Chapter8_リクエストにお答えします

リクエストがあったサンプルコードを公開する。

8-1 インターフェース

インターフェースを実装する場合、メソッドにoverrideキーワードを付ける必要がない。

抽象クラスの抽象メソッドを実装する場合、メソッドにoverrideキーワードを付ける必要がある。

EXAMPLE 例8-1 インターフェースの例

```
using System;
using System.Collections.Generic;

namespace Example2
{
	class Program : ASample, ISample
	{
		public string Name { get; set; }

		public Program(string name)
		{
			Name = name;
		}

		static void Main(string[] args)
		{
			IList<ISample> list = new List<ISample>();
			list.Add(new Program("インスタンス1"));
			list.Add(new Program("インスタンス2"));
			list.Add(new Program("インスタンス3"));

			foreach(ISample sample in list)
			{
				sample.Run();
				sample.Walk();
			}

			ASample aSample = new Program("抽象クラス実装");
			aSample.Talk();

		}

		// インターフェースの抽象メソッドを実装する際はoverride不要
		public void Run()
		{
			Console.WriteLine(Name + ":Runメソッドを実行した");
		}

		// インターフェースの抽象メソッドを実装する際はoverride不要
		public void Walk()
		{
			Console.WriteLine(Name + ":Walkメソッドを実行した");
		}

		// 抽象クラスの抽象メソッドを実装する際はoverride必要
		public override void Talk()
		{
			Console.WriteLine(Name + ":Talkメソッドを実行した");
		}
	}

	// インターフェース
	interface ISample
	{
		public abstract void Walk();

		public abstract void Run();
	}

	// 抽象クラス
	abstract class ASample
	{
		public abstract void Talk();
	}
}
```

RESULTS 例8-1の実行結果

```
インスタンス1:Runメソッドを実行した
インスタンス1:Walkメソッドを実行した
インスタンス2:Runメソッドを実行した
インスタンス2:Walkメソッドを実行した
インスタンス3:Runメソッドを実行した
インスタンス3:Walkメソッドを実行した
抽象クラス実装:Talkメソッドを実行した
```

8-2 複数のフォームを切り替える

プロジェクトからフォームを追加してもコードを書かなければフォームの切り替えはできない。

ここでは3つのフォームを用意し、切り替える方法を紹介する。

筆者はプロジェクト名を「MultiFormApp」で作成した。任意のプロジェクト名で良い。

Step01 フォームを用意する

まずフォームを追加する。1つは既にあるはずなので、2つ追加すれば良い。

筆者の環境ではフォーム名をデフォルトの「Form2.cs」、「Form3.cs」とした。

フォームの追加方法

ソリューションエクスプローラーのプロジェクトを右クリック > 追加 > 新しい項目 > 左側のメニューから「Windows Forms」を選択 > 「フォーム(Windowsフォーム)」を選択 > 適当なフォーム名を入力 > 追加

Step02 フォームをデザインする

コントロールを配置し、デザインを行う。

◆Form1

図8-1 Form1のデザイン

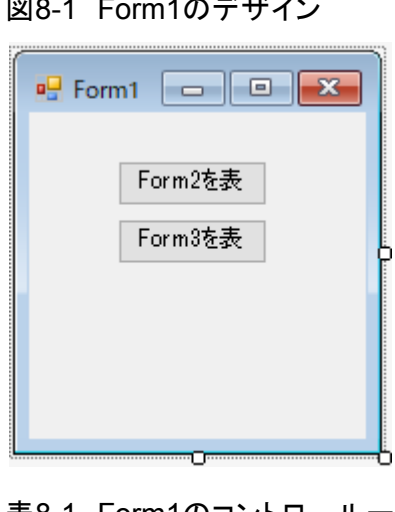


表8-1 Form1のコントロール一覧

コントロール	Name	Text	説明
Form	Form1	Form1	プロジェクト作成時に自動的に生成されるフォーム。
Button	button1	Form2を表示	クリックするとForm2を表示するためのボタン。
Button	button2	Form3を表示	クリックするとForm3を表示するためのボタン。

◆Form2

図8-2 Form2のデザイン

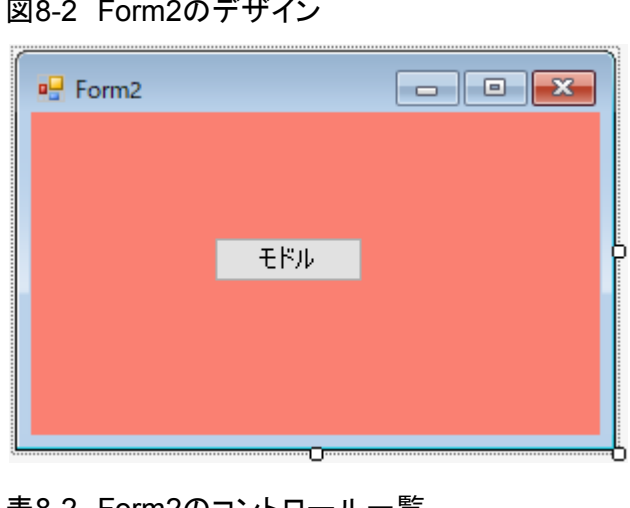


表8-2 Form2のコントロール一覧

コントロール	Name	Text	説明
Form	Form2	Form2	フォーム追加時に自動的に生成されるフォーム。 ※フォームを判別しやすいようにBackColorプロパティを変更済
Button	button1	モドル	クリックするとForm1を表示するためのボタン。

◆Form3

図8-3 Form3のデザイン

表8-2 Form2のコントロール一覧

コントロール	Name	Text	説明
Form	Form3	Form3	フォーム追加時に自動的に生成されるフォーム。 ※フォームを判別しやすいようにBackColorプロパティを変更済
Button	button1	モドル	クリックするとForm1を表示するためのボタン。

Step03 イベントハンドラを用意する

イベントハンドラを用意し、イベント(ボタンクリック)に対応する。

◆Form1

ボタン「button1」をダブルクリックし、イベントハンドラ「button1_Click」を用意。

ボタン「button2」をダブルクリックし、イベントハンドラ「button2_Click」を用意。

◆Form2

ボタン「button1」をダブルクリックし、イベントハンドラ「button1_Click」を用意。

Form2を選択後、プロパティウィンドウのイベントタブにある「FormClosed」の値に「OnFormClosed」と入力し、イベントハンドラを用意。

◆Form3

ボタン「button1」をダブルクリックし、イベントハンドラ「button1_Click」を用意。

Form2を選択後、プロパティウィンドウのイベントタブにある「FormClosed」の値に「OnFormClosed」と入力し、イベントハンドラを用意。

Step04 イベントハンドラの処理内容を書く ＋α

イベントハンドラの処理内容を書くのだが、＋αで書かなければいけない点がある。

◆Form1

以下の点を書く。

1. メンバ変数 ： 自身以外のフォーム
2. コンストラクタ ： Form2,Form3をインスタンス化してメンバ変数へ代入
3. イベントハンドラ「button1_Click」 ： Form1を非表示にする、Form2を表示する
4. イベントハンドラ「button2_Click」 ： Form1を非表示にする、Form3を表示する

コード8-1 Form1.cs

```
using System;
using System.Windows.Forms;

namespace MultiFormApp
{
	public partial class Form1 : Form
	{
		// 1. メンバ変数 ： 自身以外のフォーム
		private Form form2, form3;

		// 2. コンストラクタ ： Form2,Form3をインスタンス化してメンバ変数へ代入
		public Form1()
		{
			InitializeComponent();
			form2 = new Form2(this);
			form3 = new Form3(this);
		}

		// 3. イベントハンドラ「button1_Click」 ： Form1を非表示にする、Form2を表示する
		private void button1_Click(object sender, EventArgs e)
		{
			Hide();
			form2.Show();
		}

		// 4. イベントハンドラ「button2_Click」 ： Form1を非表示にする、Form3を表示する
		private void button2_Click(object sender, EventArgs e)
		{
			Hide();
			form3.Show();
		}
	}
}
```

解説 Form1のしくみ

1. メンバ変数 ： 自身以外のフォーム

メンバ変数にform2, form3を定義しているのは、フォームに配置したコントロール以外は(プロパティNameを利用した)変数として扱えないため。最初から存在しているフォームのForm1だけは例外で、変数として扱える。

2. コンストラクタ ： **Form2,Form3をインスタンス化してメンバ変数へ代入**

プロジェクトから追加したクラスやフォームはインスタンス生成することで使うことができる。クラスForm1はプログラム実行時に自動でインスタンス生成される(Visual Studio自体がそのようにコードを自動生成する)ため、わざわざ自分でインスタンス生成する必要はない。

Form2およびForm3のインスタンス生成時、引数にForm1のインスタンスを渡している。これは、Form2およびForm3は初期状態ではForm1のインスタンスを持っていないため、引数として渡すことでForm1のインスタンスを使えるようにする試みである。

※クラスForm2とクラスForm3に引数ありコンストラクタを定義していないとエラーが出る。

3. イベントハンドラ「**button1_Click**」 ： **Form1を非表示にする、Form2を表示する**

フォームを非表示にするにはHideメソッドを使えば良い。フォームを表示するにはShowメソッドを使う。自分自身のフォームを非表示にしてから、別フォームを表示する制御にしている。別フォームを表示してから自身のフォームを非表示にする場合だと一瞬ではあるが、フォームが2つ表示されることになってしまう。

4. イベントハンドラ「**button2_Click**」 ： **Form1を非表示にする、Form3を表示する**

(3. と同じ内容なので説明を省略する)

◆Form2

以下の点を書く。

1. メンバ変数 ： Form1のフォーム
2. コンストラクタ ： 引数をメンバ変数へ代入、Form2を非表示にする
3. イベントハンドラ「button1_Click」 ： Form2を非表示にする、Form1を表示する
4. イベントハンドラ「OnFormClosed」 ： Form1を閉じる

コード8-2 Form2.cs

```
using System;
using System.Windows.Forms;

namespace MultiFormApp
{
	public partial class Form2 : Form
	{
		// 1. メンバ変数 ： Form1のフォーム
		private Form menuForm;

		// 2. コンストラクタ ： 引数をメンバ変数へ代入、Form2を非表示にする
		public Form2(Form form)
		{
			InitializeComponent();
			menuForm = form;
			Hide();
		}

		// 3. イベントハンドラ「button1_Click」 ： Form2を非表示にする、Form1を表示する
		private void button1_Click(object sender, EventArgs e)
		{
			Hide();
			menuForm.Show();
		}

		// 4. イベントハンドラ「OnFormClosed」 ： Form1を閉じる
		private void OnFormClosed(object sender, FormClosedEventArgs e)
		{
			menuForm.Close();
		}
	}
}
```

解説 Form2のしくみ

1. メンバ変数 ： **Form1のフォーム**

Form1のインスタンス情報を持っておかないと、Form1を表示することができない。

2. コンストラクタ ： 引数をメンバ変数へ代入、**Form2を非表示にする**

Form1のインスタンス情報はコンストラクタの引数で受け取る。値を保持するためにメンバ変数menuFormに代入した後、フォーム(Form2)を非表示にしておく。初期状態でフォームが表示されていると、Form1とForm2が同時に表示されることになってしまう。なお、コンストラクタは初期状態では引数なしのコンストラクタのみがコードに存在する。コンストラクタの引数は自由に変更しても問題無い。

※Form1だけは例外で、プログラム実行時にForm1の引数なしコンストラクタが呼び出されるため、必ず引数なしのコンストラクタにしておくこと。

3. イベントハンドラ「**button1_Click**」 ： **Form2を非表示にする、Form1を表示する**

自分自身(Form2)を非表示にしてからForm1を表示する。

4. イベントハンドラ「OnFormClosed」 ： Form1を閉じる

Form2を閉じた時のイベントハンドラ。Form1を閉じる。

◆Form3

以下の点を書く。

1. メンバ変数 ： Form1のフォーム
2. コンストラクタ ： 引数をメンバ変数へ代入、Form3を非表示にする
3. イベントハンドラ「button1_Click」 ： Form3を非表示にする、Form1を表示する
4. イベントハンドラ「OnFormClosed」 ： Form1を閉じる

コード8-3 Form3.cs

```
using System;
using System.Windows.Forms;

namespace MultiFormApp
{
	public partial class Form3 : Form
	{
		// 1. メンバ変数 ： Form1のフォーム
		private Form menuForm;

		// 2. コンストラクタ ： 引数をメンバ変数へ代入、Form3を非表示にする
		public Form3(Form form)
		{
			InitializeComponent();
			menuForm = form;
			Hide();
		}

		// 3. イベントハンドラ「button1_Click」 ： Form3を非表示にする、Form1を表示する
		private void button1_Click(object sender, EventArgs e)
		{
			Hide();
			menuForm.Show();
		}

		// 4. イベントハンドラ「OnFormClosed」 ： Form1を閉じる
		private void OnFormClosed(object sender, FormClosedEventArgs e)
		{
			menuForm.Close();
		}
	}
}
```

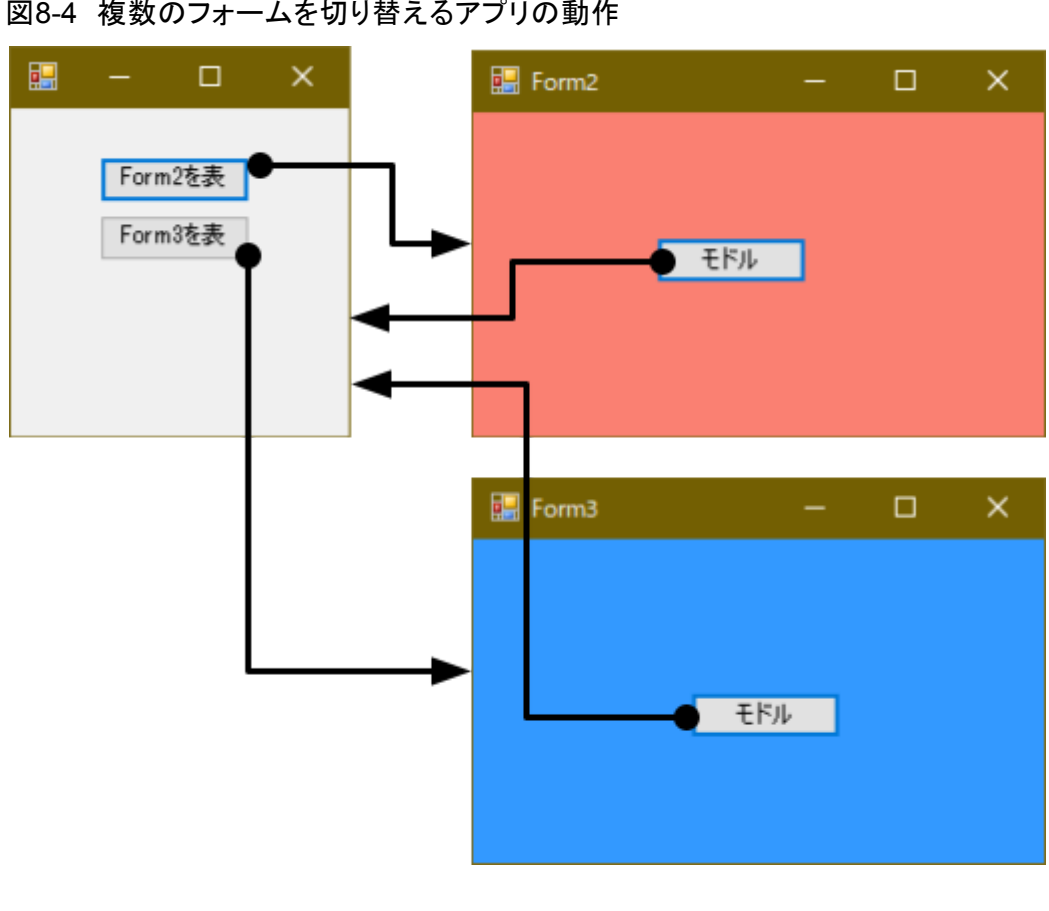
解説 Form3のしくみ

Form2のしくみと同様。

Step05 動作確認

動作確認を行う。次のイメージ通りに動作すれば成功。

図8-4 複数のフォームを切り替えるアプリの動作



8-3 画像表示を切り替える

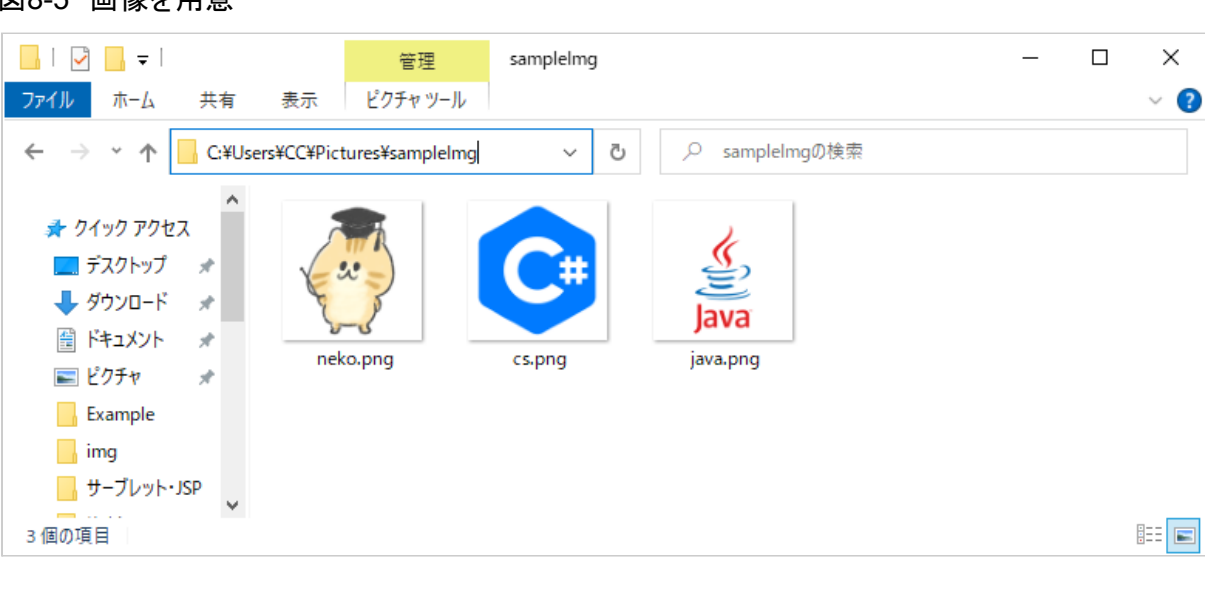
画像の表示を切り替える方法を紹介する。画像はImageBoxで表示し、併設されたButtonを押すとImageBoxの画像が切り替わる仕組みを作る。

筆者はプロジェクト名を「SampleApp」で作成した。任意のプロジェクト名で良い。

Step01 画像を用意する

適当な場所に画像を配置する。筆者は「C:\Users\CC\Pictures\sampleImg」に保存した。環境によって保存できる場所は異なるので、任意の場所で良い。

図8-5 画像を用意



Step02 フォームをデザインする

コントロールを配置し、デザインを行う。

◆Form1

図8-6 Form1のデザイン

表8-3 Form1のコントロール一覧

コントロール	Name	Text	説明
Form	Form1	SAMPLE	プロジェクト作成時に自動的に生成されるフォーム。
PictureBox	pictureBox1	*(None)*	画像を表示するためのコントロール。初期状態で画像を表示してもしなくてもいい。プロパティ「Image」で初期状態の画像を設定することが可能。プロパティ「SizeMode」で画像が収まるように値を設定すると良い。
Button	button1	画像1	クリックすると表示されている画像を用意した画像1に変更するためのボタン。
Button	button2	画像2	クリックすると表示されている画像を用意した画像2に変更するためのボタン。
Button	button3	ランダム	クリックすると表示されている画像を用意した画像からランダムに変更するためのボタン。

Step03 イベントハンドラを用意する

イベントハンドラを用意し、イベント(ボタンクリック)に対応する。

◆Form1

ボタン「button1」をダブルクリックし、イベントハンドラ「button1_Click」を用意。

ボタン「button2」をダブルクリックし、イベントハンドラ「button2_Click」を用意。

ボタン「button3」をダブルクリックし、イベントハンドラ「button3_Click」を用意。

Step04 イベントハンドラの処理内容を書く ＋α

イベントハンドラの処理内容を書くのだが、＋αで書かなければいけない点がある。

◆Form1

以下の点を書く。

- メンバ変数 ： 画像のパス情報、Randomクラスのインスタンス用変数(乱数発生用)

- コンストラクタ ： Randomクラスのインスタンス用変数にインスタンス代入
- イベントハンドラ「button1_Click」 ： 表示されている画像を用意した画像1に切り替える
- イベントハンドラ「button2_Click」 ： 表示されている画像を用意した画像2に切り替える
- イベントハンドラ「button3_Click」 ： 表示されている画像を用意した画像に切り替える（ランダム）

コード8-4 Form1.cs

```
using System;
using System.Windows.Forms;

namespace SampleApp
{
	public partial class Form1 : Form
	{
		// 1. 画像のパス情報
		// C:\Users\CC\Pictures\sampleImgに画像があることを前提としている。
		// 環境によって上記パスは変わる。
		private string[] imgPaths = {
			@"C:\Users\CC\Pictures\sampleImg\cs.png",
			@"C:\Users\CC\Pictures\sampleImg\neko.png",
			@"C:\Users\CC\Pictures\sampleImg\java.png"
		};
		// 1. Randomクラスのインスタンス用変数（乱数発生用）
		private Random random;

		public Form1()
		{
			InitializeComponent();
			// 2. Randomクラスのインスタンス用変数にインスタンス代入
			random = new Random();
		}

		// button1（画像1）をクリックしたときに発生するイベントハンドラ
		private void button1_Click(object sender, EventArgs e)
		{
			// 3. 表示されている画像を用意した画像1に切り替える
			pictureBox1.ImageLocation = imgPaths[0];
		}

		// button2（画像2）をクリックしたときに発生するイベントハンドラ
		private void button2_Click(object sender, EventArgs e)
		{
			// 4. 表示されている画像を用意した画像2に切り替える
			pictureBox1.ImageLocation = imgPaths[1];
		}

		// button3（ランダム）をクリックしたときに発生するイベントハンドラ
		private void button3_Click(object sender, EventArgs e)
		{
			// 5. 表示されている画像を用意した画像に切り替える（ランダム）
			pictureBox1.ImageLocation = imgPaths[random.Next(imgPaths.Length)];
		}
	}
```

```
}
```

解説　Form1のしくみ

1. メンバ変数　：　画像のパス情報、**Random**クラスのインスタンス用変数（乱数発生用）

画像のパス情報をstring型の配列で用意する。このパス情報は画像を配置した場所によって異なる。

Randomクラスのインスタンス用変数は使いまわすためにメンバ変数とした。

2. コンストラクタ　：　**Random**クラスのインスタンス用変数にインスタンス代入

Randomクラスのインスタンスを生成し、用意しておいたメンバ変数randomに代入。こうすることで、変数randomをどのイベントハンドラからでも利用可能になる。

3. イベントハンドラ「**button1_Click**」　：　表示されている画像を用意した画像1に切り替える

画像1は配列imgPathsのインデックス0が相当するものとし、PictureBoxコントロールのプロパティ「ImageLocation」に値を代入することで画像の切り替えを行っている。

4. イベントハンドラ「**button2_Click**」　：　表示されている画像を用意した画像2に切り替える

画像2は配列imgPathsのインデックス1が相当するものとし、PictureBoxコントロールのプロパティ「ImageLocation」に値を代入することで画像の切り替えを行っている。

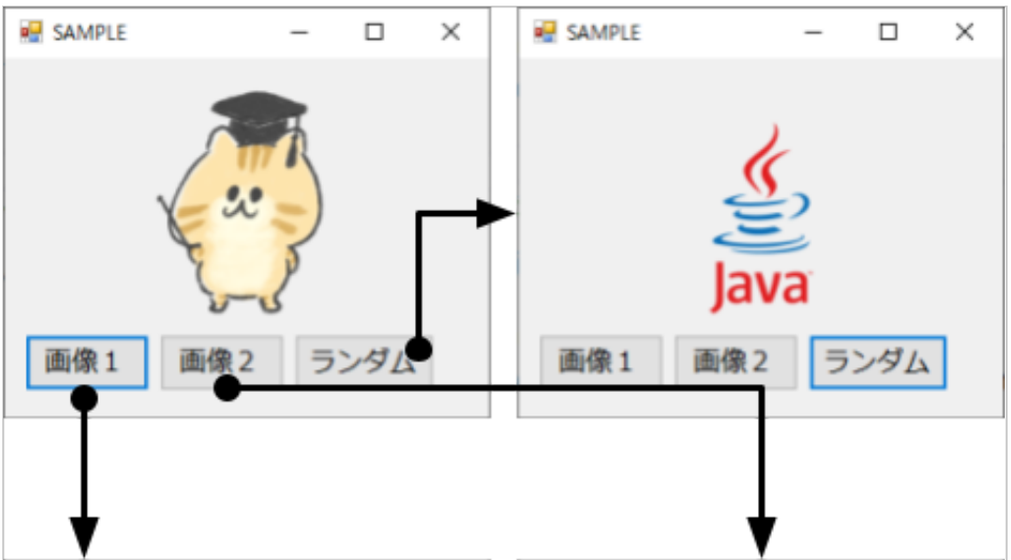
5. イベントハンドラ「**button3_Click**」　：　表示されている画像を用意した画像に切り替える（ランダム）

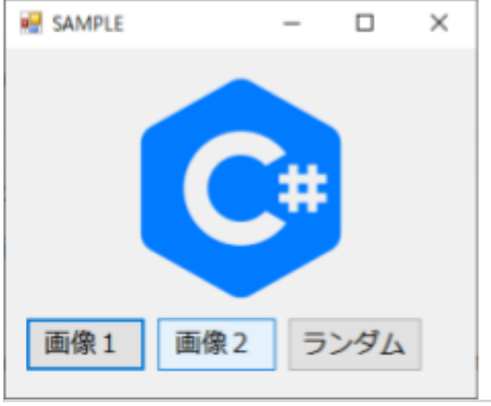
配列imgPathsのランダムなインデックスを選び、PictureBoxコントロールのプロパティ「ImageLocation」に値を代入することでランダムな画像の切り替えを行っている。

Step05　動作確認

動作確認を行う。次のイメージ通りに動作すれば成功。

図8-7　画像表示を切り替えるサンプル





8-4 NewtonSoft.Jsonの使い方

JSON用のパッケージであるNewtonSoft.Jsonの使い方は公式ページのIntroductionで確認できるが、実際にサンプルを作成しながら使い方を紹介する。

Step01 まずはJSONの理解から

JSONを理解しなくてはJSON用のパッケージを使うことは難しいだろう。JSONは最初は難しく見えるが、仕組みを知ってしまえばとても理解しやすいフォーマットである。

JSONの基本フォーマット

JSONは以下のフォーマットが基本である。

```
{"キー名" : 値}
```

具体的な値を当てはめると、次の様になる。

```
{"name" : "takashi"}
```

JSONのデータ型

JSONの値にはデータ型がある。データ型は以下の通り。

- 文字列
- 数値
- 真偽値
- 配列
- オブジェクト
- null

◆文字列

文字をダブルクォーテーションで括って記述する。

```
{"name" : "takashi"}
```

◆数値

半角数字で記述する。

```
{"age" : 16}
```

◆真偽値

true / falseで記述する。

```
{"isPoison" : false}
```

◆配列

角カッコで括り、要素をカンマで区切って記述する。

```
{"HPs" : [50, 98, 0, 100]}
```

◆オブジェクト

波カッコで括ってキーと値をペアで記述する。キー・値のペアが複数存在する場合はカンマで区切る。波カッコの中には文字列や数値などの値・配列・オブジェクトを含むことができる。つまり、オブジェクトの中にオブジェクトがあって、さらにその中にオブジェクトがある・・・といった事が可能。

```
{
      "name" : "takashi",
      "age" : 16,
      "HP" : 98,
      "isPoison" : false
}
```

◆null

JavaやC#のnullと同じ意味で、nullと記述する。

```
{"weapon" : null}
```

JSONのその他のフォーマット

JSONは以下のフォーマットが基本である、と最初に述べた。

```
{"キー名" : 値}
```

上記フォーマットはオブジェクト型としてのデータである。データは必ずオブジェクト型でなければならないルールは無いので、配列のデータを扱っても問題ない。

```
[50, 98, 0, 100]
```

数値や文字列のみの場合、次の様になる。複数の値を列挙したい場合は配列かオブジェクト形式にする必要がある。

```
"takashi"
```

ちなみに、配列に型は無いので以下の様に要素の型がなんでもアリで実現可能。

```
["takashi", 123, "hiroshi", false, null]
```

結局のところ、データのフォーマットはオブジェクト型にしておいて、必要なものだけ中に付け足すのが一番汎用性が高く、わかりやすい（ということを筆者は実体験を元に感じている）。

トークン(token)

数値や文字列、配列、オブジェクトを紹介したが、それぞれの単位をトークンと呼ぶ。JSONはトークンの集まりである。

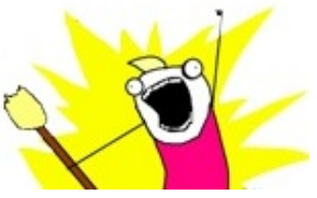
JSONのサンプル

JSONデータを記す。

```
{
      "turn" : 3,
      "bossBattle" : false,
      "players" : [
            {
                  "name" : "戦士",
                  "HP" : 255,
                  "MP" : 0,
                  "status" : {
                        "isPoison" : false,
                        "powerUp" : true,
                        "defenseUp" : true
                  }
            },
            {
                  "name" : "勇者",
                  "HP" : 198,
                  "MP" : 92,
                  "status" : {
                        "isPoison" : false,
                        "powerUp" : true,
                        "defenseUp" : true
```

```
                }
            },
            {
                "name" : "僧侶",
                "HP" : 45,
                "MP" : 188,
                "status" : {
                    "isPoison" : true,
                    "powerUp" : false,
                    "defenseUp" : true
                }
            },
            {
                "name" : "魔法使い",
                "HP" : 120,
                "MP" : 235,
                "status" : {
                    "isPoison" : false,
                    "powerUp" : false,
                    "defenseUp" : false
                }
            }
        ]
}
```

Step02　NewtonSoft.Jsonのクラス群を知る



JSON形式のデータを受け取った場合、名前空間「Newtonsoft.Json.Linq」のクラス群を使えばデータを分解し、プログラムで使うことができる。次のStep03から例を挙げながら使い方を紹介する。

Newtonsoft.Json.Linq Namespace

NewtonSoft.Jsonのクラス群

◆JValue

```
public class JValue : JToken, IEquatable<JValue>,
      IFormattable, IComparable, IComparable<JValue>, IConvertible
```

数値や文字列等の値を表す。

◆JArray

```
public class JArray : JContainer, IList<JToken>,
      ICollection<JToken>, IEnumerable<JToken>, IEnumerable
```

配列を表す。

◆JObject

```
public class JObject : JContainer, IDictionary<string, JToken>,
      ICollection<KeyValuePair<string, JToken>>, IEnumerable<KeyValuePair<string,
JToken>>, IEnumerable, INotifyPropertyChanged, ICustomTypeDescriptor,
INotifyPropertyChanging
```

オブジェクトを表す。

◆JContainer

```
public abstract class JContainer : JToken,
      IList<JToken>, ICollection<JToken>, IEnumerable<JToken>,
      IEnumerable, ITypedList, IBindingList, IList, ICollection,
      INotifyCollectionChanged
```

他のトークンを含むトークンを表す。JArrayクラス、JObjectクラスの基本クラス（親クラス）。

◆JToken

```
public abstract class JToken : IJEnumerable<JToken>,
      IEnumerable<JToken>, IEnumerable, IJsonLineInfo, ICloneable,
      IDynamicMetaObjectProvider
```

トークンを表す。JValueクラス、JContainerクラスの基本クラス（親クラス）。

Step03　検証用アプリを作成する

NewtonSoft.Jsonのクラス群（名前空間「Newtonsoft.Json.Linq」）の使い方を説明するうえで検証用アプリを利用する。検証用アプリはWindowsフォームアプリケーションで作成し、プロジェクト名は任意でよい。筆者はプロジェクト名を「NewtonsoftJsonSample」とした。

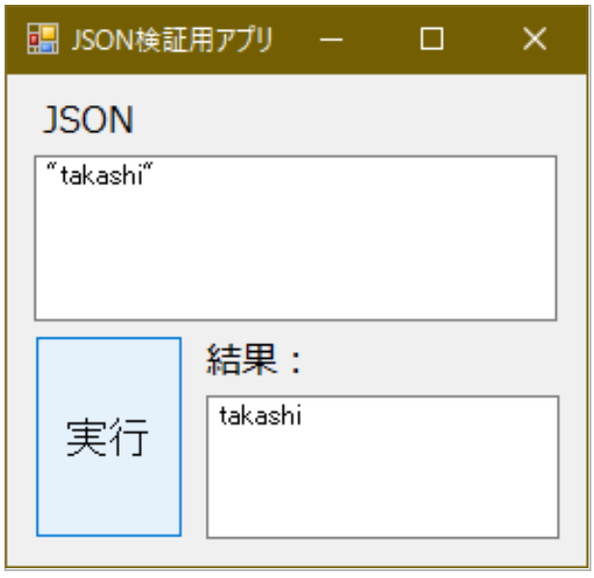
検証用アプリをデザインする

用意したJSONデータをどのようにプログラムで処理するかを紹介していくうえで、次の様な検証用フォームアプリを用意する。

◆デザイン

用意したJSONデータをどのようにプログラムで処理するかを紹介していくうえで、次の様な検証用フォームアプリを用意する。

図8-8　JSON検証用アプリ



「JSON」部のテキストボックスに入力したデータを使って「実行」ボタンをクリックすれば、処理結果が「結果」部のテキストボックスの内容に反映される仕組みとする。

数値や文字列、オブジェクト、配列をJSONデータから取り出すイベントハンドラを実装する。

表8-4　JSON検証用アプリのコントロール一覧

コントロール	**Name**	**Text**	説明
Form	Form1	JSON検証用アプリ	プロジェクト作成時に自動的に生成されるフォーム。
Label	label1	JSON	ラベル。
TextBox	jsonDataTextBox	（JSONデータ）	JSONデータを入力するためのテキストボックス。「Multiline」プロパティをTrueにして複数行入力できるようにしている。
Button	executeBtn	実行	クリック時にイベントハンドラを呼び出すためのボタン。
Label	label2	結果：	ラベル。
TextBox	resultTextBox		結果を表示するためのテキストボックス。「Multiline」プロパティをTrueにして複数行入力できるようにしている。

◆イベントハンドラ

ボタン「executeBtn」のイベント「Click」に値「OnExecuteBtnClicked」を入力し、イベントハンドラを用意。

◆JValueのサンプル

JSONデータ

数値や文字列を用意する。

```
"takashi"
```

イベントハンドラ「OnExecuteBtnClicked」を実装する

以下のコードをForm1.csに記述する。※今回に限りForm1.csの全コードを記述する。次回からはイベントハンドラ「OnExecuteBtnClicked」のみ記述予定。

コード8-5　Form1.cs

```
using Newtonsoft.Json.Linq;
using System;
using System.Windows.Forms;

namespace NewtonsoftJsonSample
{
	public partial class Form1 : Form
	{
		public Form1()
		{
			InitializeComponent();
		}

		private void OnExecuteBtnClicked(object sender, EventArgs e)
		{
			JToken jt = JToken.Parse(jsonDataTextBox.Text);
			JValue jv = jt as JValue;
			resultTextBox.Text = jv.Value.ToString();
		}
	}
}
```

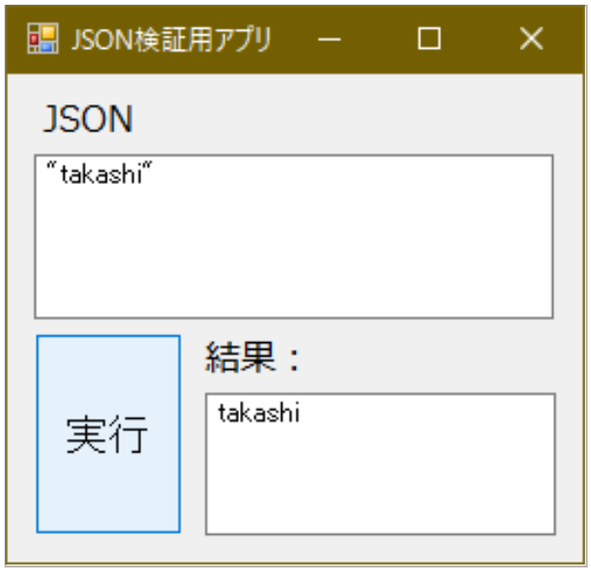
解説

JSONデータはJToken.Parseメソッドを使ってJTokenのインスタンスとすることができる。JToken型のままでは値の取得ができないため、as演算子を使ってJValueにキャストしたのち、プロパティ「Value」で値を取得する。プロパティ「Value」の型はObject型であるため、テキストをセットする場合はToStringメソッドで文字列を取得する必要がある。

実行結果

以下の図の様にJSONデータから文字列「takashi」を取得・テキストボックスに反映されていれば成功。

図8-9　JSON検証用アプリ（JValueのサンプル）



◆JArrayのサンプル

JSONデータ

配列を用意する。

```
[50, 98, 0, 100]
```

イベントハンドラ「OnExecuteBtnClicked」を実装する

以下のコードをForm1.csに適用する。

コード8-6　Form1.cs（イベントハンドラ「OnExecuteBtnClicked」のみ記述）

```
private void OnExecuteBtnClicked(object sender, EventArgs e)
{
	StringBuilder sb = new StringBuilder();
	JToken jt = JToken.Parse(jsonDataTextBox.Text);
	JArray ja = jt as JArray;
	foreach(JValue jv in ja)
	{
		sb.Append(jv.Value.ToString());
		sb.Append(", ");
	}
	resultTextBox.Text = sb.ToString();
}
```

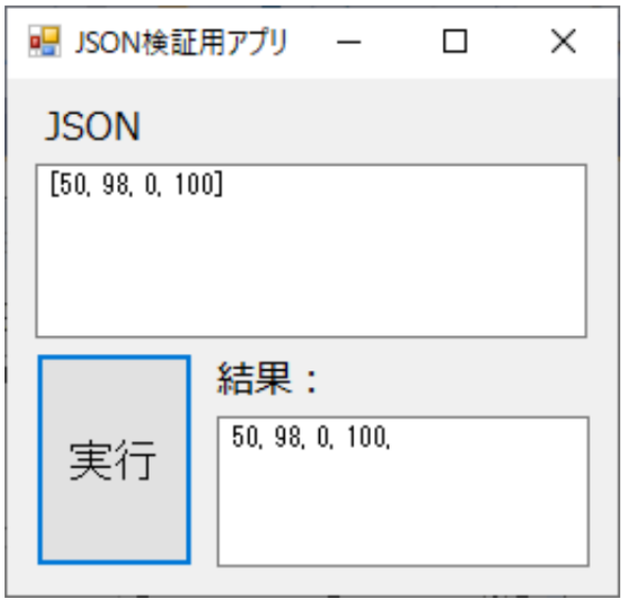
解説

JToken.Parseメソッドを使ってJTokenのインスタンスを得るところまではJValueのサンプルと同じ。今回はJSONデータの中身が配列型なので、as演算子を使ってJArrayにキャストしている。JArrayクラスはIEnumerableインターフェースを実装しているので、foreachが使える。foreachで値を取得（＋今回はカンマ区切り）してやれば、後は結果用のテキストボックスに反映するだけである。

実行結果

以下の図の様にJSONデータから配列を取得・テキストボックスに反映されていれば成功。

図8-10　JSON検証用アプリ（JArrayのサンプル）



◆JObjectのサンプル

JSONデータ

オブジェクトを用意する。

```
{
      "name" : "takashi",
      "age" : 16,
      "HP" : 98,
      "isPoison" : false
}
```

イベントハンドラ「OnExecuteBtnClicked」を実装する

以下のコードをForm1.csに適用する。

コード9-7 PaintApp.xamlのイベントハンドラ"OnExecuteWinChime()"の内容記述

```
private void OnExecuteWinChime(object sender, EventArgs e)
    {
        StringBuilder sb = new StringBuilder();
        var jsonOptions = new JsonSerializerOptions { };
        JObject jo = JObject.Parse(
        ParseJson(jsonParserString, jsonOptions));
        foreach(KeyValuePair<string, JToken> kvp in jo)
        {
            sb.Append(kvp.Key);
            sb.Append("\n\n");
            sb.Append(kvp.Value.ToString());
            sb.Append("\n\n");
        }
        resultTextBox.Text = sb.ToString();
    }
```

解説

JToken.Parseメソッドを使ってJTokenのインスタンスを得るところまでは、Step8-4やステップ4と同じ。今回は、JSONデータの中身がオブジェクト型なので、sbを基準とする使ってJObjectにキャストしているところ、JObjectでラスのforeachインターフェースを実装しているので、foreachが使える。簡単そうなのにsystemParsing_JTokenごとなかでは要求が変更、foreachでキーと値を取得！やはり、JSONはとかでもしくりやんど、単は基礎表現やキスわっかりこれ体するだけである。

実行結果

以下の画面が表示されればデータの表示を取得できるわっかりに注意ありたいの成功。

図9-7 JSONを配列でアプリにDeploy中サンプル

Step8-4 オリジナルアプリを作成する

Yes