).Title); }
23         }
24
25         private static string description;
26         /// <summary>
27         /// アプリケーションの詳細を取得します。
28         /// </summary>
29         public static string Description
```

```
30          {
31              get { return description ?? (description =
    ((AssemblyDescriptionAttribute)Attribute.GetCustomAttribute(assembly,
    typeof(AssemblyDescriptionAttribute))).Description); }
32          }
33
34          private static string company;
35          /// <summary>
36          /// アプリケーション開発元を取得します。
37          /// </summary>
38          public static string Company
39          {
40              get { return company ?? (company =
    ((AssemblyCompanyAttribute)Attribute.GetCustomAttribute(assembly,
    typeof(AssemblyCompanyAttribute))).Company); }
41          }
42
43          private static string product;
44          /// <summary>
45          /// アプリケーションのプロダクト名を取得します。
46          /// </summary>
47          public static string Product
48          {
49              get { return product ?? (product =
    ((AssemblyProductAttribute)Attribute.GetCustomAttribute(assembly,
    typeof(AssemblyProductAttribute))).Product); }
50          }
51
52          private static string copyright;
53          /// <summary>
54          /// アプリケーションのコピーライトを取得します。
55          /// </summary>
56          public static string Copyright
57          {
58              get { return copyright ?? (copyright =
    ((AssemblyCopyrightAttribute)Attribute.GetCustomAttribute(assembly,
    typeof(AssemblyCopyrightAttribute))).Copyright); }
59          }
60
61          private static string trademark;
62          /// <summary>
63          /// アプリケーションのトレードマークを取得します。
64          /// </summary>
65          public static string Trademark
66          {
67              get { return trademark ?? (trademark =
    ((AssemblyTrademarkAttribute)Attribute.GetCustomAttribute(assembly,
    typeof(AssemblyTrademarkAttribute))).Trademark); }
68          }
69
70          private static Version version;
71          /// <summary>
72          /// アプリケーションのバージョンを取得します。
73          /// </summary>
74          public static Version Version
75          {
76              get { return version ?? (version = assembly.GetName().Version); }
77          }
78
79          private static string versionString;
```

```
 80          /// <summary>
 81          /// アプリケーションのバージョン文字列を取得します。
 82          /// </summary>
 83          public static string VersionString
 84          {
 85              get { return versionString ?? (versionString = string.Format("{0}{1}{2}{3}",
     Version.ToString(3), IsBetaMode ? " β" : "", Version.Revision == 0 ? "" : " rev." + Version.Revision,
     IsDebugMode ? " Debug Mode" : "")); }
 86          }
 87
 88          /// <summary>
 89          /// ビルド時の CLR バージョン文字列を取得します。
 90          /// </summary>
 91          public static string CLRBuildVersion
 92          {
 93              get { return System.Reflection.Assembly.GetExecutingAssembly().ImageRuntimeVersion; }
 94          }
 95
 96          /// <summary>
 97          /// 実行中の CLR バージョン文字列を取得します。
 98          /// </summary>
 99          public static string CLRExecuteVersion
100          {
101              get { return System.Runtime.InteropServices.RuntimeEnvironment.GetSystemVersion(); }
102          }
103
104          /// <summary>
105          /// デバッグモードかどうか確認します。
106          /// </summary>
107          public static bool IsDebugMode
108          {
109  #if DEBUG
110              get { return true; }
111  #else
112              get { return false; }
113  #endif
114          }
115
116          /// <summary>
117          /// ベータ版かどうか確認します。
118          /// </summary>
119          public static bool IsBetaMode
120          {
121  #if BETA
122              get { return true; }
123  #else
124              get { return false; }
125  #endif
126          }
127      }
128  }
```

14 行目で現在実行中のアセンブリを取得し、これを元にリフレクション機能を利用しながら各情報を取得しています。バージョン情報は 76 行目のように取得できます。また、ここで得たバージョン情報を元に独自の書式のバージョン表記に変換したものを VersionString プロパティとして 83 行目に定義しています。85 行目を見ると複雑に見えますが、 "?" 記号による二項演算子で条件によって文字列を追加したりしなかたりしているだけなので、ひとつひとつ分解しながら見ていけばわかると思います。

ここで、107 行目や 119 行目に定義している IsDebugMode プロパティや IsBetaMode プロパティを見てみましょう。#if ～ #endif で条件分岐させることで、コンパイルシンボルに "DEBUG" が定義されているときは IsDebugMode プロパティが true となるようにしています。同様にコンパイルシンボルに "BETA" が定義されているときは IsBetaMode プ

ロパティが true となるようにしています。このプロパティを用いることで VersionString プロパティが返す文字列を切り替えるようにしています。つまり、Debug モードでビルドしたり、β版としてビルドした場合、バージョン情報にその情報が表示されるようにしています。

コンパイラシンボルはソリューションエクスプローラーの "Properties" をダブルクリックすると開く設定画面で設定できます。

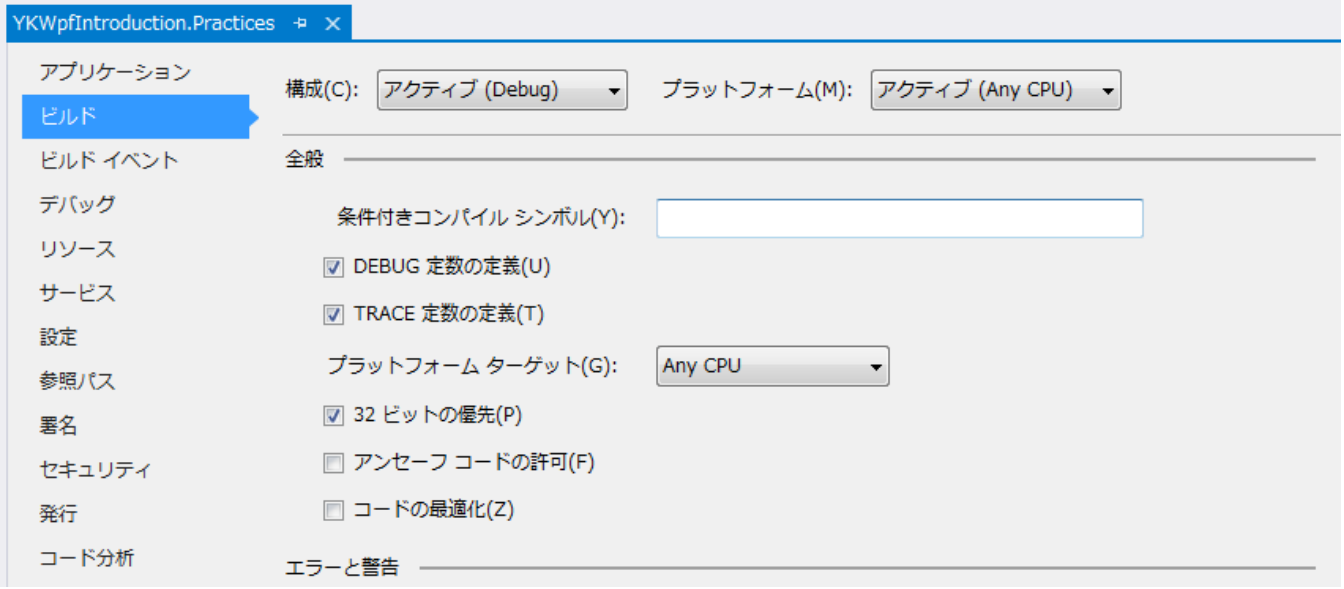


図 5.15：「ビルド」メニューの上部にある「条件付きコンパイラシンボル」で任意文字列を設定できる

Debug 構成に対する設定では、図 5.15 にあるように「DEBUG 定数の定義」チェックボックスがデフォルトで true になっているので、Debug 構成によるコンパイルでは自動的に "DEBUG" というコンパイルシンボルが認識されることになります。β版としての構成を作成した場合、「条件付きコンパイルシンボル」のテキストボックスに "BETA" と入力することで、"BETA" をコンパイルシンボルとして認識するようになります。

5.5.3　バージョン情報を表示させる

それでは VersionView ウィドウにバージョン情報を表示させるために、VersionViewModel クラスで必要なプロパティを公開しましょう。

コード 5.27：バージョン情報として表示するための情報を公開する

VersionViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      using YKWpfIntroduction.Practices.Models;
4
5      /// <summary>
6      /// VersionView ウィンドウに対するデータコンテキストを表します。
7      /// </summary>
8      internal class VersionViewModel : NotificationObject
9      {
10         /// <summary>
11         /// アプリケーションの正式名称を取得します。
12         /// </summary>
13         public string ProductName
14         {
15             get { return ProductInfo.Product; }
16         }
17
18         /// <summary>
19         /// アプリケーション名を取得します。
20         /// </summary>
```

```
21          public string Title
22          {
23              get { return ProductInfo.Title; }
24          }
25
26          /// <summary>
27          /// バージョン番号を取得します。
28          /// </summary>
29          public string Version
30          {
31              get { return "Ver." + ProductInfo.VersionString; }
32          }
33
34          /// <summary>
35          /// 著作権情報を取得します。
36          /// </summary>
37          public string Copyright
38          {
39              get { return ProductInfo.Copyright; }
40          }
41      }
42  }
```

これらのプロパティを使って VersionView ウィンドウを次のように変更します。

コード 5.28：バージョン情報として表示するための情報を公開する

VersionViewModel.cs

```
1  <Window x:Class="YKWpfIntroduction.Practices.Views.VersionView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="{Binding Title}"
5          SizeToContent="WidthAndHeight">
6      <StackPanel Margin="10">
7          <TextBlock Text="{Binding ProductName}" FontSize="20" TextAlignment="Center" />
8          <TextBlock Text="{Binding Version}" TextAlignment="Center" />
9          <Separator />
10         <TextBlock Text="{Binding Copyright}" TextAlignment="Center" />
11     </StackPanel>
12 </Window>
```

このようにすると、バージョン情報ウィンドウは次のように表示されます。

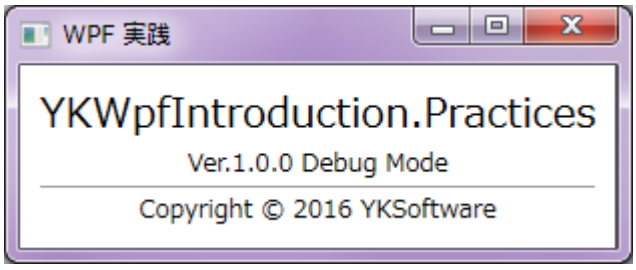


図 5.16：バージョン情報ウィンドウ

バージョンのリビジョン番号に 0 以外の数値を入れたり、コンパイルシンボルに BETA を定義したりすると表示されるバージョン番号の文字列が変化するのでトライしてみてください。

5.6　メニューにショートカットキーを割り当てる

「開く」メニューや「終了」メニューなどを選択することでそれぞれ ViewModel で公開されているコマンドが実行されるようになりました。しかし、多くのアプリケーションでは、このようなメニューを選択しなくても、キーボードのショートカットキーによって同じ処理を開始させることができます。例えば「開く」メニューは Ctrl+O というショートカットキーが一般的です。WPF でも同じようにショートカットキーを割り当てることができます。次のコードは「開く」メニューに Ctrl+O をショートカットキーに割り当てています。

コード 5.29：バージョン情報として表示するための情報を公開する

VersionViewModel.cs

```
 1 <Window x:Class="YKWpfIntroduction.Practices.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:YKWpfIntroduction.Practices.Views.Behaviors"
 5         xmlns:vw="clr-namespace:YKWpfIntroduction.Practices.Views"
 6         Title="MainView" Height="300" Width="300"
 7         b:WindowClosingBehavior.Callback="{Binding ClosingCallback}">
 8     <Window.InputBindings>
 9         <KeyBinding Modifiers="Control" Key="O" Command="{Binding OpenFileCommand}" />
10     </Window.InputBindings>
11 
12     <DockPanel>
13         <Menu DockPanel.Dock="Top">
14             <MenuItem Header="ファイル (_F)">
15                 <MenuItem Header="開く(_O)"
16                           Command="{Binding OpenFileCommand}"
17                           b:CommonDialogBehavior.Title="ファイルを開く"
18                           b:CommonDialogBehavior.Filter="画像ファイル (*.bmp; *.jpg;
    *.png)|*.bmp;*.jpg;*.png|すべてのファイル (*.*)|*.*"
19                           b:CommonDialogBehavior.Multiselect="False"
20                           b:CommonDialogBehavior.Callback="{Binding DialogCallback}"
21                           />
22                 <Separator />
23                 <MenuItem Header="終了 (_X)" Command="{Binding ExitCommand}" />
24             </MenuItem>
25             <MenuItem Header="ヘルプ (_H)"...>
32         </Menu>
33 
34         <StatusBar DockPanel.Dock="Bottom">
35         </StatusBar>
36 
37         <Grid Background="MediumSeaGreen">
38         </Grid>
39     </DockPanel>
40 </Window>
```

8 行目のように、InputBindings プロパティに KeyBinding クラスを羅列することでコマンドにショートカットキーを割り当てることができます。ここでは「開く」メニューと同じコマンドに対して Ctrl+O というショートカットキーを割り当てたいので、KeyBinding クラスの Modifiers プロパティに "Control"、Key プロパティに "O" を指定することで Ctrl+O というショートカットキーを表し、Command プロパティに「開く」メニューに割り当てているコマンドと同じ OpenFileCommand プロパティをデータバインディングしています。

このまま実行し、メニューを選択せずに Ctrl+O キーを押すと、確かにファイルを開くダイアログが表示されるようになります。しかし、「開く」メニューを表示しても、そこにショートカットキーが "Ctrl+O" であることが表示されていません。そこで、メニューにもショートカットキーが "Ctrl+O" であることを表示させるために、「開く」メニューに該当する MenuItem コントロールを変更します。

コード 5.30：バージョン情報として表示するための情報を公開する

VersionViewModel.cs

```
15                 <MenuItem Header="開く(_O)"
```

```
16                          InputGestureText="Ctrl+O"
17                          Command="{Binding OpenFileCommand}"
18                          b:CommonDialogBehavior.Title="ファイルを開く"
19                          b:CommonDialogBehavior.Filter="画像ファイル (*.bmp; *.jpg;
    *.png)|*.bmp;*.jpg;*.png|すべてのファイル (*.*)|*.*"
20                          b:CommonDialogBehavior.Multiselect="False"
21                          b:CommonDialogBehavior.Callback="{Binding DialogCallback}"
22                          />
```

16 行目にある InputGestureText プロパティで文字列を追加しています。このプロパティを設定することで、図 5.17 のように指定された文字列が表示されるようになります。

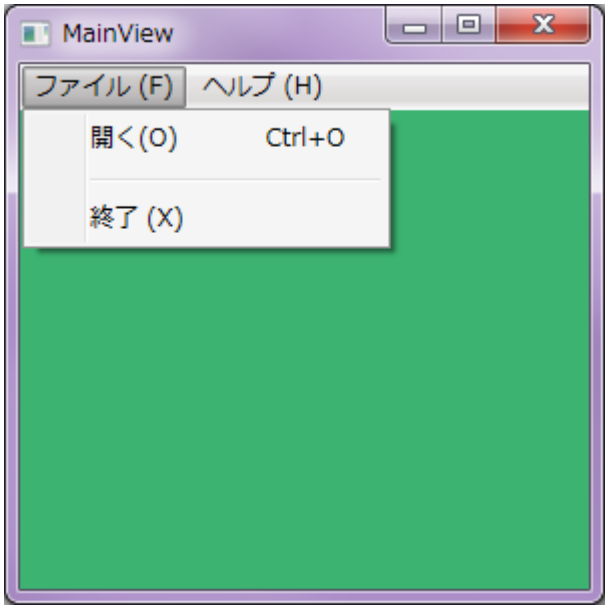


図 5.17：指定された文字列がメニューに表示されるようになる

InputGestureText プロパティはあくまでも表示する文字列を指定するためだけのものなので、実際にショートカットキーを割り当てるときは必ず KeyBinding クラスを使って設定しましょう。

5.7　ステータスバーにコンテンツを表示させる

ステータスバーにコンテンツを表示させるときは、StatusBarItem コントロールを使用しましょう。

コード 5.31：ステータスバーにコントロールを配置する

MainView.xaml

```
 1 <Window x:Class="YKWpfIntroduction.Practices.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:YKWpfIntroduction.Practices.Views.Behaviors"
 5         xmlns:vw="clr-namespace:YKWpfIntroduction.Practices.Views"
 6         Title="MainView" Height="300" Width="300"
 7         b:WindowClosingBehavior.Callback="{Binding ClosingCallback}">
 8     <Window.InputBindings...>
11
12     <DockPanel>
13         <Menu DockPanel.Dock="Top"...>
34
35         <StatusBar DockPanel.Dock="Bottom">
36             <StatusBarItem DockPanel.Dock="Right">
37                 <TextBlock Text="{Binding CurrentTime, StringFormat='yyyy/MM/dd HH:mm'}" />
38             </StatusBarItem>
39             <Separator DockPanel.Dock="Right" />
40             <TextBlock />
41         </StatusBar>
42
43         <Grid Background="MediumSeaGreen">
44         </Grid>
45     </DockPanel>
46 </Window>
```

StatusBarItem コントロールをコンテナとして、その子要素として配置したいコントロールを記述します。上記の例では現在時刻を TextBlock コントロールで表示しようとしています。CurrentTime プロパティは MainViewModel クラスで用意する必要があります。

StatusBarItem コントロールに対して DockPanel.Dock 添付プロパティを指定すると、StatusBar コントロールの中で指定した位置に配置されるようになります。StatusBar コントロールは内部に DockPanel と同じように子要素を配置する仕組みを備えています。また、Separator コントロールを使うと、簡単に区切り線を追加できます。ここでは現在時刻を右端に表示し、その境界線として区切り線を表示させています。また、残りの領域をすべて別のコントロールで占有させるようにするために、40 行目で TextBlock コントロールをダミーとして配置しています。

続いて現在時刻を取得するために MainViewModel クラスを次のように変更します。

コード 5.32：ステータスバーにコントロールを配置する

MainView.xaml

```
  1 namespace YKWpfIntroduction.Practices.ViewModels
  2 {
  3     using System;
  4     using System.Timers;
  5
  6     /// <summary>
  7     /// MainView ウィンドウに対するデータコンテキストを表します。
  8     /// </summary>
  9     internal class MainViewModel : NotificationObject
 10     {
 11         ファイルを開く
 49
 50         アプリケーションを終了する
 89
 90         バージョン情報を表示する
136
```

```
137         #region 現在時刻を取得する
138         /// <summary>
139         /// 現在時刻を更新するためのタイマー
140         /// </summary>
141         private Timer _timer;
142 
143         private DateTime _currentTime;
144         /// <summary>
145         /// 現在時刻を取得します。
146         /// </summary>
147         public DateTime CurrentTime
148         {
149             get
150             {
151                 if (this._timer == null)
152                 {
153                     this._currentTime = DateTime.Now;
154 
155                     this._timer = new Timer(1000);
156                     this._timer.Elapsed += (_, __) =>
157                     {
158                         this.CurrentTime = DateTime.Now;
159                     };
160                     this._timer.Start();
161                 }
162                 return this._currentTime;
163             }
164             private set { SetProperty(ref this._currentTime, value); }
165         }
166         #endregion 現在時刻を取得する
167     }
168 }
```

現在時刻を取得するためのプロパティを CurrentTime プロパティとして 147 行目で定義しています。時刻は刻々と変化していくため、タイマーによってその変化を捉えるようにしています。155 行目で、 1[s] 間隔で Timer.Elapsed イベントが発生するようにインスタンスを生成し、その Elapsed イベントハンドラをラムダ式による匿名関数で与えています。イベントハンドラでの処理は CurrentTime プロパティを現在時刻に更新することのみなので、158 行目だけとなります。タイマーの設定が終了したら実際に動作させるため、160 行目で Start() メソッドを呼び出しています。

Timer クラスをインスタンス化するタイミングは初めて CurrentTime プロパティを取得されるときとなっています。このとき、Timer クラスの設定をおこなうだけでは、そのとき取得する _currentTime の値は一度も更新されていないため、DateTime 構造体の既定値のままとなっています。これを防止するために 153 行目で一度現在時刻を取得させています。

アプリケーションを実行すると、図 5.18 のように表示されるようになります。

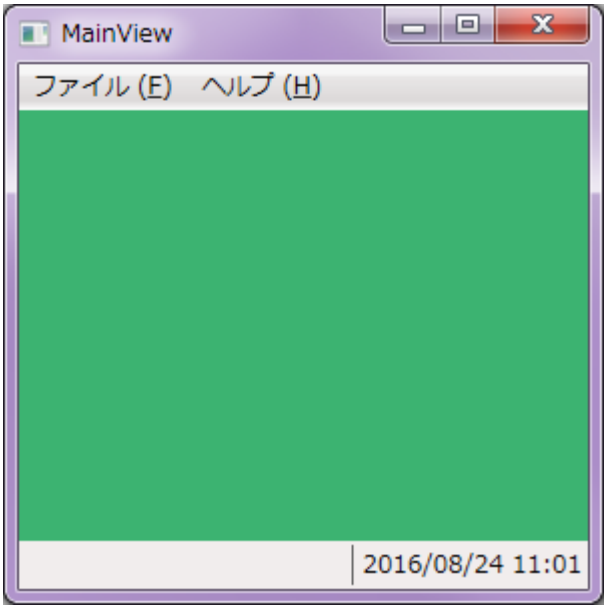


図 5.18：ステータスバーの右端に現在時刻が表示される

5.8 まとめ

メニューとステータスバーを持つ UI を作成する場合は DockPanel パネルが最適です。

メニューは Menu コントロールに MenuItem コントロールを入れ子にすることで構築でき、それぞれに Command プロパティをデータバインディングすることで ViewModel 側で処理がおこなえるようになります。

ファイルを開いたり、アプリケーションを終了したりするとき、View 側で処理しなければならないことがありますが、添付ビヘイビアを用いることで ViewModel 側のコールバックメソッドを View 側から参照することができるようになるため、MVVM パターンにしたがった責務の分担をおこなうことができます。

子ウィンドウを開く方法はいくつかありますが、ここで紹介したように添付ビヘイビアを用いた方法で開くことができます。

ステータスバーは StatusBar コントロールに StatusBarItem コントロールを羅列することでコンテンツを表示できます。StatusBarItem に DockPanel.Dock 添付プロパティを設定することでその配置を指定できます。

6 ItemsControl で学ぶ基礎

ItemsControl コントロールとは、コレクションの各要素を並べて表示するための基本的なコントロールです。このコントロールから派生したコントロールとして ComboBox、ListBox、DataGrid、TreeView コントロールなどが良く知られています。ItemsControl コントロールを使いこなすことで、これらのコントロールを自由にカスタマイズすることができるようになるため、UI 構築技術の幅が一気に広がるので、必ず修得しましょう。WPF アプリケーションプロジェクトを新たに作成してから進めましょう。

6.1 ItemsControl コントロールでコレクションを並べてみよう

ItemsControl コントロールで並べるためのコレクションデータを用意しましょう。まず Model として Person クラスを定義します。

コード 6.1：人物情報を表すクラスを定義する

Person.cs

```
1  namespace YKWpfIntroduction.Practices.Models
2  {
3      /// <summary>
4      /// 人物を表します。
5      /// </summary>
6      internal class Person
7      {
8          /// <summary>
9          /// 名字を取得または設定します。
10         /// </summary>
11         public string FamilyName { get; set; }
12
13         /// <summary>
14         /// 名前を取得または設定します。
15         /// </summary>
16         public string FirstName { get; set; }
17
18         /// <summary>
19         /// 氏名を取得します。
20         /// </summary>
21         public string FullName { get { return this.FamilyName + this.FirstName; } }
22
23         /// <summary>
24         /// 性別を取得または設定します。
25         /// </summary>
26         public Gender Gender { get; set; }
27
28         /// <summary>
29         /// 年齢を取得または設定します。
30         /// </summary>
31         public int Age { get; set; }
32
33         /// <summary>
34         /// 認証済みかどうかを取得または設定します。
35         /// </summary>
36         public bool IsAuthenticated { get; set; }
37     }
38 }
```

氏名や年齢などのプロパティを持っています。Gender というのは性別を表し、次のように定義されている列挙型です。

コード 6.2：性別を表す列挙型を定義する

Gender.cs

```
namespace YKWpfIntroduction.Practices.Models
{
    /// <summary>
    /// 性別を表します。
    /// </summary>
    internal enum Gender
    {
        /// <summary>
        /// 男性を表します。
        /// </summary>
        Male,

        /// <summary>
        /// 助成を表します。
        /// </summary>
        Female,

        /// <summary>
        /// 性別不明を表します。
        /// </summary>
        Unknown,
    }
}
```

このように定義した Person クラスを ViewModel で複数インスタンス化し、コレクションとして保持します。

コード 6.3：人物情報のコレクションを持つ ViewModel

MainViewModel.cs

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

    /// <summary>
    /// MainView ウィンドウに対するデータコンテキストを表します。
    /// </summary>
    internal class MainViewModel : NotificationObject
    {
        /// <summary>
        /// 新しいインスタンスを生成します。
        /// </summary>
        public MainViewModel()
        {
            this.People = Enumerable.Range(0, 100).Select(x => new Person()
            {
                FamilyName = "田中",
                FirstName = x + "太郎",
                Age = x,
                Gender = (x % 2 == 0) ? Gender.Male : Gender.Female,
                IsAuthenticated = x % 3 == 0,
            }).ToList();
        }

        private List<Person> _people;
        /// <summary>
```

```
29          /// 人物情報のコレクションを取得します。
30          /// </summary>
31          public List<Person> People
32          {
33              get { return this._people; }
34              private set { SetProperty(ref this._people, value); }
35          }
36      }
37  }
```

17 行目で People プロパティを初期化しています。ここでは「2.8 System.Linq による拡張メソッド」で説明した Linq による拡張メソッドを使用して 100 人分の人物情報を生成しています。

この People プロパティを使って Person クラスのコレクションを表示してみましょう。MainView ウィンドウの XAML を次のように編集します。

コード 6.4：ItemsControl コントロールでコレクションを並べる

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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      <ItemsControl ItemsSource="{Binding People}" />
6  </Window>
```

ItemsSource プロパティに並べたいコレクションを指定します。このまま実行すると次のようにクラス名が並んでしまいます。

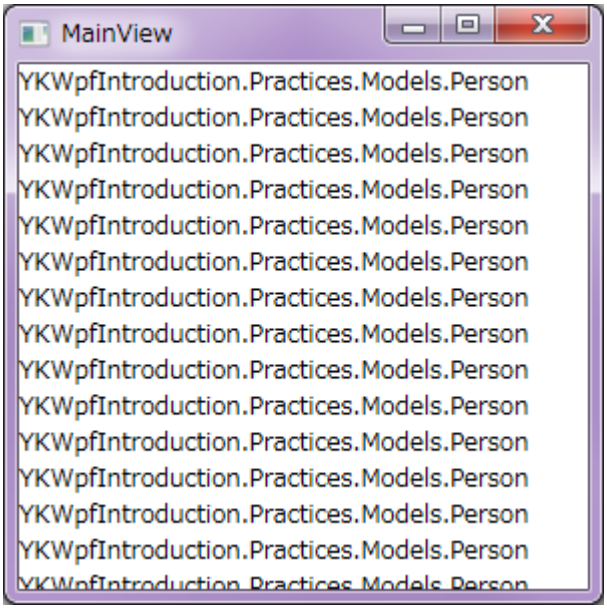


図 6.1：Person クラスの完全修飾名が並んでしまう

ItemsControl コントロールは、何も指定しない場合、指定されたコレクションの各アイテムを文字列として変換して並べてしまいます。Person クラスは文字列に変換する機能を持っていないため、クラス名となって表示されるようになります。

これでは何の情報かまったくわかりません。そこで、各アイテムを表現する方法を指定するために、ItemsControl.ItemTemlate プロパティに DataTemplate クラスを指定しましょう。

コード 6.5：各アイテムの表現方法を指定する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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      <ItemsControl ItemsSource="{Binding People}">
6          <ItemsControl.ItemTemplate>
7              <DataTemplate>
8                  <TextBlock Text="{Binding FullName}" />
9              </DataTemplate>
10         </ItemsControl.ItemTemplate>
11     </ItemsControl>
12 </Window>
```

DataTemplate クラス内に、各アイテムをどのように表示するかを指定します。ここでは TextBlock コントロールを用いて氏名を表示するようにしています。ここで、各アイテムのデータコンテキストはコレクションの各要素となることに注意しなければなりません。8 行目だけを見ると MainViewModel クラスの FullName プロパティとデータバインディングしているように見えますが、そうではありません。ItemsControl.ItemTemplate 内の DataTemplate で表現する UI のデータコンテキストは ItemsControl.ItemsSource プロパティに指定されているコレクションの各アイテムとなります。このサンプルの場合、Person クラスのコレクションを指定しているため、DataTemplate 内のデータコンテキストは Person クラスとなります。したがって、8 行目の FullName プロパティは Person クラスの FullName プロパティを指します。

このように指定すると、氏名が羅列されるようになります。

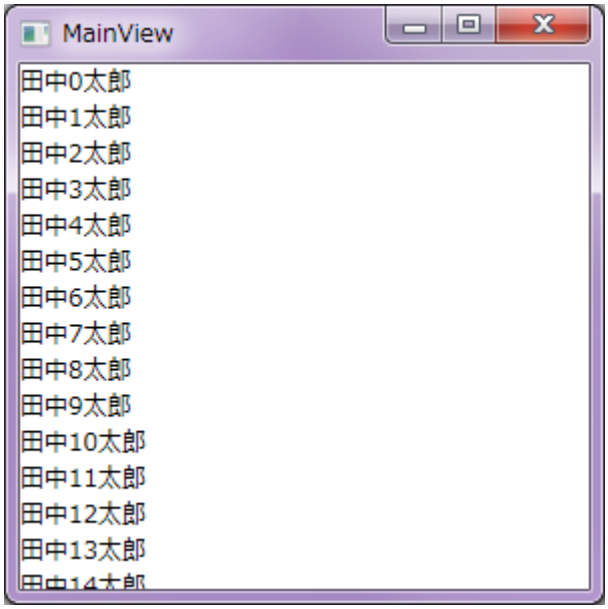


図 6.2：氏名が羅列されるようになる

ところで、人物データは 100 人分用意しましたが、ウィンドウが狭いため、すべてのデータを閲覧できません。表示領域からデータがはみ出てしまう場合は、スクロールバーを表示するようにしましょう。スクロールバーを表示するコントロールに ScrollViewer コントロールがあるので、これを次のように使います。

コード 6.6：ScrollViewer コントロールの中にアイテムを並べるようにする

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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      <ItemsControl ItemsSource="{Binding People}">
6          <ItemsControl.Template>
7              <ControlTemplate>
8                  <ScrollViewer>
9                      <ItemsPresenter />
10                 </ScrollViewer>
11             </ControlTemplate>
12         </ItemsControl.Template>
13         <ItemsControl.ItemTemplate>
14             <DataTemplate>
15                 <TextBlock Text="{Binding FullName}" />
```

```
16              </DataTemplate>
17          </ItemsControl.ItemTemplate>
18      </ItemsControl>
19  </Window>
```

このように ItemsControl コントロールのテンプレートを指定することで、アイテムを並べるためのコンテナ ItemsPresenter コントロールを ScrollViewer コントロールの中に入れるようにすることで、ItemsPresenter コントロールで並べられたアイテムが表示領域をはみ出たときに、ScrollViewer の機能によってスクロールできるようにしています。

6.2 ItemsControl コントロールのカスタマイズ

ItemsControl コントロールには、その表現方法をカスタマイズするためのプロパティとして次のようなものが用意されています。また、これらを図に表すとのようになります。

表 6.1：ItemsControl コントロールをカスタマイズするためのプロパティ

プロパティ名	説明
Template	ItemsControl 内部の構成そのものを決定します。
ItemsPanel	アイテムを並べるためのパネルを指定します。
ItemContainerStyle	各アイテムを並べるために用意されるコンテナの Style を指定します。
ItemTemplate	各アイテムを表現するテンプレートを指定します。

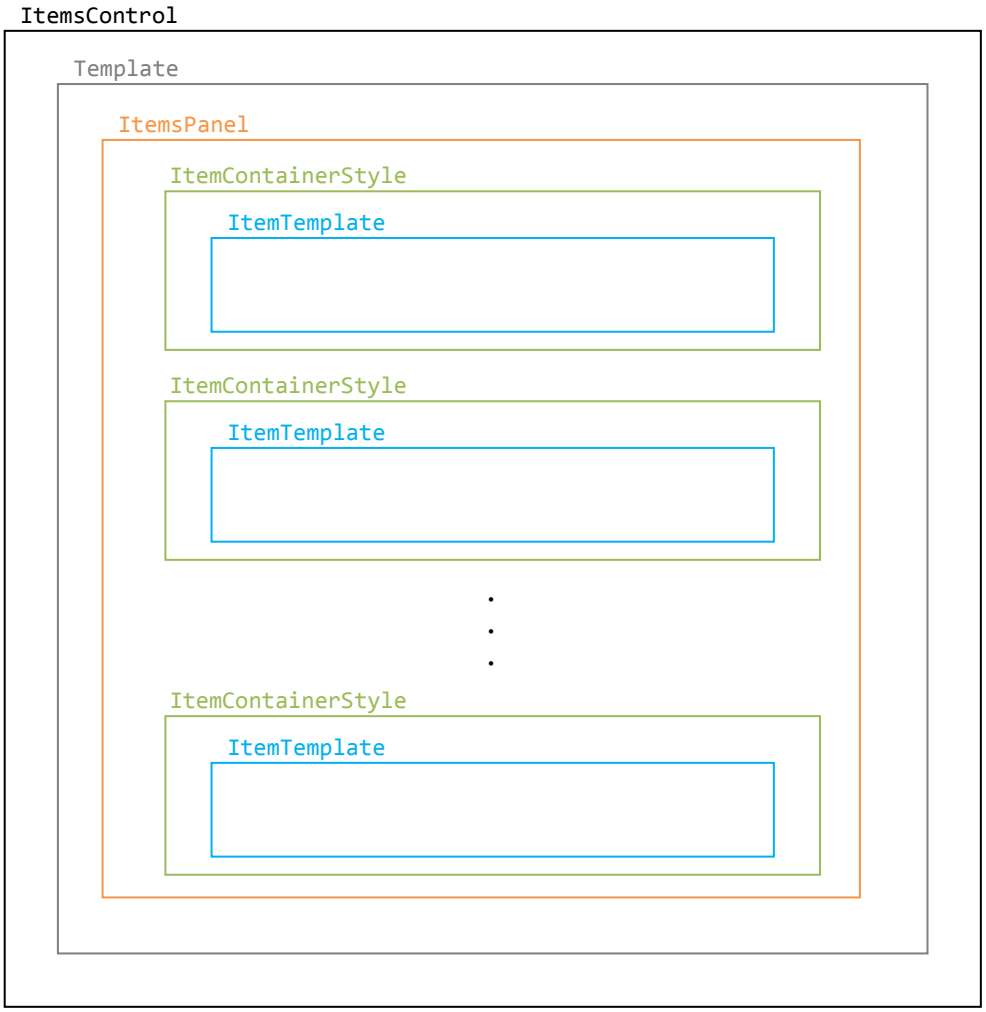


図 6.3：ItemsControl コントロールにおけるそれぞれのプロパティの役割

ItemsPanel プロパティで、各アイテムを並べるためのパネルを指定します。デフォルトでは StackPanel パネルが指定されているため、各アイテムが縦に並ぶことになります。

ItemContainerStyle プロパティで、各アイテムを並べるために用意されるコンテナの Style を指定します。ItemsControl コントロールによって用意されるコンテナは ContentPresenter コントロールですが、ListBox コントロールの場合は ListBoxItem コントロール、ComboBox コントロールの場合は ComboBoxItem コントロールがそれぞれコンテナとなります。

6.3 ListBox コントロールをカスタマイズしてみる

前節で説明したカスタマイズするための各プロパティについて、ListBox コントロールを用いたサンプルコードを掲載します。

コード 6.7：ListBox コントロールをカスタマイズする

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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="700">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <TextBlock Grid.Row="0" Grid.Column="0" Text="標準の ListBox" />
        <TextBlock Grid.Row="0" Grid.Column="1" Text="カスタマイズした ListBox" />

        <ListBox Grid.Row="1" Grid.Column="0" ItemsSource="{Binding People}" />

        <ListBox Grid.Row="1" Grid.Column="1" ItemsSource="{Binding People}">
            <ListBox.Template>
                <ControlTemplate TargetType="{x:Type ListBox}">
                    <Border>
                        <ItemsPresenter />
                    </Border>
                </ControlTemplate>
            </ListBox.Template>
            <ListBox.ItemsPanel>
                <ItemsPanelTemplate>
                    <WrapPanel Orientation="Horizontal" />
                </ItemsPanelTemplate>
            </ListBox.ItemsPanel>
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel>
                        <TextBlock>
                            <TextBlock.Text>
                                <MultiBinding StringFormat="氏名:{0} ({1})">
                                    <Binding Path="FullName" />
                                    <Binding Path="Age" />
                                </MultiBinding>
                            </TextBlock.Text>
                        </TextBlock>
                        <TextBlock Text="{Binding IsAuthenticated}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
            <ListBox.ItemContainerStyle>
                <Style TargetType="{x:Type ListBoxItem}">
                    <Setter Property="Template">
                        <Setter.Value>
                            <ControlTemplate TargetType="{x:Type ContentControl}">
                                <Border Background="{TemplateBinding Background}" Margin="1">
                                    <ContentPresenter />
```

```
55                                </Border>
56                            </ControlTemplate>
57                        </Setter.Value>
58                    </Setter>
59                    <Style.Triggers>
60                        <Trigger Property="IsMouseOver" Value="True">
61                            <Setter Property="Background" Value="LightGray" />
62                        </Trigger>
63                        <Trigger Property="IsSelected" Value="True">
64                            <Setter Property="Background" Value="Plum" />
65                        </Trigger>
66                    </Style.Triggers>
67                </Style>
68            </ListBox.ItemContainerStyle>
69        </ListBox>
70    </Grid>
71 </Window>
```

比較として、カスタマイズする前の ListBox コントロールも同時に表示するように Grid パネルで分けています。

カスタマイズした ListBox コントロールは 20 行目から記述されています。各アイテムを並べるためのパネルとして WrapPanel パネルを 30 行目で指定しています。各アイテムは 34 行目に指定されている DataTemplate にしたがって表示されます。ここでは StackPanel パネル以下に並ぶ TextBlock コントロールで氏名、年齢および認証済みかどうかをテキストで表示するようにしています。

ListBox コントロールの各アイテムのコンテナは ListBoxItem コントロールとなるため、ItemContainerStyle の TargetType プロパティには ListBoxItem クラスを指定しています。その Template プロパティとして 52 行目の ControlTemplate を指定しています。ここではマウスが上に乗った状態か、またはアイテムが選択された状態であることをわかりやすくするために、背景色を変更するようにしています。59 ～ 66 行目の Triggers プロパティで動的に背景色を変更するように指定していますが、ここでは説明を割愛します。

このアプリケーションを実行すると次のようになります。アイテムの並びが WrapPanel パネルによって並んでいることがわかります。また、実際にアイテムを選択すると背景色が変更されていることがわかります。

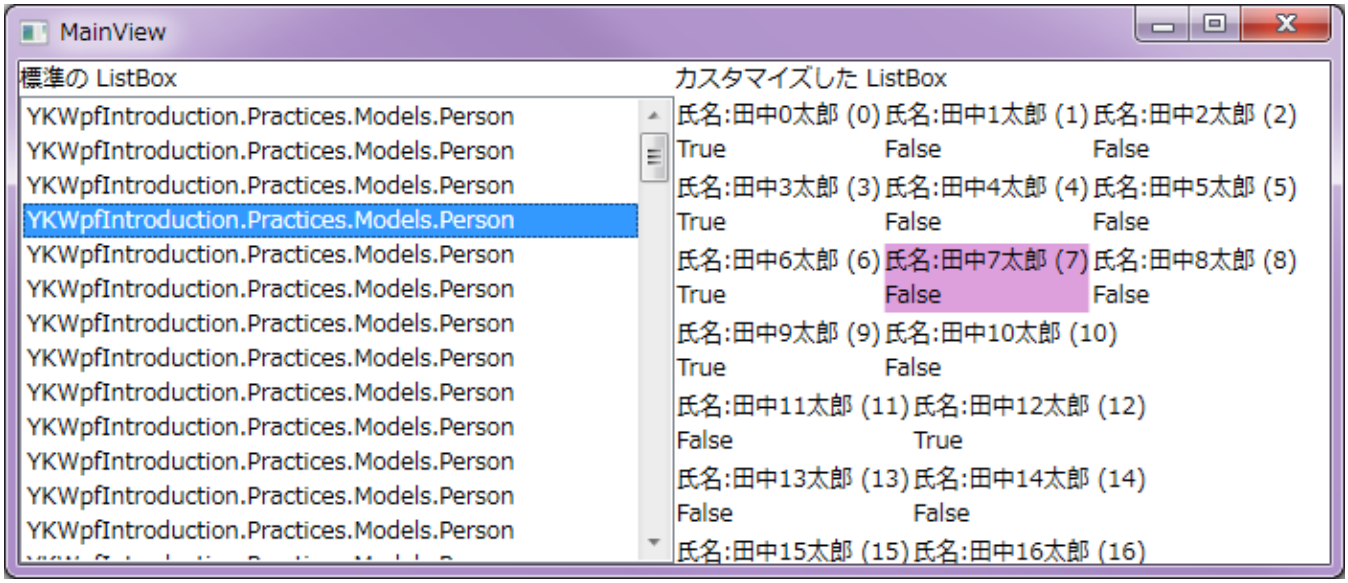


図 6.4：ListBox がカスタマイズされた様子

6.4 ComboBox コントロールをカスタマイズしてみる

前節のカスタマイズを ComboBox コントロールに適用するとどうなるでしょうか。コード 6.7 で ListBox コントロールの ItemTemplate プロパティに適用した DataTemplate をそのまま ComboBox コントロールの ItemTemplate プロパティに適用してみましょう。

コード 6.8：ComboBox コントロールをカスタマイズする

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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="700">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <StackPanel Grid.Column="0">
            <TextBlock Text="標準の ComboBox" />
            <ComboBox ItemsSource="{Binding People}" SelectedIndex="0" />
        </StackPanel>

        <StackPanel Grid.Column="1">
            <TextBlock Text="カスタマイズした ComboBox" />
            <ComboBox ItemsSource="{Binding People}" SelectedIndex="0">
                <ComboBox.ItemTemplate>
                    <DataTemplate>
                        <StackPanel>
                            <TextBlock>
                                <TextBlock.Text>
                                    <MultiBinding StringFormat="氏名:{0} ({1})">
                                        <Binding Path="FullName" />
                                        <Binding Path="Age" />
                                    </MultiBinding>
                                </TextBlock.Text>
                            </TextBlock>
                            <TextBlock Text="{Binding IsAuthenticated}" />
                        </StackPanel>
                    </DataTemplate>
                </ComboBox.ItemTemplate>
                <ComboBox.ItemContainerStyle>
                    <Style TargetType="{x:Type ComboBoxItem}">
                        <Setter Property="Template">
                            <Setter.Value>
                                <ControlTemplate TargetType="{x:Type ContentControl}">
                                    <Border Background="{TemplateBinding Background}" Margin="1">
                                        <ContentPresenter />
                                    </Border>
                                </ControlTemplate>
                            </Setter.Value>
                        </Setter>
                        <Style.Triggers>
                            <Trigger Property="IsMouseOver" Value="True">
                                <Setter Property="Background" Value="LightGray" />
                            </Trigger>
                            <Trigger Property="IsSelected" Value="True">
                                <Setter Property="Background" Value="Plum" />
                            </Trigger>
                        </Style.Triggers>
                    </Style>
```

```
54                  </ComboBox.ItemContainerStyle>
55              </ComboBox>
56          </StackPanel>
57      </Grid>
58  </Window>
```

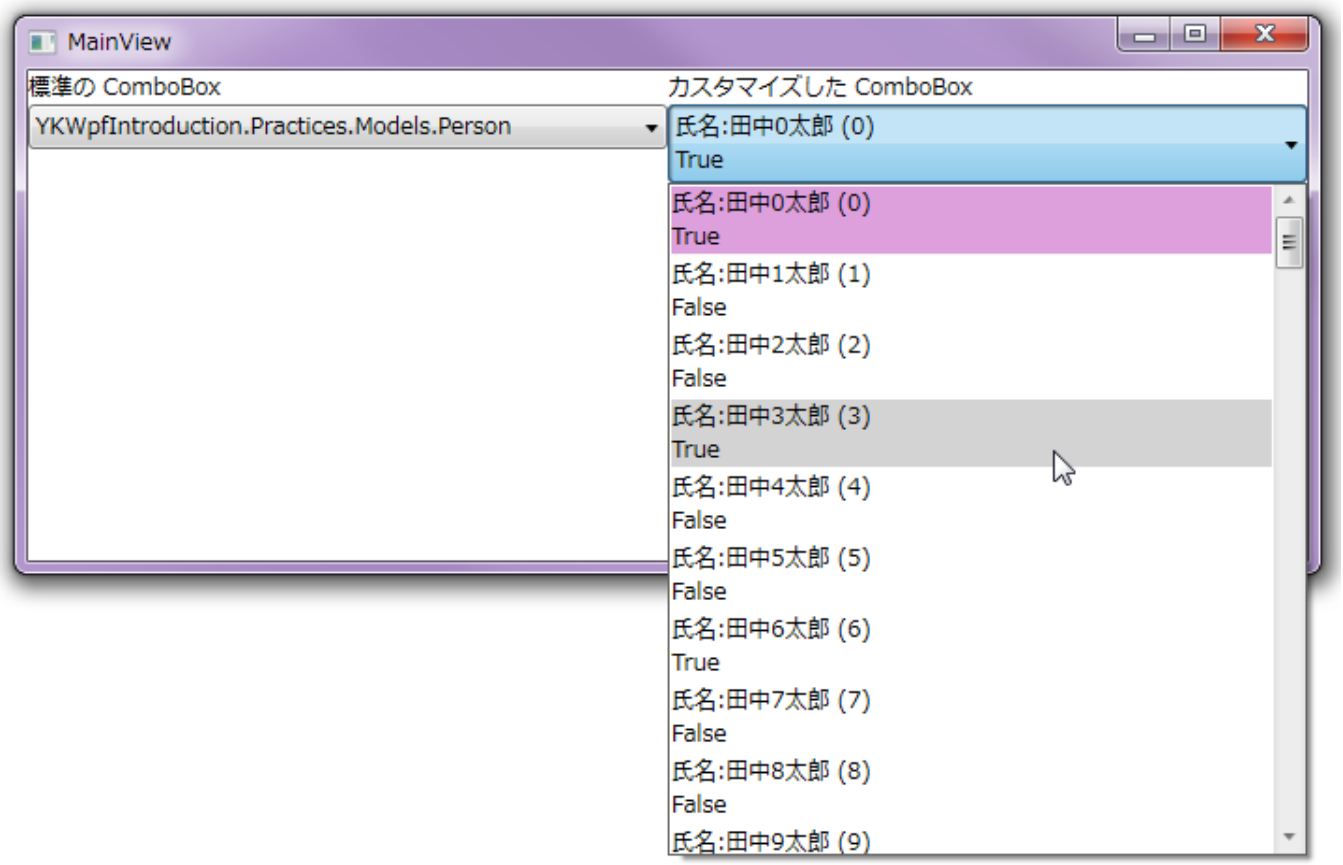


図 6.5：ComboBox がカスタマイズされた様子

今までは 1 行でしか表示できなかった ComboBox コントロールも、DataTemplate にしたがって様々なコンテンツで表現できるようになります。

6.5 コレクションの子要素を追加/削除してみよう

改めて MainViewModel を次のように変更し、コレクションの子要素を追加したり削除したりできるようにしましょう。

コード 6.9：コレクションの要素を追加/削除するコマンドを用意する

MainViewModel.cs

```
namespace YKWpfIntroduction.Practices.ViewModels
{
    using System.Collections.Generic;
    using YKWpfIntroduction.Practices.Models;

    /// <summary>
    /// MainView ウィンドウに対するデータコンテキストを表します。
    /// </summary>
    internal class MainViewModel : NotificationObject
    {
        private List<Person> _people = new List<Person>();
        /// <summary>
        /// 人物情報のコレクションを取得します。
        /// </summary>
        public List<Person> People
        {
            get { return this._people; }
            private set { SetProperty(ref this._people, value); }
        }

        private int _count;

        private DelegateCommand _addCommand;
        /// <summary>
        /// 追加コマンドを取得します。
        /// </summary>
        public DelegateCommand AddCommand
        {
            get
            {
                return this._addCommand ?? (this._addCommand = new DelegateCommand(
                _ =>
                {
                    this._count++;
                    var person = new Person()
                    {
                        FamilyName = "田中",
                        FirstName = this._count + "太郎",
                        Age = this._count,
                    };
                    this.People.Add(person);
                    this.DeleteCommand.RaiseCanExecuteChanged();

                    System.Diagnostics.Debug.WriteLine(person.FullName + " を追加しました。");
                }));
            }
        }

        private DelegateCommand _deleteCommand;
        /// <summary>
        /// 削除コマンドを取得します。
        /// </summary>
        public DelegateCommand DeleteCommand
        {
            get
```

```
56              {
57                  return this._deleteCommand ?? (this._deleteCommand = new DelegateCommand(
58                  _ =>
59                  {
60                      this.People.RemoveAt(this.People.Count - 1);
61                      this.DeleteCommand.RaiseCanExecuteChanged();
62                  },
63                  _ => this.People.Count > 0));
64              }
65          }
66      }
67  }
```

コンストラクタ内でコレクションの要素を用意せず、AddCommand プロパティのコマンドによってコレクションの要素をひとつずつ追加できるようにしています。また、DeleteCommand プロパティのコマンドによってコレクションの最後尾の要素をひとつずつ削除できるようにしています。どちらのコマンドも、実行すると People コレクションの数が変更されることで DeleteCommand プロパティの実行可能判別条件の結果に影響するため、RaiseCanExecuteChanged() メソッドを呼び出してその変更を通知するようにしています。44 行目には、コレクションに要素を追加したときに Visual Studio の出力ウィンドウにメッセージを表示するようにしています。

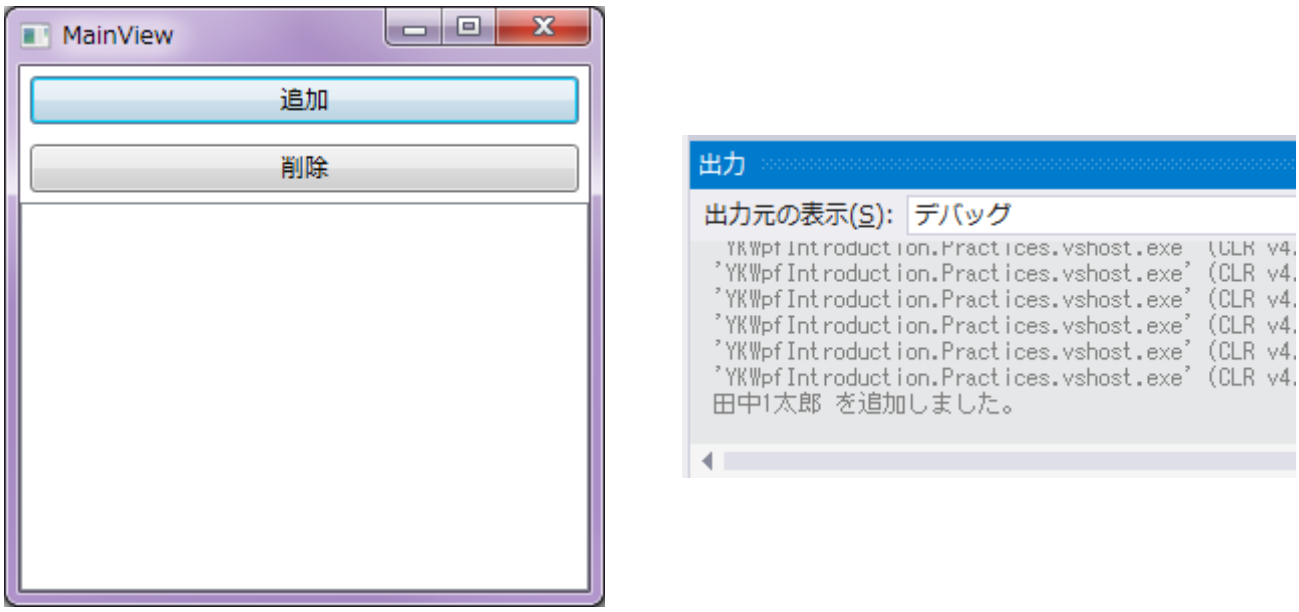
このサンプルコードの動作を確認するために次のような MainView ウィンドウを用意します。「追加」ボタンや「削除」ボタンで ListBox コントロールに表示するコレクションを追加/削除するようにしています。

コード 6.10：コレクションの要素を追加/削除するボタンを持つ UI

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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="追加" Command="{Binding AddCommand}" Margin="5" />
7          <Button DockPanel.Dock="Top" Content="削除" Command="{Binding DeleteCommand}" Margin="5" />
8          <ListBox ItemsSource="{Binding People}">
9              <ListBox.ItemTemplate>
10                 <DataTemplate>
11                     <StackPanel>
12                         <TextBlock Text="{Binding FullName}" />
13                         <TextBlock Text="{Binding Age, StringFormat=({0})}" />
14                     </StackPanel>
15                 </DataTemplate>
16             </ListBox.ItemTemplate>
17         </ListBox>
18     </DockPanel>
19 </Window>
```

このアプリケーションを実行してみましょう。すると、のように「追加」ボタンを押しても ListBox コントロールに追加された要素が表示されません。しかし、出力ウィンドウを確認するとメッセージが表示されているため、MainViewModel クラスの People コレクションには追加されているようです。なぜ ListBox コントロールに変更が反映されないかというと、それは変更通知をおこなっていないからです。



(a) 「追加」ボタンを押しても表示されない　　(b) AddCommand は正常に処理されている

図 6.6：「追加」ボタンを押しても ListBox コントロールに反映されない

AddCommand プロパティによってコマンドが実行されたとき、People.Add() メソッドによって People コレクションに子要素が 1 つ追加されます。このとき、People プロパティ自体に変化があったわけではないため People プロパティの set アクセサは実行されていません。つまり People プロパティの子要素が変更されたということを UI 側が認識できていないことになります。

では People.Add() メソッドを呼び出した後、"RaisePropertyChanged("People");" として強制的に People プロパティの変更通知を呼べばいいかというと、そうではありません。People プロパティのインスタンス自体には何の変化もなく、あくまでも People プロパティの子要素が変化した、ということを通知しなければいけません。このように子要素が変化したということを通知するには INotifyCollectionChanged インターフェースによる CollectionChanged イベントを発行する必要があります。そして、これを実装しているコレクションとして ObservableCollection<T> クラスというものが用意されています。したがって、UI にデータバインディングするコレクションデータは、特に理由がない限り ObservableCollection<T> クラスとするべきです。

MainViewModel クラスの People プロパティの型を List<Person> クラスから ObservableCollection<Person> クラスに変更しましょう。List<T> クラスと ObservableCollection<T> クラスは名前空間が異なるので、冒頭の using ディレクティブに変更があることに注意しましょう。

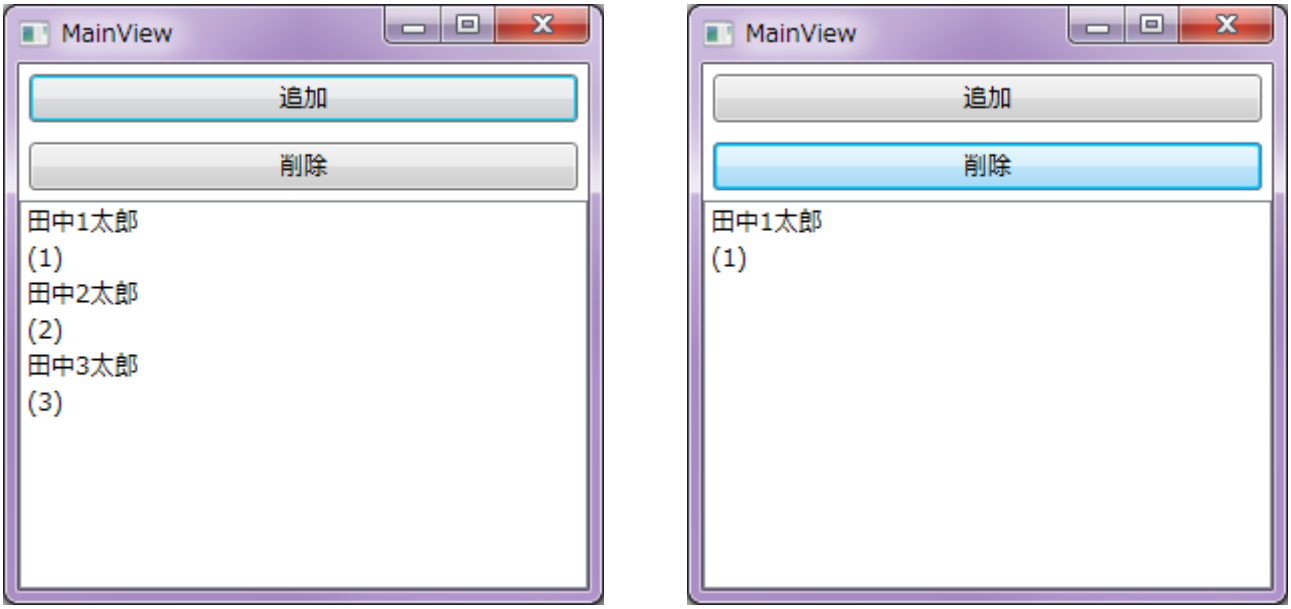
コード 6.11：ObservableCollection<T> クラスを用いたコレクションデータ

MainViewModel.cs

```
1  namespace YKWpfIntroduction.Practices.ViewModels
2  {
3      using System.Collections.ObjectModel;
4      using YKWpfIntroduction.Practices.Models;
5
6      /// <summary>
7      /// MainView ウィンドウに対するデータコンテキストを表します。
8      /// </summary>
9      internal class MainViewModel : NotificationObject
10     {
11         private ObservableCollection<Person> _people = new ObservableCollection<Person>();
12         /// <summary>
13         /// 人物情報のコレクションを取得します。
14         /// </summary>
15         public ObservableCollection<Person> People
16         {
17             get { return this._people; }
18             private set { SetProperty(ref this._people, value); }
19         }
```

```
20
21          private int _count;
22
23          private DelegateCommand _addCommand;
24          /// <summary>
25          /// 追加コマンドを取得します。
26          /// </summary>
27          public DelegateCommand AddCommand...
48
49          private DelegateCommand _deleteCommand;
50          /// <summary>
51          /// 削除コマンドを取得します。
52          /// </summary>
53          public DelegateCommand DeleteCommand...
66      }
67  }
```

これで People コレクションの子要素が追加されたり削除されたりしたときに CollectionChanged イベントが発行されるため、UI 側が認識できるようになりました。実行してみると、ListBox コントロールに表示されるようになります。



(a) 「追加」ボタンを押すと追加される　　(b) 「削除」ボタンを押すと削除される

図 6.7：子要素の数が変更されたことを ListBox コントロールが認識している

6.6　親要素のデータコンテキストとデータバインディングする

ItemsControl コントロールで並べられる各アイテムは、そのデータコンテキストが親要素のデータコンテキストではなく、コレクション要素のクラスとなります。このことから色々と不都合なことが起こります。例えば、前節ではコレクション要素を削除するために「削除」ボタンを用意しましたが、ListBox コントロールの外側に配置されています。これを ListBox コントロールの各アイテムに表示するようにしてみましょう。すると、DeleteCommand の実装をどうすれば良いか悩むと思います。

コード 6.12：各アイテムが「削除」ボタンを持つ場合

MainView.xaml

```
 1 <Window x:Class="YKWpfIntroduction.Practices.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="追加" Command="{Binding AddCommand}" Margin="5" />
 7         <ListBox ItemsSource="{Binding People}">
 8             <ListBox.ItemTemplate>
 9                 <DataTemplate>
10                     <StackPanel Orientation="Horizontal">
11                         <Button Content="X" Command="{Binding DeleteCommand}" Margin="5" />
12                         <StackPanel>
13                             <TextBlock Text="{Binding FullName}" />
14                             <TextBlock Text="{Binding Age, StringFormat=({0})}" />
15                         </StackPanel>
16                     </StackPanel>
17                 </DataTemplate>
18             </ListBox.ItemTemplate>
19         </ListBox>
20     </DockPanel>
21 </Window>
```

11 行目のように記述した場合、各アイテムのデータコンテキストが DeleteCommand プロパティを持たなければなりません。しかし、各アイテムのデータコンテキストは今の場合は Person クラスで、このクラスが自分自身を削除するコマンドを持つというのはシステムとして非常に複雑になってしまいます。

そこで、11 行目を次のように書き換えます。

コード 6.13：RelativeSource プロパティによる参照先の変更

MainView.xaml

```
11                         <Button Content="X" Command="{Binding DataContext.DeleteCommand,
    RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type ListBox}}}"
    CommandParameter="{Binding .}" Margin="5" />
```

Command プロパティに対する Binding クラスの中身が少し長くなっています。Binding クラスの RelativeSource プロパティを設定することで、データバインディングで参照するデータコンテキストを相対的に指定することができるようになります。ここでは RelativeSource に "FindAncestor" というキーワードを指定しています。"Ancestor" というのは直訳すると "先祖" という意味ですが、ここでは "親要素を辿る" ということを意味します。さらに "AncestorType" として親要素の型に ListBox クラスを指定しています。つまり、"親要素である ListBox コントロールをデータバインディングの参照先とする" ように RelativeSource プロパティを設定しています。そして、"DeleteCommand" と記述していた部分は "DataContext.DeleteCommand" というようにしています。これは、RelativeSource プロパティによってデータバインディングで参照するクラスが ListBox クラスとなったため、そこから MainViewModel クラスを辿るには ListBox クラスが持つ DataContext プロパティを参照する必要があるからです。

さらに、ここでは CommandParameter プロパティに対して自分自身をデータバインディングで与えています。"自分自身" とは、つまり各アイテムのデータコンテキストである Person クラスです。各アイテムが削除ボタンを持つということは、そのボタンを押すと該当するアイテムが削除されるようにしなければいけません。そのため、DeleteCommand プロパティに対して削除対象となるアイテムが誰なのかを明示的に与える必要があります。これを実現するために、CommandParameter プロパティを介して削除対象である自分自身を参照させています。

このように変更した上で、MainViewModel クラスでは、DeleteCommand プロパティを次のように変更します。

コード 6.14：CommandParameter を受け取る

MainViewModel.cs

```
49          private DelegateCommand _deleteCommand;
50          /// <summary>
51          /// 削除コマンドを取得します。
52          /// </summary>
53          public DelegateCommand DeleteCommand
54          {
55              get
56              {
57                  return this._deleteCommand ?? (this._deleteCommand = new DelegateCommand(
58                  p =>
59                  {
60                      this.People.Remove(p as Person);
61                      this.DeleteCommand.RaiseCanExecuteChanged();
62                  }));
63              }
64          }
```

XAML で与えられた CommandParameter プロパティは、コマンド実行処理の入力引数として与えられるようになります。したがって、58 行目のように今まではアンダースコア "_" で未使用変数として表現していたパラメータを "p" に変更し、60 行目でこれを Person クラスにキャストして使用しています。また、各アイテムが削除ボタンを持つようになったため、実行可能判別処理を省略しています。

6.7 まとめ

ItemsControl コントロールのカスタマイズ方法をマスターすれば、ListBox コントロールや ComboBox コントロールなども同様にカスタマイズできるようになります。この方法をマスターすることで、例えば子要素に CheckBox コントロールを持つ ListBox コントロールや、画像や動画を含む ComboBox コントロールなども簡単に作ることができるようになります。

各子要素のデータコンテキストが親要素のデータコンテキストと異なることに注意が必要です。コレクションの各要素がデータバインディングする相手も INotifyPropertyChanged インターフェースを実装する必要がある場合もあるため、NotificationObject クラスを基本クラスとして適切にプロパティ変更通知をおこないましょう。

コレクションの子要素を動的に追加/削除する場合は ObservableCollection<T> クラスを使用するようにしましょう。コレクションの子要素の変化は INotifyCollectionChanged インターフェースによる CollectionChanged イベントを発行させる必要があり、ObservableCollection<T> クラスは INotifyCollectionChanged インターフェースを実装している標準クラスです。特に理由がない限りは ObservableCollection<T> クラスによるコレクションを公開するようにしましょう。

7 スタイルとトリガの基礎

WPF ではコントロールに対して Style を定義することができます。Style によってその外観を共通化したり、動的な変化を与えるトリガを設定したりできるようになります。本章ではスタイルとトリガについて基本的な使い方を紹介します。新たな WPF プロジェクトを作成してから進めましょう。

7.1 スタイルを指定する方法

まず、ウィンドウの真ん中に Button コントロールを配置しましょう。やり方はいろいろありますが、ここでは次のように配置します。

コード 7.1：Button コントロールを中心に配置する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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      <Grid>
6          <Border HorizontalAlignment="Center" VerticalAlignment="Center">
7              <Button />
8          </Border>
9      </Grid>
10 </Window>
```

Content プロパティなどを何も指定していないため、非常に小さなボタンが表示されます。

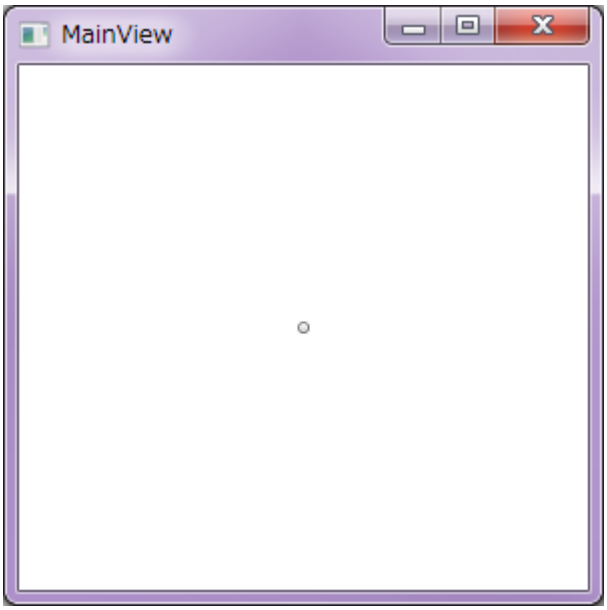


図 7.1：真ん中にボタンが配置されている

この Button コントロールに対して Style を定義していきましょう。例えば次のように Style から Content プロパティを設定します。

コード 7.2：Button コントロールの Style を定義する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.Views.MainView"
2          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>
        <Border HorizontalAlignment="Center" VerticalAlignment="Center">
            <Button>
                <Button.Style>
                    <Style TargetType="{x:Type Button}">
                        <Setter Property="Content" Value="Click me." />
                    </Style>
                </Button.Style>
            </Button>
        </Border>
    </Grid>
</Window>
```

Style クラスはそのスタイルがどのコントロールに対するものなのかを指定する TargetType プロパティがあるので、これに対して Button クラスの型情報を指定します。そして、Style クラスの中に Setter クラスを羅列していくことでそのスタイルを定義していきます。上記では Content プロパティに "Click me." という文字列を設定しています。実行すると次のようにコンテンツが設定されていることが確認できます。

図 7.2：Style を通して Content プロパティが設定されている

他にも Setter を並べることで色々な設定をおこなうことができます。

コード 7.3：Style に Setter を並べることで設定を追加する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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>
        <Border HorizontalAlignment="Center" VerticalAlignment="Center">
            <Button>
                <Button.Style>
                    <Style TargetType="{x:Type Button}">
                        <Setter Property="Content" Value="Click me." />
                        <Setter Property="Padding" Value="10,5" />
                        <Setter Property="FontFamily" Value="Consolas" />
                        <Setter Property="FontSize" Value="24" />
                    </Style>
                </Button.Style>
```

```
16              </Button>
17          </Border>
18      </Grid>
19  </Window>
```

図 7.3：FontFamily などを設定された Button コントロール

また、Setter の Value に入れ子のクラスを指定したいときは、Setter のタグの中に Value を書かずに次のように記述します。

コード 7.4：Setter.Value プロパティを入れ子で指定する

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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      <Grid>
6          <Border HorizontalAlignment="Center" VerticalAlignment="Center">
7              <Button>
8                  <Button.Style>
9                      <Style TargetType="{x:Type Button}">
10                         <Setter Property="Content" Value="Click me." />
11                         <Setter Property="Padding" Value="10,5" />
12                         <Setter Property="FontFamily" Value="Consolas" />
13                         <Setter Property="FontSize" Value="24" />
14                         <Setter Property="Background">
15                             <Setter.Value>
16                                 <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
17                                     <GradientStop Color="LightGray" Offset="0" />
18                                     <GradientStop Color="Orange" Offset="1" />
19                                 </LinearGradientBrush>
20                             </Setter.Value>
21                         </Setter>
22                     </Style>
23                 </Button.Style>
24             </Button>
25         </Border>
26     </Grid>
27 </Window>
```

LinearGradientBrush はグラデーションで塗りつぶすためのブラシです。実行結果は次のようになります。

図 7.4：グラデーションで塗りつぶされた Button コントロール

このように Style を通してプロパティを設定することができます。

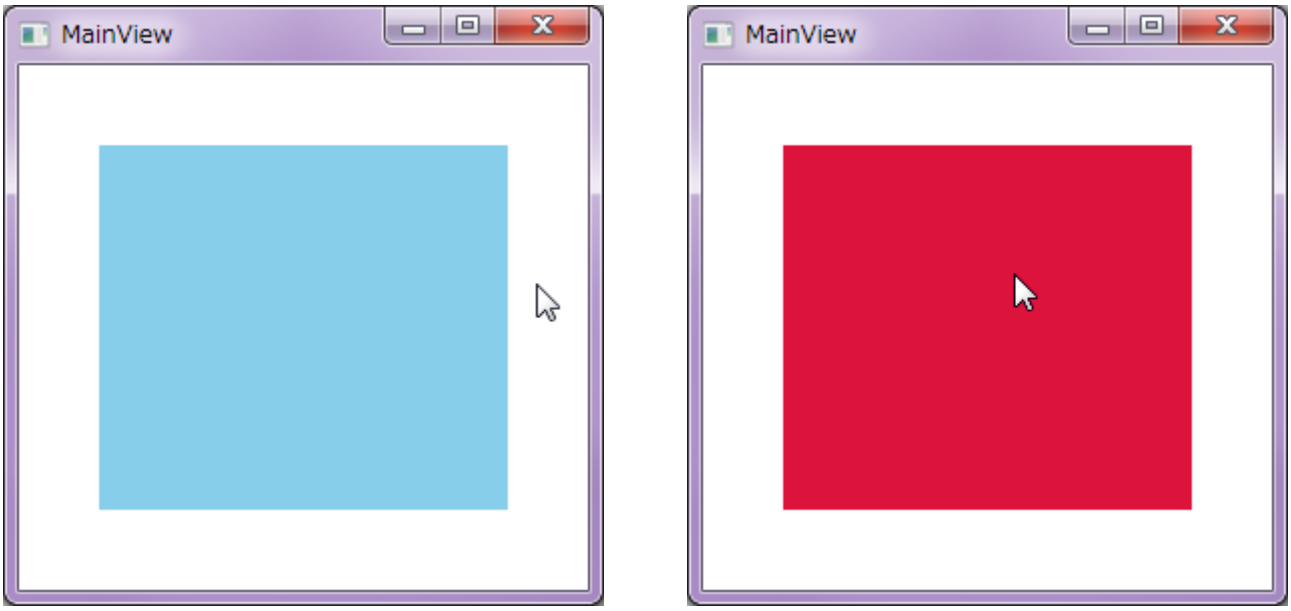
7.2 トリガを指定する方法

Style ではプロパティをただ設定するだけでなく、あるトリガによってその設定を動的に変更することができます。例えば次のようなコードでは、マウスがそのコントロールに乗っているかどうかによって背景色が変化します。

コード 7.5：Setter.Value プロパティを入れ子で指定する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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>
        <Border Margin="40">
            <Border.Style>
                <Style TargetType="{x:Type Border}">
                    <Setter Property="Background" Value="SkyBlue" />
                    <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value="True">
                            <Setter Property="Background" Value="Crimson" />
                        </Trigger>
                    </Style.Triggers>
                </Style>
            </Border.Style>
        </Border>
    </Grid>
</Window>
```



(a) マウスが乗っていない状態　　(b) マウスが乗っている状態

図 7.5：IsMouseOver プロパティの変化にしたがって Background プロパティが変化する

トリガを設定するときは、Style.Triggers プロパティにトリガを羅列します。トリガにはいくつか種類があり、Trigger クラスは自身のプロパティの値をトリガとするためのクラスです。この他に外部のデータをトリガとするための DataTrigger クラス、ルートイベントをトリガとするための EventTrigger クラスなどがあります。

7.3 Style を共有する

例えば次のようなコードがあったとします。

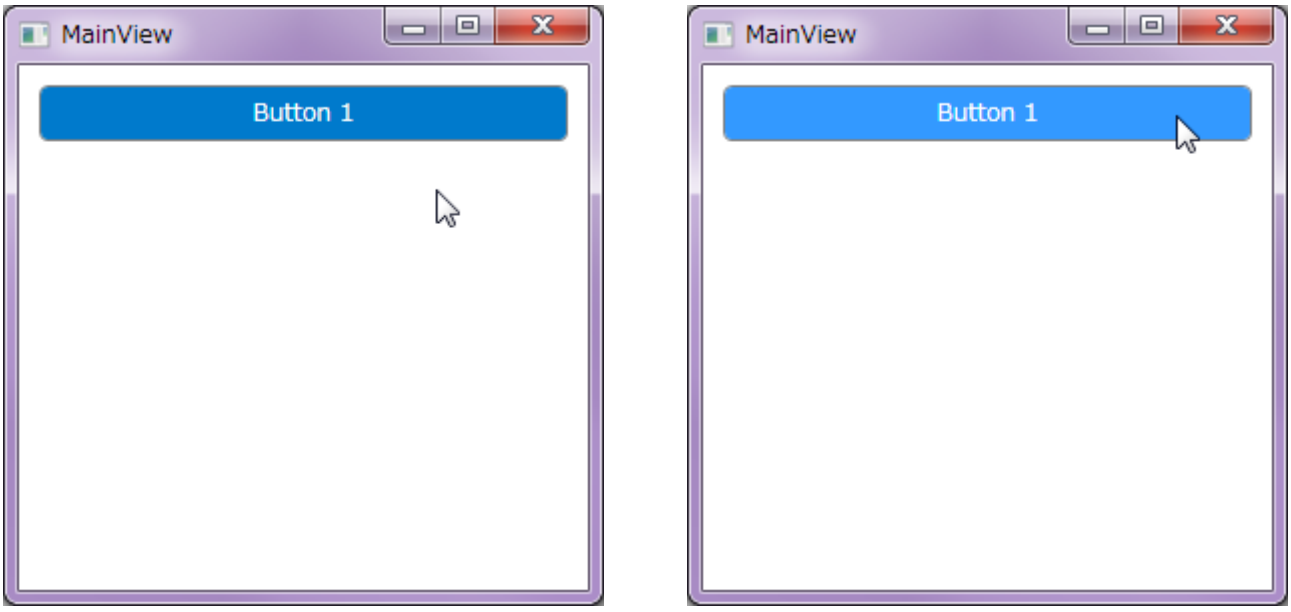
コード 7.6：Style で Button コントロールのテンプレートを指定する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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">
    <StackPanel Margin="10">
        <Button Content="Button 1">
            <Button.Style>
                <Style TargetType="{x:Type Button}">
                    <Setter Property="Foreground" Value="White" />
                    <Setter Property="Background">
                        <Setter.Value>
                            <SolidColorBrush Color="#FF007ACC" />
                        </Setter.Value>
                    </Setter>
                    <Setter Property="Margin" Value="4" />
                    <Setter Property="Padding" Value="4" />
                    <Setter Property="Template">
                        <Setter.Value>
                            <ControlTemplate TargetType="{x:Type Button}">
                                <Border Background="{TemplateBinding Background}"
                                        Padding="{TemplateBinding Padding}"
                                        BorderBrush="Gray"
                                        BorderThickness="1"
                                        CornerRadius="4">
                                    <ContentPresenter HorizontalAlignment="{TemplateBinding
HorizontalContentAlignment}"
                                                      VerticalAlignment="{TemplateBinding
VerticalContentAlignment}" />
                                </Border>
                            </ControlTemplate>
                        </Setter.Value>
                    </Setter>
                    <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value="True">
                            <Setter Property="Background">
                                <Setter.Value>
                                    <SolidColorBrush Color="#FF3399FF" />
                                </Setter.Value>
                            </Setter>
                        </Trigger>
                        <Trigger Property="IsPressed" Value="True">
                            <Setter Property="Background">
                                <Setter.Value>
                                    <SolidColorBrush Color="#FF007ACC" />
                                </Setter.Value>
                            </Setter>
                        </Trigger>
                    </Style.Triggers>
                </Style>
            </Button.Style>
        </Button>
    </StackPanel>
</Window>
```

Template プロパティを指定しているため、Buton コントロールの外観は Template プロパティに指定されている

ControlTemplate コントロールの内容にしたがうことになります。この実行結果は次のようになります。



(a) マウスが乗っていない状態　　(b) マウスが乗っている状態

図 7.6：背景色が動的に変化する新たな Button コントロール

10 行目で Background プロパティがやや暗めの青色に設定されており、ControlTemplate クラスの中では 20 行目の Border コントロールでその Background プロパティが反映されるように TemplateBinding によって指定されています。これがデフォルトの背景色となりますが、32 行目でマウスが乗っている状態のときにトリガを設定し、そのときの背景色を 32 行目でやや明るめの青色に変更するように設定しています。また、39 行目ではボタンが押されている状態のときにやや暗めの青色に変更することで、ボタンを押したときにボタンとして動作していることをユーザーがわかるようにしています。

このように Template プロパティを使用することで、標準コントロールである Button コントロールの外観を独自のコントロールとして簡単に書き換えることができます。

しかし、このままでは少し不便です。というのは、次のコードのように新たに Button コントロールを追加すると、そちらには Style が指定されていないので標準の外観になってしまうからです。

コード 7.7：別の Button コントロールを追加

MainView.xaml

```
 1  <Window x:Class="YKWpfIntroduction.Practices.Views.MainView"
 2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4          Title="MainView" Height="300" Width="300">
 5      <StackPanel Margin="10">
 6          <Button Content="Button 1">
 7              <Button.Style...>
49          </Button>
50
51          <Button Content="Button 2" />
52      </StackPanel>
53  </Window>
```

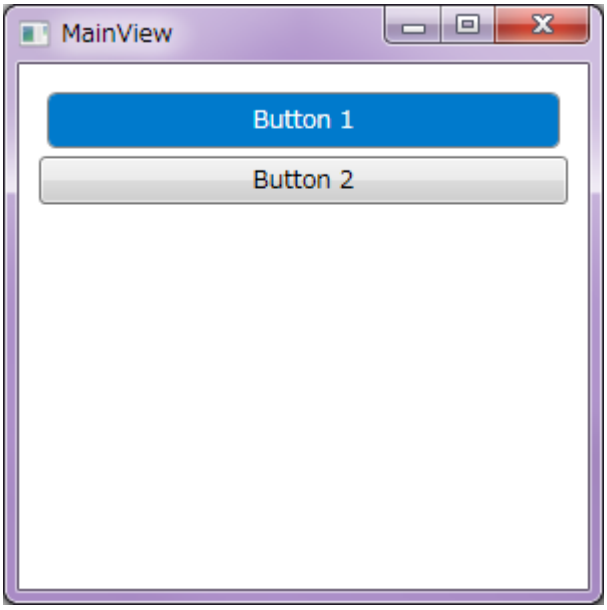


図 7.7：別の Button コントロールには反映されない

Style を共有する場合は、Style をリソースとして定義し、利用する側はリソースから参照するようにする必要があります。コード 7.7 を、リソースを用いて書き換えると次のようになります。

コード 7.8：StaticResource として Style を参照する

MainView.xaml

```
<Window x:Class="YKWpfIntroduction.Practices.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">
    <Window.Resources>
        <Style x:Key="ButtonStyle" TargetType="{x:Type Button}">
            <Setter Property="Foreground" Value="White" />
            <Setter Property="Background">
                <Setter.Value>
                    <SolidColorBrush Color="#FF007ACC" />
                </Setter.Value>
            </Setter>
            <Setter Property="Margin" Value="4" />
            <Setter Property="Padding" Value="4" />
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="{x:Type Button}">
                        <Border Background="{TemplateBinding Background}"
                                Padding="{TemplateBinding Padding}"
                                BorderBrush="Gray"
                                BorderThickness="1"
                                CornerRadius="4">
                            <ContentPresenter HorizontalAlignment="{TemplateBinding
HorizontalContentAlignment}"
                                              VerticalAlignment="{TemplateBinding
VerticalContentAlignment}" />
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background">
                        <Setter.Value>
```

```
33                          <SolidColorBrush Color="#FF3399FF" />
34                      </Setter.Value>
35                  </Setter>
36              </Trigger>
37              <Trigger Property="IsPressed" Value="True">
38                  <Setter Property="Background">
39                      <Setter.Value>
40                          <SolidColorBrush Color="#FF007ACC" />
41                      </Setter.Value>
42                  </Setter>
43              </Trigger>
44          </Style.Triggers>
45      </Style>
46  </Window.Resources>
47
48  <StackPanel Margin="10">
49      <Button Content="Button 1" Style="{StaticResource ButtonStyle}" />
50      <Button Content="Button 2" Style="{StaticResource ButtonStyle}" />
51  </StackPanel>
52 </Window>
```

Window クラスの Resources プロパティにリソースとして先ほどの Style を定義し直しています。このとき、6 行目のように "x:Key" としてそのリソースに名前を付けています。

このリソースを使用するときは、50、51 行目のように "StaticResorce" として参照するようにします。

図 7.8：同一の Style を共有している

また、"x:Key" によるリソースキーを指定しなかった場合、その Style はターゲットとするクラスの基本スタイルとして定義されます。つまり、50、51 行目で Style プロパティを指定しなくても自動的に適用されるようになります。

コード 7.9：リソースキーを指定しない場合はデフォルトスタイルとなる

MainView.xaml

```
1  <Window x:Class="YKWpfIntroduction.Practices.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      <Window.Resources>
6          <Style TargetType="{x:Type Button}">
7              <Setter Property="Foreground" Value="White" />
8              <Setter Property="Background">
9                  <Setter.Value>
```

```
10                  <SolidColorBrush Color="#FF007ACC" />
11              </Setter.Value>
12          </Setter>
13          <Setter Property="Margin" Value="4" />
14          <Setter Property="Padding" Value="4" />
15          <Setter Property="Template">
16              <Setter.Value>
17                  <ControlTemplate TargetType="{x:Type Button}">
18                      <Border Background="{TemplateBinding Background}"
19                              Padding="{TemplateBinding Padding}"
20                              BorderBrush="Gray"
21                              BorderThickness="1"
22                              CornerRadius="4">
23                          <ContentPresenter HorizontalAlignment="{TemplateBinding
    HorizontalContentAlignment}"
24                                            VerticalAlignment="{TemplateBinding
    VerticalContentAlignment}" />
25                      </Border>
26                  </ControlTemplate>
27              </Setter.Value>
28          </Setter>
29          <Style.Triggers>
30              <Trigger Property="IsMouseOver" Value="True">
31                  <Setter Property="Background">
32                      <Setter.Value>
33                          <SolidColorBrush Color="#FF3399FF" />
34                      </Setter.Value>
35                  </Setter>
36              </Trigger>
37              <Trigger Property="IsPressed" Value="True">
38                  <Setter Property="Background">
39                      <Setter.Value>
40                          <SolidColorBrush Color="#FF007ACC" />
41                      </Setter.Value>
42                  </Setter>
43              </Trigger>
44          </Style.Triggers>
45      </Style>
46  </Window.Resources>
47
48  <StackPanel Margin="10">
49      <Button Content="Button 1" />
50      <Button Content="Button 2" />
51  </StackPanel>
52 </Window>
```

ワンポイント 7.1：リソースは親要素を検索する

ここでは Window クラスの Resources プロパティにリソースを定義しましたが、このリソースを使用する要素から辿れる親要素であればどのクラスの Resources プロパティでも構いません。例えばコード 7.9 の場合、48 行目の StackPanel クラスの Resources プロパティに定義しても同じ結果が得られます。定義するリソースの使用範囲を限定したい場合などで使い分けることができます。また、複数ウィンドウでリソースを共有する場合は Application クラスの Resources プロパティに指定する必要があります。Application.Resources プロパティは App.xaml で指定できます。

ワンポイント 7.2：リソースには Style 以外も定義できる

ここでは Style クラスをリソースに定義しましたが、リソースには XAML で定義できるクラスであれば何でも構いません。例えばコード 7.9 では 10 行目と 40 行目で同じ設定の SolidColorBrush を使用しているので 1 つ分のメモリを余計に消費しています。SolidColorBrush クラスをリソースとして定義し、これを共有するようにすることでメモリの節約にもなります。この他にも DataTemplate クラスや ControlTemplate クラスもリソースとして定義することができます。

8 おわりに

C# によるコンソールアプリケーション作成から始まり、WPF による UI 構築や、MVVM パターンを意識したアプリケーション開発に関する基礎を紹介しました。

WPF によるアプリケーション開発では、C# によるコーディングよりもむしろ XAML による表現方法をどのようにすればよいのかについて悩む場面が多くなると思います。WPF が難しく感じる理由の一つとしてこの XAML があると思います。XAML 力を高くすることで一つの壁を超えることができると思いますので、繰り返し練習を重ねてください。本書では説明していませんが、独自のコントロールを作成できるユーザーコントロールやカスタムコントロールなどを作成することでより XAML に対する理解が深まると思いますので、興味のある方はそちらのほうも是非チャレンジしてみてください。

MFC ではある機能を実装するとき、どういうコードを書けばいいのだろう、ということによく悩んでいましたが、WPF+C# ではそういうことよりもむしろ、どんなコンテンツで表現すればいいのだろう、ということに悩むことが多いと思います。これは WPF+C# 引いては .NET Framework が小難しい処理をすべてラップしてくれているおかげで、開発者が本来悩むべき問題、つまり「このアプリをどうやって作るのか」ではなく、「このアプリはどうやって使うのか（使われるべきか）」という問題に正面から取り組めるようになったのだと思います。開発者は常にユーザーの視点からアプリの設計をすべきで、ユーザーエクスペリエンスの高いアプリケーションを開発するために様々な機能を盛り込まなければいけません。そのためのツールとして WPF+C# が選択肢の一つとして挙げられると思いますので、本書を読破した方はより応用的な技術も修得して、今後の開発人生の糧にしていただきたいと思います。

改訂履歴

改訂年月日	バージョン	概　要
2016 年 8 月 25 日	Ver.1.0.0	初版リリース。

Windows Presentation Foundation 実践

2016 年 8 月 25 日 初版