# ソースコードの色

予約語 ：int,if,typeof など

Enum ：など

Class ：UserControl など

コメント：

文字列：String

# プロジェクトファイル( .csproj )

## .NET バージョンごとのシンボル

.NET のバージョンごとのシンボル定義の仕方

e.g.

// 一つずつ書くタイプ

<PropertyGroup Condition=" '$(TargetFrameworkVersion)' == 'v3.5' ">

<DefineConstants>NET10;NET20;NET30;NET35;$(DefineConstants)</DefineConstants>

</PropertyGroup>

<PropertyGroup Condition=" '$(TargetFrameworkVersion)' == 'v4.0' ">

<DefineConstants>NET10;NET20;NET30;NET35;NET40;$(DefineConstants)</DefineConstants>

</PropertyGroup>

// 一気に書くタイプ

<PropertyGroup>

<DefineConstants Condition=" $(TargetFrameworkVersion.Replace('v', '')) &gt;= 2.0 ">$(DefineConstants)NET10;NET20;$(DefineConstants)</DefineConstants>

<DefineConstants Condition=" $(TargetFrameworkVersion.Replace('v', '')) &gt;= 3.5 ">$(DefineConstants)NET10;NET20;NET30;NET35;$(DefineConstants)</DefineConstants>

<DefineConstants Condition=" $(TargetFrameworkVersion.Replace('v', '')) &gt;= 4.0 ">$(DefineConstants)NET10;NET20;NET30;NET35;NET40;$(DefineConstants)</DefineConstants>

<DefineConstants Condition=" $(TargetFrameworkVersion.Replace('v', '')) &gt;= 4.5 ">$(DefineConstants)NET10;NET20;NET30;NET35;NET40;NET45;$(DefineConstants)</DefineConstants>

<DefineConstants Condition=" $(TargetFrameworkVersion.Replace('v', '')) &gt;= 4.6 ">$(DefineConstants)NET10;NET20;NET30;NET35;NET40;NET45;NET46;$(DefineConstants)</DefineConstants>

<DefineConstants Condition=" $(TargetFrameworkVersion.Replace('v', '')) &gt;= 4.7 ">$(DefineConstants)NET10;NET20;NET30;NET35;NET40;NET45;NET46;NET47;$(DefineConstants)</DefineConstants>

</PropertyGroup>

情報元

https://code.i-harness.com/ja/q/2c9aca

+α バージョン対応表

https://qiita.com/nskydiving/items/3af8bab5a0a63ccb9893

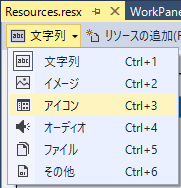
## .NETバージョン確認

インストールされている.NETのバージョン確認

以下のレジストリを参照：

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP

## アイコン

メニュー：プロジェクトのProperties → Resources ダブルクリック。

先にアイコンを選択（右図参照）→　リソースの追加

・16×16, 32×32, 64×64 の３種類あればWindows ではOK。

実行ファイルのアイコン

プロジェクト右クリック → アプリケーションタブ → リソース（GroupBox）→ アイコン

但しこっちの方が楽：

// アイコンの設定。デザイナでごちゃごちゃやるより、こっちの方が確実

this.Icon = Properties.Resources.IconMain;

イメージファイルの違い

ビットマップ画像 ：１ピクセルに１～４バイト持たせて表現

ベクター画像 ：複雑な計算式で図形を表す。

# 基本

## バイト／ビット演算

加算 int added = 0x001 | 0x002; 結果：0x003

削除 flag & ~

判定（論理積） (target & 0x02) == 0x02 0x02 (0010) を含む

排他的論理和 string test = Convert.ToString(0b0110 ^ 0b0011, 2); 　// 結果："0101"

シフト << or >> ※ToStringの第二引数の ”2” は2進数にする事を表している

string test1 = Convert.ToString(0b0010 << 1, 2); // 結果："100"

string test2 = Convert.ToString(0b0010 >> 1, 2); // 結果："1" (001)

バイト変換は System.BitConverter を用いる

int iVal = BitConverter.ToInt32(byteArray, 0);

var byteArray = BitConverter.GetBytes(iVal);

## ループ

Enumerable.Range

foreach (var i in Enumerable.Range(0, 3))

{

}

※実はPythonに影響されて実装したのではなかろうか。

## 文字列操作

・\r\n 改行コード

・Format(ToString)時のフォーマット指定 e.g. {0:x8} ⇒8桁の16進数

x 16進数

・String.IndexOf() 指定された文字列があるか調べ、その位置を知る

(string, StringComparison)

### エスケープ文字

\' シングルクォーテーション「'」

\" ダブルクォーテーション「"」

\\ バックスラッシュ「\」

\0 null文字

\a ベル（警告音）

\b バックスペース

\f フォームフィード

\n 改行（ラインフィード）

\r キャリッジリターン

\t 水平タブ

\u 16進表記のUnicode文字 ※「\u」に続けて16進数4桁を記述する

\U 16進表記のUnicode文字 ※「\U」に続けて16進数8桁を記述する

\x 16進表記のASCII文字 ※「\x」に続けて16進数1～4桁を記述する

\v 垂直タブ

### 削除方法

.Remove(start, length) 開始位置から指定の文字数を削除

.Replace(oldString, newString) newString に””（空白）をいれれば任意の文字列を削除

文字列操作には、StringBuilder というクラスもあるらしい

### 正規表現

var reg = new System.Text.RegularExpressions.Regex(@"(\s)\1{1,}");

string replaced = reg.Replace(originalStr, " ");

### 全半角判断

Unicode文字で判断する場合

カタカナ

return ('\u30A0' <= c && c <= '\u30FF')

|| ('\u31F0' <= c && c <= '\u31FF')

|| ('\u3099' <= c && c <= '\u309C')

|| ('\uFF65' <= c && c <= '\uFF9F');

全角漢字

('\u4E00' <= c && c <= '\u9FCF') ||

('\uF900' <= c && c <= '\uFAFF') ||

('\u3400' <= c && c <= '\u4DBF');

（参考）<https://dobon.net/vb/dotnet/string/ishiragana.html>

### 全角⇔半角の置換

if (targetStr != null && !System.Text.RegularExpressions.Regex.IsMatch(targetStr, @"^[a-zA-Z0-9]+$"))

{ // 全角文字が使われた場合

// 変換可能な全角文字を全て半角文字に変換。 VB のdll を使うor自作する

string halfStr = Microsoft.VisualBasic.Strings.StrConv(cellValStr, Microsoft.VisualBasic.VbStrConv.Narrow, 0);

// 残った全角文字（つまり半角に変換できなかったもの）を削除

string removed = System.Text.RegularExpressions.Regex.Replace(halfStr, @"[^a-zA-Z0-9]", string.Empty);

}

※注意 []内外で^の意味が異なる。　^[a-z] 行頭の小文字 [^a-z] 小文字以外の１文字

（補足）+は直前の文字の繰り返し $は行末文字

### 埋め込み

MyString = $"Arg1:{arg1}IntArg:{intarg}"

## データ型

### プレフィックス

0b バイナリ e.g. int val = 0b0101

0x 16進数 e.g. int val = 0xFF0A

### ToString書式

x 16進数 e.g. string.Format(“{0:x8}”, iVal); ８桁の16進数

※2進、8進はConvert.ToString()メソッドを用いる。

e.g. 2進数の場合 Convert.ToString(val, 2)

r 丸め

c 通貨

### 型比較

var test1 = new TestClass1();

bool isBase = (typeof(TestClass1) == obj.GetType());

### DateTime

文字通り、日時と時間を扱う

コントロールとしてはSystem.Windows.Forms.DateTimePickerがある

<https://learn.microsoft.com/ja-jp/dotnet/api/system.windows.forms.datetimepicker?view=windowsdesktop-7.0>

#### ToString書式

https://dobon.net/vb/dotnet/string/datetimeformat.html

## 型スイッチ

object val;

switch (val)

{

case int i:

break;

default:

break;

}

※判定対象の変数はobject型じゃないとだめらしい。

※上の例のint i のように、使おうが使うまいが変数を生成する必用がある。

Type でやる場合は：

switch (Type.GetTypeCode(type))

{

case TypeCode.Int32:

break;

…

}

## constとreadonly

### const

constフィールドは、コンパイル時定数の扱い。（MSDN）

変数のように扱える定数（暗黙的 static）

宣言時にのみ初期化可能（コンパイル時に値が埋め込まれる）

readonly より実行速度が速い

switch文やデフォルト引数に使える

インスタンスを new した結果は割り当てられない（C#の組み込み型のみ）

※注意 コンパイル時に決定されてしまうため、クラスライブラリなどでpublicなconst定数を定義しそれをアセンブリをまたいで参照してしまった場合、将来ライブラリ側の定数を変更しても参照されたアセンブリ側をコンパイルしなおさない限り更新されないという問題がある(constのバージョニング問題)

### readonly

readonlyフィールドは、実行時定数の扱い。（MSDN）

実際は、読み取り専用の代入不可な変数

宣言時の他に、コンストラクタ内でも初期化可能

定数であるconstよりは、僅かに実行速度が遅い

switch文やデフォルト引数には使えない

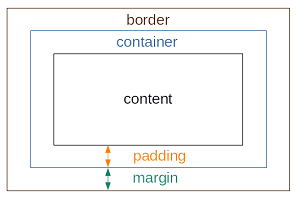
インスタンスを new した結果を割り当てられる

### static readonly

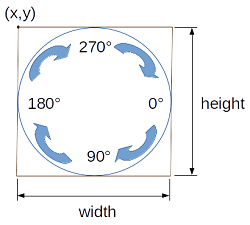
constが使いたいけど、使えない場合に、static readonly を使用する。

## 画像描画

### Location座標



### ArcとPie



引数sweepAngle はn°分進む　という意味で，

n°地点からm°地点までを表すわけでは無い点に注意。

### 注意点

LineやEllipseでPathを作成する場合，必ず一筆書きになるように注意する。

## アプリの実行ディレクトリ

System.IO.Path.GetDirectoryName() メソッドを使う

引数

System.Reflection.Assembly.GetExecutingAssembly().Location dllの場合

System.Reflection.Assembly.GetEntryAssembly().Location exeの場合

注意！GetEntryAssemblyメソッドはアンマネージコードから呼び出された時などではNULLを返す。

参考：<http://smdn.jp/programming/netfx/environment/1_process/>

## 予約語関連

### ?：null 許容型

int? length = list?.Count

→ listがnull だった時はlengthにはnullが入る。

### ??：null 合体演算子

?? の左側の値がnullな場合

int length = list?.Count ?? 0;

→ list がnullだった場合はlengthには0が入る

### using：ライブラリの読み込み

using \_2D = System.Drawing.Drawing2D; // 変数的に使う事もできる

# VB関連

## 予約語の比較

|  |  |  |
| --- | --- | --- |
| C# | VB | 概要 |
| static | shared | 静的メンバ |
| internal | Friend | アセンブリ内のみのアクセス許可 |
| this | Me | Class内で自身を表す |
|  |  |  |
| base | MyBase |  |
| override |  |  |
| typeof | TypeOf |  |

（参考）<https://dobon.net/vb/dotnet/vb2cs/vb2cstable.html>

継承

Partial Class Form1

Inherits System.Windows.Forms.Form

## VB独自のもの

### Option Strict Off

Onの場合は以下の条件でコンパイル時エラーとなる

暗黙的な縮小変換

遅延バインディング

結果が Object 型となる暗黙の型指定

### Option Explicit On

### On Error

エラー処理ルーチンを指定する。

→Tryを使うべきでは？

### StringクラスでVB独自の事項

Space$ スペースの数を指定。

Debug.Print("fuga" & Space(15) & "hoge")

文字列の結合は慣例的に「&」が使われる。「+」でも問題は無かった。

### My

様々な機能を持ったクラスへのエイリアスが含まれる特殊な名前空間。

### DateAndTime

名前空間：Microsoft.VisualBasic

TimeOfDay 但し、DateTime.Todayの時間版と言うべきもので、固定なだけで日付情報も含まれてしまう。

## VBとC#の混在

概要

組織内でVB→C#移行期など、VBとC#の混在を達成したい場合がある

結論

基本的に１プロジェクト内に混在させる事は不可能。

一方で、１ソリューション内に複数プロジェクトを混在させるのは可能な為、C#プロジェクトとVBプロジェクトを分けて片方にプロジェクト参照する。

メモ

・

## メモ

デリゲート（action）と匿名メソッドの書き方

Private Sub timerStyle()

Dim counter As Integer = 0

' タイマー時間経過時に実行する匿名メソッド

Dim action = Sub()

System.Console.WriteLine(counter)

counter += 1 ' カウントアップ

End Sub

' 単位は秒

Dim timer = New System.Timers.Timer(5)

' 時間経過時のイベント登録

AddHandler timer.Elapsed,

Function(sender, e) action

' タイマー開始

timer.Start()

End Sub

## その他

宣言と生成を同時に行う方法

MyClass.MyMethod(New MyModelClass() With {.MyPropeartye = ""})

# 各種機能

## 概要

### 4.8以前

|  |  |  |  |
| --- | --- | --- | --- |
| 機能名 | C# ver | .NET ver | 概要 |
| 非同期  async/await | 5.0 | 4.5 |  |
| タプル  ValueTuple | 7.0(2017.3) | 4.7 |  |
| switchパターンマッチング | 7.0(2017.3) | 4.7 |  |

#### ラムダ式

### .NET Core以降

|  |  |  |  |
| --- | --- | --- | --- |
| 機能名 | C# ver | .NET ver | 概要 |
| Span<T>, Memory<T> | 7.2(2017.11) | Core 2.1 |  |
| Null 許容参照型 | 7.2(2017.11) | Core 2.1 |  |
| Interfaceのデフォルト実装 | 7.2(2017.11) | Core 2.1 |  |
| Range型, Index型 | 7.2(2017.11) | Core 2.1 |  |
| 非同期ストリーム  await foreach | 8.0(2019.9) | Core 3.0 |  |
| switch式 | 8.0(2019.9) | Core 3.1 | switch文の簡略化 |
| Entity Framework |  |  | .NET標準のORM |
| レコードクラス  Record types | 9.0 | 5 |  |
| ホット リロード | 10.0 | 6 |  |
| レコード構造体  Record structs | 10.0 | 6 |  |
| 生文字列リテラル | 11.0 | 7 |  |

### 参考

新機能を超簡単に説明する：<https://qiita.com/nskydiving/items/2ff8285acb72c4e59caf>

<https://qiita.com/toRisouP/items/18b31b024b117009137a>

公式：<https://learn.microsoft.com/ja-jp/dotnet/csharp/>

## 10.0の機能

### レコードクラス

public record Person(string FirstName, string LastName);

## 非同期処理（マルチスレッド）

### Task

#### staticメソッド

##### Task.Run

最も基本的な使い方。

System.Threading.Tasks.Task.Run(() =>

{

System.Threading.Thread.Sleep(5000); // 重い処理

});

なお、Task.Factory.StartNew()　というものもある。基本的には同じらしい。

※.NET4.0でも使用可能なので、XP向きアプリケーションにはこちらを用いる。19.02.07

asyncで使う場合

Task<bool> task = Task.Run(async () =>//async：await を可能にする→非同期メソッド

{ // 非同期メソッドは手順書だととらえる

await semaphoreSlim.WaitOneAsync(); //await：このTaskの完了まで待つ。

return true;

});

bool bRet = task.Result; //結果を取得する

但し、Task.Runは数ある非同期タスクの内、同期で組み込みたい場合、またはTask.Run中に非同期ラムダを書く（ようするに下側[asyncで使う場合] の例）場合以外は極力使わない。

### スレッドの待機

WhenAll

Task<int> hogeAync = Task.Run(async () =>

{

await HevyFunctionAsync();

return 10;

});

var result = await Task.WhenAll(hogeAync); //Taskの配列を引数にし、１つ１つ待って実行

ConfigureAwait

内部処理のawait後に戻る先のスレッドを特定しないようにできる。

private async Task NiceAsyncMethod()

{

await Task.Delay(1000).ConfigureAwait(false); //　スレッドは適当なものが選ばれる

}

SynchronizationContext

参考：Taskの待ち方<https://qiita.com/takutoy/items/d45aa736ced25a8158b3>

### Semaphore

非同期処理で必要になってくるオブジェクトのロック処理をlockより軽く行う。

また、SemaphoreSlim の方にはasync のメソッドがあるので便利。

名前空間System.Threading

コンストラクタ

public SemaphoreSlim (int initialCount, int maxCount);

initialCount [Int32]：同時に許可するリクエスト番号の初期数

この値を0にするといきなりロックされる。別スレッドでReleaseすれば使えるが、そんな使い方はしないと思う。

maxCount [Int32]：リクエスト最大数。initialCount以上である必要がある。

プロパティ

CurrentCount 名前が紛らわしい。現在許可できる残りリクエスト数の事。

オンラインドキュメント：

<https://qiita.com/acple@github/items/8f63aacb13de9954c5da>

### マルチスレッドの注意点

async voidはTaskの進行状況をわざわざ捨てる行為なので、強く非推奨。

TaskをWait()するとデッドロックの可能性が非常に高い。

public void KusoMethod()

{

KusoAsyncMethod().Wait();

}

private async Task KusoAsyncMethod()

{

await Task.Delay(1000);

}

これだけでデッドロックする。

非常に分かりやすい、Web上の説明：

非同期メソッドは、awaitする内部タスクを開始した後、自分のスレッドを一旦解放します。そして、その内部タスクが完了したとき、処理の続きを「前と同じスレッドで」実行します。その間に、「前のスレッドを既に誰かが使っていたら？」「そして、そのスレッドが解放されるためには、Taskの実行が完了しないといけないとしたら？」はい。デッドロックです。

System.Threading.Thread.Sleep メインスレッドと同じスレッドを使って処理するらしい

System.Threading.Tasks.Task.Delay 別のスレッドを使って待機処理を行う

⇒ System.Threading.Thread.Sleepを複数回使うと、アプリケーションがフリーズしてしまう事があるらしい

### delegate

自作のイベントハンドラ（例）

public class MyEventArg : EventArgs

{

public string StringValue { get; set; }

public object Value { get; set; }

}

public delegate void MyEventHandler(object sender, MyEventArg e);

注意点

利用先でイベントハンドラが登録されないままEventがInvokeされるとSystem.NullReferenceExceptionが発生してしまう。それを防ぐため、無意味なイベントハンドルメソッドを初期値として登録した方が良い

public class MyClass

{

public event MyEventHandler MyEventInvoked;

public MyClass()

{

// System.NullReferenceException を防ぐために空の EventHandler を指定しておく

this.MyEventInvoked = (obj, e) => { };

// イベントハンドラ（delegate）を使う側でこのようにnull確認後に実行する手もある

MyEventHandler?.Invoke(this, new MyEventArg());

}

}

## switch式

例

public string MyMethod(object arg)

{

return obj switch

{

int i when x > 0 => i.ToString(),

float f when x > 0 => f.ToString(),

\_ => throw new ArgumentOutOfRangeException();

}

}

## 暗号化

System.Security.Cryptography.RijndaelManaged();　クラスで暗号化ができる。

## COMの利用

COM は自身がどれだけ参照されているかを示す「参照カウンタ」を持っており、それを０にしてやる事で自身を解放する。

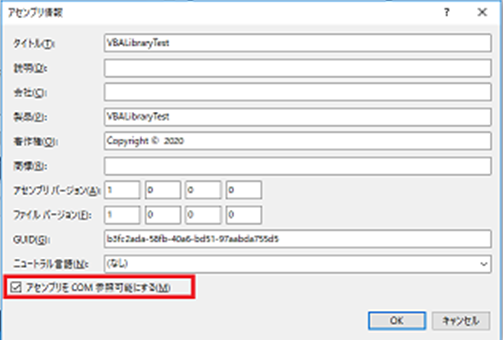
具体的には以下のメソッドを利用する。前者がカウンタを１減らし、後者は０にする。

System.Runtime.InteropServices.Marshal.ReleaseComObject

System.Runtime.InteropServices.Marshal.FinalReleaseComObject

## 別環境での実行

### VBA



1.　 C#（.NET）：COMからの参照を可能に

クラスライブラリのプロジェクトを作成し、プロパティ→ アセンブリ情報

→ COMからの参照を可能にするに✓

２．C#（.NET）：GUIDを作成。

Visual Studioのツール → GUIDの作成。

その後、クラスを作成し、そのクラスに以下の２つのアトリビュートをつける

System.Runtime.InteropServices.ProgId() MyNamespace.MyClassName　と文字列で指定

System.Runtime.InteropServices.Guid() アプリごとに作られるID。先ほど作成したもの。

３．コマンドプロンプトでregasm.exe 使う。

regasm.exe はC:\Windows\Microsoft.NET\Framework\v4.0.30319　などにある。

.NET のバージョンに応じた別々の実行ファイルがある

regasm /codebase /tlbファイル(DLL)のフルパス ：：登録

regasm /codebase /tlb ファイル(DLL)のフルパス /u ：：登録解除

なお、変更が生じるたびに登録解除・登録をする必要があるらしい。

このとき、ExcelなどDLLを使用するアプリケーションが起動しているとエラーになってしまう模様。

４．エクセル：アセンブリを参照

VBA画面（Alt+F11）で　メニュー：ツール → 参照設定

２．　で付けたProgId の名前空間名が表示されているはずなので、チェックを入れる。

VBAのソースコードはこのようになる：

Sub test()

Dim obj As New MyNamespace.MyClassName

MsgBox obj.Hello

Set obj = Nothing // リソースの開放

End Sub

情報元）<https://sites.google.com/site/torimemoforwork/vba/cdepuroguramuwoshuitevbadeshiyongsuru>

（CやC++で作ったものはDLL読み込み[Declare Function/Sub] を利用する点に注意）

# プリディレクティブ

#pragma warning disable IDE0017

エラー一覧に出てくる警告：IDE0017 が表示されないようにする。（当然、非推奨）

# .NET資料

## 命名規則

<https://dobon.net/vb/dotnet/beginner/namingrules.html>

## .NETバージョンについて

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ver | Windows | Win Server | Visual Studio | C# | 新機能 |
| 1 | 2000, XP | 2000, 2003 | 2002 | 1 |  |
| 1.1 | 2000, XP, Vista | 2000,2003,2008 | 2003 | 1.2 | ODBCとOracle Database用のデータ接続  IPv6 |
| 2 | 2000, XP, Vista, 7, 8, 8.1, 10 | 2000,2003,2008,  2012, 2016 | 2005 | 2 | 64ビットシステム  .NET Micro Framework  [C#]ジェネリック　[C#]イテレータ |
| 3 | XP, Vista,7,8, 8.1, 10 | 2003,2008,  2012, 2016 | 2005 | 2 | WPF・WCF・WF・WCS |
| 3.5 | XP, Vista,7,8, 8.1, 10 | 2003,2008,  2012, 2016 | 2008, 2010 | 3 | ASP.NET MVC　ASP.NET AJAX  [C#]LINQ　[C#]ラムダ式  [C#]暗黙的型付け(var) |
| 4 | XP, Vista, 7 | 2003, 2008 | 2012 | 4 | F#言語  Dynamic Language Runtime  MEF　Velocity　Windows タッチ  ADO.NET Entity Framework  [C#]Parallel Extensions(Parallel/PLINQ)  [C#]dynamic型  [C#]オプション引数/名前付き引数 |
| 4.5 | 7, 8, 8.1, 10 | 2008, 2012 | 2013 | 5 |  |
| 4.6 | 7, 8, 8.1, 10 | 2008,2012, 2016 | 2015 | 6 |  |
| 4.7 | 7, 8, 8.1, 10 | 2008,2012, 2016 | 2017 | 7 |  |
| 4.8 | 7, 8, 8.1, 10 | 2008,2012, 2016,2019 | 2019 | 7.3 |  |
|  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ver | Windows | Win Server | Visual Studio | C# | 新機能 |
| Core 1.0 | 7, 8, 8.1, 10 | 2012, 2016 | 2015 Update 3, 2017 | 6 | ASP.NET Core  オープンソース化  Mac/Linux/iOS/Android対応 |
| Core 1.1 | 7, 8, 8.1, 10 | 2012, 2016 | 2015 Update 3, 2017 | 6 | 1380個のAPI追加 |
| Core 2.0 | 7, 8, 8.1, 10 | 2012, 2016 | 2017 15.5 | 7.1 | .NET Standard 2.0対応  ASP.NET Core 2.0  Entity Framework Core 2.0  ML.NET |
| Core 2.1 | 7, 8, 8.1, 10 | 2012, 2016 | 2017 15.8.6 | 7.2 | ASP.NET Core 2.1 Entity Framework Core 2.1 .NET Core グローバルツール HttpClient のパフォーマンス改善 Windows互換機能パック [C#]Span<T>, Memory<T> |
| Core 2.2 | 7, 8, 8.1, 10 | 2012, 2016 | 2017 15.9 | 7.3 | ASP.NET Core 2.2  Entity Framework Core 2.2 |
| Core 3.0 | 7, 8, 8.1, 10 | 2012, 2016 | 2019 | 8 | .NET Standard 2.1対応  WPF・Windows Forms  [C#]Null 許容参照型  [C#]Interfaceのデフォルト実装  [C#]非同期ストリーム  [C#]Range型, Index型 |
| Core 3.1 | 7, 8, 8.1, 10 | 2012, 2016 | 2019 16.4 | 8 | C++/CLI対応　ASP.NET Core 3.1  Entity Framework Core 3.1 |
| 5 | 7, 8, 8.1, 10 | 2012, 2016 | 2019 16.8 | 9 | 「.NET Framework」と「.NET Core」を統合　ASP.NET Core 5.0　Entity Framework Core 5.0 |

（参考）<https://qiita.com/nskydiving/items/3af8bab5a0a63ccb9893>

## Linq

### 便利な拡張メソッド

Except(IEnumerable<T>) 対象のlistに含まれない要素を抽出する。

TakeWhile　　　条件を満たす間、要素を取得します。

values.TakeWhile(v => v < 9)

SkipWhile　　　条件を満たす間、要素をスキップします

SkipWhile(v => v < 9).Write();

ThenBy 　　　SortByと併用して並び替えの第二キーとして用いる。

ThenByDescending　　　　ThenByの降順バージョン

### 要素番号

Select 拡張メソッドには引数２つのオーバーロードがあり、要素のindex 番号を利用できる。

但しクエリ式では利用できない 18.07.18

例）

var indexArray = this.\_list.Select((x, i) => x.IntMember > 0 ? i : -1);

### 動的クエリ

// 動的ラムダ自体は以下のようにして作成する

var lamda = System.Linq.Expressions.Expression.Lambda<Func<int, int, bool>>();

var param = System.Linq.Expressions.Expression.Parameter(typeof(TestClass), "x");

// where(TestClass x => まで

var left\_val = System.Linq.Expressions.Expression.Property(param, "IntMember");

// where(TestClass x=> x.IntMember まで

var right\_val = System.Linq.Expressions.Expression.Constant(10, typeof(int));

var body = System.Linq.Expressions.Expression.GreaterThan(left\_val, right\_val);

var lamda = System.Linq.Expressions.Expression.Lambda<Func<TestClass, bool>>(body\_min, param).Compile();

var match = this.\_list.Where(lamda)

## Timer

System.Windows.Forms.Timer

OnTick イベントで処理を行う。

同期なので、コントロールにもアクセスできる。

System.Windows.Threading.DispatcherTimer　???

同期なので、コントロールにもアクセスできる。

System.Threading.Timer

•unsafe

•軽量

•一回作成すると設定したタイマーイベントの内容が変更できない

var theadTimer = new System.Threading.Timer((\_) => Debug.WriteLine("Threading.Timer {0}", Thread.CurrentThread.ManagedThreadId));

System.Timers.Timer

•unsafe

•サーバベース・タイマと呼ばれ、高精度

ただし、16ms以下のインターバルはOSの制約から指定しても意味がないらしく、実際にはほぼ同等らしい

System.Diagnostics.Stopwatch stopwatch

ただ時間を測るだけ

Reset : Stop 状態にして時間を戻す。再度Start する必要がある。

Restart : Run の状態のまま時間だけ戻す。

## Config

// config で自作クラスを保存する方法

[global::System.Configuration.UserScopedSettingAttribute()]

public System.Collections.Generic.List<string> Lines

{

get

{

return ((System.Collections.Generic.List<string>)(this["Lines"]));

}

set

{

this["Lines"] = value;

}

}

→ 但しconfigファイルが不安定になった記憶がある18.09.13

## Controls

### Control 全般

#### Invoke

別スレッドからのコントロールへのアクセス

if (this.InvokeRequired)

{

this.Invoke(new MethodInvoker(() => { /\* 処理 但し回帰した方が楽\*/ }));

}

else

{

/\* 処理 \*/

}

19.06.05 Invoke はリソース的負荷が非常に大きい事を知る。以下注意点：

・細かすぎず、処理時間が長くなりすぎない範囲で取る。

・下のような方法がある事を知る。これでwait すればOK かも？

await Task.Run(() => this.Invoke((MethodInvoker)(() => textBox1.Text = i.ToString())));

レベル高い資料<http://bbs.wankuma.com/index.cgi?mode=al2&namber=6843&KLOG=18>

⇒代替案としてBackgroundWorker を使ってみてもいかも。

##### Invokeの違い

・Delegate.Invoke ：同じスレッドで同期的に実行

・Delegate. BeginInvoke ：スレッドプールスレッド上で非同期に実行

・Control.Invoke ：UIスレッドで実行。呼び出しスレッドは完了するまで待機(PostMessage)

・Control.BeginInvoke ：UIスレッドで実行。呼び出しスレッドは待機しない

#### Designer.cs

内では名前空間を含んだクラス名を書かないと、グラフィックデザイナ上で編集できなくなる。

#### Paintメソッド

#### Text プロパティの継承

Textプロパティに以下の２つのアトリビュートをつけないとデザイナで編集できない。

[System.ComponentModel.Browsable(true)]

[System.ComponentModel.DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]

#### ドラッグ＆ドロップ

まずは対象コントロールのAllowDropプロパティをTrueにする。

DragEnterイベントハンドラと、DragDropイベントハンドラを用いる。

#### ツールチップ（ヒント）

・DataGridView

CellToolTipTextNeededイベントを利用、EventArgs.ToolTipText で設定。

・フォーム

ToolTip1.Show(string, IWin32Window, Point, int); ※第4引数はduration

・その他コントロール

System.Windows.Forms.ToolTipを使用、.SetToolTip(Control, string) メソッドで設定。

デザイナで設定した方が楽。

・BorderColor

境界線の色を変えたい時があるが，.NET標準ではコントロールの境界線の色を変える事は出来ない。

唯一できるのが，FlatStyle をFlatにする方法。以下を参考。

Button btn = (Button)this.Controls.Find(“btn”, true)[0];

btn.FlatStyle = FlatStyle.Flat

btn.FlatAppearance.BorderColor = Color.Red;

btn.FlatAppearance.BorderSize = 3;

それ以外ではカスタムコントロールを作って，自分で描写するしかない。

・Tag プロパティ

各controlにobject型の値を割り当てる事ができる。e.g. menuItem1.Tag = hoge;

#### サイズ変更イベント

オーバーライドできるサイズ変更イベントにはOnSizeChangedとOnResizeがある。

OnSizeChangedイベントは内部でOnResizeを呼び出している。

(On)Resize：サイズ変更による再描写が必要な時に呼び出される。

(On)SizeChanged：実際のサイズ変更をトリガとするイベントハンドラ。

注意点として，SizeChangedイベントはFormなどのInitializeComponent でも発生する。

#### 多言語化

##### 基本

CultureInfoのコンストラクタはCultureInfo.Name を引数に取る

リソースファイル名はRecources.[xxx].resx とし、このxxxはCulture名と完全に同じにしておく。

大本は英語にしておく。オーバーライド用ファイルないに特定プロパティが無くても、大本の方を自動的に利用するらしい。

Formを継承したクラスにはLanguage(string)とLocalizable(bool)というプロパティがあるが、そこに直接値を入力してもよい。その場合、そのFormファイルの階層下にある（例Form1.resx）にリソース情報を埋め込む事になり、煩雑だし、ファイル数も多くなる。

日本語はja-JP

##### 現在選択中のCultureについて

以下の４プロパティがあるが、下の２つはThreadクラスからもアクセスできるというだけで、実質同じインスタンスを参照している、との事。

System.Globalization.CultureInfo.CurrentCulture;

System.Globalization.CultureInfo. CurrentUICulture;

System.Threading.Thread.CurrentThread.CurrentCulture;

System.Threading.Thread.CurrentThread.CurrentUICulture;

##### ２つのCultureの違い

CurrentCulture 日時の書式、数値の表示形式など。

CurrentUICulture ユーザーインターフェイスに表示されるメッセージなど。

（参考）<https://smdn.jp/programming/netfx/locale/0_abstract/>

### WindowsForm 関連

#### Load, Dispose のイベント

Load 時のイベントの順番

•コンストラクタ

•Control.HandleCreated

•Control.BindingContextChanged

•Form.Load フォームが

•Control.VisibleChanged

•Form.Activated

•Form.Shown フォームが初めて表示された時に１回だけ発生

（参考）<https://docs.microsoft.com/ja-jp/dotnet/desktop/winforms/order-of-events-in-windows-forms?view=netframeworkdesktop-4.8&redirectedfrom=MSDN>

Dispose時のイベントの順番

•Form.Closing

•Form.FormClosing

•Form.Closed

•Form.FormClosed

•Form.Deactivate

#### Show/Close

##### イベント

Shown フォームが初めて表示された時に１回だけ発生する

##### メソッド

ShowとShowDialogの違い

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| メソッド | 概要 | Closeした時 | Disposeする方法 | 備考 |
| Show | モードレスで開く。  （他のダイアログの操作が可能） | Unload＋Dispose | Close又はDisposeメソッドを呼び出す。 |  |
| ShowDialog | モーダルで開く。  （他のダイアログの操作は不可） | インスタンス維持 | Disposeメソッドを呼び出す。 |  |

参考：<https://dobon.net/vb/dotnet/form/showform.html>

Close モードレス(Show)で開いた場合はShow() する事で再表示が可能。

⇒ Dispose() を呼び出さないとリソースは開放されない。

モーダル(ShowDialog)で開いた場合はリソースが開放される。

Hide フォームを非表示にする。.Visible = false しただけ。

#### サブダイアログのテンプレ

this.MaximizeBox = false;

this.MinimizeBox = false;

this.ShowIcon = false;

this.ShowInTaskbar = false;

this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle;

this.StartPosition = System.Windows.Forms.FormStartPosition.CenterParent;

#### DialogResult

例えば、Formを継承したクラスの中でbuttonClick イベントハンドラで以下の様にすると、buttonClick イベントが終了した後にフォームがCloseする。

重要なのは、別スレッドで行ってもそれが動作する、と言う点。

this.DialogResult = OK; // 自身のDialogResulに値を入れる

#### Key​Preview プロパティ

通常KeyDownイベントは、現在フォーカスをもっているコントロールだけで発生する。

Formなど親コントロールでも発生させる為には、このプロパティにtrueを代入する。

また、KeyDownイベントで用いるKeyPressEventArgsの　Handled プロパティにtrueを入れる事で、実際にKeyDownが発生した子コントロールではKeyDownイベントを発生させない事もできる。

### TextBox

.SelectedText = “hoge”; で現在のカーソル位置に任意の文字列を挿入できる。

.SelectionStart でカレット位置を取得できる。

### NumericUpDown

.DecimalPlaces 小数点の位置。

### ComboBox

#### キーボードからの直接入力

DropDownStyle プロパティの値で挙動が変わる

DropDown ドロップリスト＋直接入力※１

DropDownList ドロップリストのみ

※１　背景色が灰色っぽくなってしまう。変えるためにはオーナードローするしかない。

### DataGridView

#### DataGridView全般

##### セルのイベント発生順

マウスでクリック

CellBeginEdit なぜか値の編集とみなされる

CellEndEdit 即座にEndEditが発生

F2やクリックなどで編集を開始

CellBeginEdit 値の編集に突入

CellDirtyStateChanged del, bs,キー操作を開始した瞬間に１回発生

CellParsing 入力値の解析と型変換をカスタマイズ。まだ編集中である。

編集をカーソルの移動、Enterキー押下、他セルやオブジェクトへのフォーカスの移行で発生：

CellLeave 自セルがフォーカスを失うと発生。（Enterやカーソルボタンなど）

CellValidating 移動により値がコミット。（編集中の値の検証に用いる）

CellValueChanged ここで値が変更されていれば発生。（変更がなければ発生しない）

編集が終了すると発生

CellValueChanged 値が変更されていれば発生。（上で発生していれば発生しない）

CellDirtyStateChanged 値のコミットによってもう一度発生。ValueChangedの後に必ず発生。

CellValidated 最終的に検証が完了したとみなされる。CurrentCellDirtyStateChanged

**カーソル移動が無ければValidatingが発生しない点に注意！**

CellEndEdit ここで編集終了とみなす。セル移動があればCellEnterが発生。

##### VirtualMode

e.g.)

.VirtualMode = true ; //

バインド、非バインド、仮想　の内の一つで、仮想モードは bind 処理を自身で(override?) 実装する。　※注意点！ 仮想モードの時は rows.Insert ができない。

ちなみに非バインドはデータが readonly で、小さいサイズのデータを表示だけする場合に適している、との事

参考）ContextMenuStripMenuNeeded イベント発生に必要な設定項目

##### BindingList

BindingList＋DataGridViewは非常に便利だが，気を付ける点も多い。

・IList とBindingListの変換

new BindingList<TestClass>( \_list );

・AllowUserToAddRows = true の時に　BindingList でバインドして、

バインドされたフィールドで items.Add(); するとエラーが起こる。バインドした \_bindingList側を追加しなければならない。

・DataGridViewの新規行をCellContentClickなどのプログラムで処理しようとすると上手くいかない。

デバッグすると \_bindingListの要素が増えているのにGUI（表示）の方は追加前のListを維持している。Web上でもたびたび話題になる有名なバグ。

その場合，DataSourceを一度nullにしてからの再バインドが必要だが，DataSourceをnullにした瞬間に\_bindingListの方まで追加前のListに戻ってしまうという，更なるバグがある。（バグだらけ）

結果として，以下の様な処理が必要になる。少しマヌケだが，これしか方法がない。

var temp = new List<TestClass>(this.\_bindList1);

this.dataGridView1.DataSource = null;　　// この時点でthis.\_bindList1の要素数が変化する

this.\_bindList1 = new BindingList<TestClass>(temp);

this.dataGridView1.DataSource = this.\_bindList1;

・IEnumerable<T> は表示できない。ToList() などにすると表示される。

・Rows[index または名前].DataBoundItem で割り当てた object を取得できる。

・DataSourceに値を割り当てる前に初期化しないと表示がうまく行かないらしい　18.07.06

this.dataGridView1.DataSource = new List<SomeClass>();

##### 右クリック

CellContextMenuStripNeededイベント：

RowContextMenuStripNeededイベント：

・Rowヘッダー上で右クリックをしても発生しない。

・セル部を右クリックした際に、CellContextMenuStripNeededの後に発生する。

#### DataGridViewで，値編集完了を捕捉する

##### 状況別の補足方法：BindingList

BindingListを使用する場合

BindingListにListChangedというイベントがあるが，発生条件がやや複雑。

|  |  |  |
| --- | --- | --- |
| アクセス先 | 発生 | 例 |
| プロパティ　（※） | × | this.\_bindingList1[i].IntMember = 1; |
| 要素 | 〇 | this.\_bindingList1[i] = item; |
| BindingListのメソッド | 〇 | this.\_bindingList1.Add(item); |
| DataSource | × | dataGridView1.DataSource = this.\_bindList1; |
| DataSourceをnull | 〇 | dataGridView1.DataSource = null;  ※なぜかnullを入れる時は発生する！ |

*（※）ComboxColumnなどを使った場合，DataPropertyNameのプロパティにアクセスするので，*

*ListChangedイベントは発生しない。*

注意点：

\_bindingListバインドしている時に要素を削除(※1) → ListChanged という流れになるが，このイベントハンドラの時点では何故かDataGridViewインスタンスの行数と \_bindingListのCount数が一致しない。例えば要素数3→2の時，dgv.RowCount は4(※2)で\_bindingList.Countは2となる。

その為か、ここでdgv.Refresh などを行うと整合性が取れない為かエラーが発生する。(※3)

⇒ 結論的にはこのイベントハンドラも使わない方がいいかもしれない。マイクロソフト，こんなんばっか。

*（補足1）追加の時は，DataGridViewの追加行セルに侵入した瞬間に \_bindingListのCountも増えていたので問題にはならないはず。*

*（補足2）調べてみると削除の時はBindListChangedイベントの後にDataGridView.RowsRemovedイベントが発生している。削除の時はRowsRemovedイベントを使った方が良いかもしれない。*

（※１.）RemoveAtメソッドなど直接だろうが，DataGridViewのDelボタンだろうが同じだった。

（※２．dgv現在行3＋1[新規行] と思われる）

（※３.）DataErrorイベントハンドラで管理は出来るので，どうしても必要ならそのように対処する。

##### 状況別の補足方法：その他

DataGridViewButtonColumnの場合

基本的に値の変更を捕捉するイベントはない。上のBindingList.ListChangedを使えるように工夫するか，CellContentClickで値を捕捉して，なんらかの処理を自力で書くことになる。

DataGridViewComboBoxColumnの場合

CellValidatedが適当。ComboBoxColumnは実質このイベントハンドラしか選択肢がない。

逆にComboBoxColumnが無ければこのイベントハンドラはあまり使わない。（かも）20.09.09

DataGridViewCheckBoxColumnの場合

DataGridViewTextBoxColumnの場合

##### 値編集に関する補足

編集中に値の整合性対策を行うならば，CellValidatingイベントが適切。

DataGridViewの値編集終了イベントはいくつかあるが、CellValidated イベントが最も適切。

なぜCellEndEditではダメ？20.09.09

（参考）[セルのイベント発生順](#_セルのイベント発生順)

DataGridView.DataErrorイベントを利用する事で，ユーザーが不適切な値を入力した事を捕捉できる。

BindingListのListChanged イベントのEventArgで，対象Listにどのような変化が起こったのかも捕捉できる。

jsonとenum

#### DataGridViewColumns

###### 背景色

Column1.Style.Fill.SetBackgroundColor(Color.FromArgb(227,227,227);

##### DataGridViewComboboxColumns

this.ColumnCmb.ValueMember = "Value";

this.ColumnCmb.DisplayMember = "Display";

this.ColumnCmb.DataSource = (from EnumA em in Enum.GetValues(typeof(EnumA))

select new {Value = em, Display = em.ToString\_FromEnum()}).ToArray();

// ToStringEnum という拡張メソッドをあらかじめ作っておく

上の方法でList<T> などをDataSourceに割り当てられる。

※但しListの中に空白に対応した要素が無いとエラーが起こる。

※ValueMember に文字列を割り当てた場合はDisplayMemberの値でComboboxの値に代入する事はできない。

なお、SelectedIndexChanged はEditingControlShowingイベントとCellEndEditイベントを組み合わせて行う。

参考：<http://dobon.net/vb/dotnet/datagridview/selectedindexchanged.html>

###### 動的なDataTableを割り当てる

var itemTable = new DataTable("ItemTable");

itemTable.Columns.Add("Display", typeof(string));

itemTable.Columns.Add("Value", typeof(int));

for (int idx =0; idx < items.Count; idx++)

{

var item = items[idx];

string text = item.ToString();

itemTable.Rows.Add(text, idx);

};

this.Column1.ValueMember = "Value";

this.Column1.DisplayMember = "Display";

this.Column1.DataSource = itemTable;

#### DataGridViewRows

・DataBoundItem　DataSource で割り当てたobject を取得

・DefaultCellStyle.BackColor =

##### 行を手動で追加する

var row = new DataGridViewRow();

row.CreateCells(this.dataGridView1); // 列情報を伝える

targetFieldList.ForEach(x =>

{

int clmIdx = this.dataGridView1[x.DeviceName].Index;

row.Cells[clmIdx].Value = "hoge";

});

※但しVirtualMode = true だと機能しない 18.12.13

#### CurrentRow　の注意点

新規行が有効（AllowUserToAddRowがTrue）でフォーカスがdataGridViewにない時はCurrentRowが正しく取得できない。フォーカスを失った時点でCurrentCell が自動的に移動する仕様らしい上、厄介なことにRowEnterイベントなどもここで発生する。

解決策：SelectedCells プロパティなどで取得するしかない。

#### CellToolTipTextNeeded

このイベントはDataSourceプロパティが設定されているか、VirtualModeプロパティがTrueの時にしか発生しない

（補足）CellToolTipは特定のセル上にポインタを一定時間以上置いたときに表示されるTip文字列。

#### その他のメソッドなど

DefaultValuesNeeded 新規行に入った時などのデフォルト値を操作する。

### TableLayoutPanel

元コントロールのサイズに合わせて自動的に位置、サイズを変更してくれるコントロール。

#### 注意事項

・内部に配置するコントロールの.Dockプロパティは大体Fillを選択する

・デザイナでFillを選択するとデザイナが使いにくくなるので、利用するFormのコンストラクタあたりでやっておいた方が良い。

#### セルの結合

TableLayoutPanelのプロパティではなく、配置するコントロールのRowSpan/ColumnSpanプロパティ。

### MenuStrip

・境界線 ： “-” で境界線が引かれる

・DropDownOpening イベント：表示時に発生する。動的メニューが作れる。

see also [コンボボックスでenum を選択](#_コンボボックスでenum_を選択)

### PictureBox

動的に画像を作成するには、Bitmapオブジェクトを作成し、Graphicsオブジェクトを用いてjpgなどを**描写**し、必要があればその上から文字列を**描写**する。

参考）<https://dobon.net/vb/dotnet/graphics/pictureboximageanddrawimage.html>

#### プロパティ

##### ImageLocation

PictureBox1.ImageLocation = "C:\test.bmp"

##### SizeMode

|  |  |
| --- | --- |
| Normal | 画像をPictureBoxの左上に表示する。画像の大きさは、基のまま。 |
| StretchImage | PictureBoxのサイズに合わせて画像を拡大または縮小して表示。 |
| AutoSize | PictureBoxの大きさを画像の大きさと同じにして表示する。 |
| CenterImage | 画像をPictureBoxの中央に表示する。画像の大きさは、基のまま。 |
| Zoom | 画像を比率は保持して拡大／縮小して表示。.NET Framework 2.0以降。 |

##### 画像ファイルのロック

Image.FromFile(bmpPath) で画像ファイルを読み込むと、通常ファイルがロックされてしまうので、必ずDisposeされるような処理とする必要がある。

また、FileStream で画像を読み出すと、ファイルがロックされない。

using (var fs = new System.IO.FileStream(fileName, FileMode.Open, FileAccess.Read))

using (var img = System.Drawing.Image.FromStream(fs))

{

// 処理

}

## アトリビュート

### デザイナ

[System.ComponentModel.Category("表示")] デザイナのプロパティ分類を指定する

[System.ComponentModel.Browsable(true)] デザイナに表示するか。

※Text プロパティをoverride の際は通常表示されない。もしくは隠したいプロパティに。

[System.ComponentModel.DesignerSerializationVisibility()]

デザイナで変更、つまりダーティなプロパティをシリアライズの対象にするか決める。

Visible 対象；Contentコレクションの要素を対象に； Hidden除外する

情報元：<http://d.hatena.ne.jp/Kazzz/20061014/p1>

[System.ComponentModel.Description("説明文")]

[System.ComponentModel.DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)]

デザイナでの編集結果をresx にシリアライズする

Content オブジェクトの内容のコードを生成。コレクションの要素など。

Hidden シリアライズ対象から除外。シリアライズに向かない自作クラスなどに適す。

Visible シリアライズする。Textプロパティなど、値型に適す。

[DefaultEvent("ButtonDown")]：デザイナでのダブルクリック時のイベントハンドラの指定。

### シリアライズ

[System.Xml.Serialization.XmlRoot("class")]: クラスに付けてタグの名前を変更する

[System.Xml.Serialization.XmlIgnore] : シリアライズしないメンバ

[System.Runtime.Serialization.DataContract]：クラスに付けて、そのクラスをjsonなどのシリアライズ対象にする

[System.Runtime.Serialization.DataMember]：メンバに付けて、そのメンバをjsonなどのシリアライズ対象にする

[System.Runtime.Serialization.IgnoreDataMember]：そのメンバをjsonなどのシリアライズ対象から外す。

※System.Runtime.Serialization を使う際は，参照　からライブラリの追加が必要。

### その他

using System.Runtime.InteropServices;

[System.Diagnostics.DebuggerDisplay("Count = {Count}")] ：ウォッチリストで見る表題

指定していない場合はToString() メソッドが使われるらしい。

# 構造化

## IEnumable

下の方に簡易的なクラスレファレンスを書いたが、配列系は非常に多用される為、専用の項を作った。

### それぞれの違い

おそらく多用するのは以下の４つ

配列(Array)

System.Array: IList, ICollection, IEnumerable,

特徴：要素数固定長。あまり使われないが、低レベルデバイスなどと通信する際などに使う。

リスト(List)

System.Collections.Generic. List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IEnumerable, IList, ICollection　など

特徴：要素の追加、削除が容易。

辞書(Dictionary)

System.Collections.Generic.Dictionary:　 IEnumerable, IDictionary, ICollection など

System.Collections.Generic. List<T>

特徴：分類ごとに要素をまとめる。GroupByの結果なんかをDictionaryに変換する事が多い。

System.Collections.IEnumerable

クラスでは無いので比較対象ですらない。

特徴：forなどの反復処理で使えるようにする。

### メソッド

#### Contain

対象オブジェクトのEquals メソッドを用いる。また、Array (配列)にはContain メソッドはなく、Enumerable 系に変換して行う。またはindexOf を用いる。

→ Any メソッドの方が楽(.NET 3.5以上)

#### Sort()

対象オブジェクトのCompareTo メソッドを用いる。引数、戻り値なし。

対象クラスは[IComparable](#_IComparable)を継承している必要がある。

(compare) OrderBy

#### OrderBy

拡張メソッドなどを引数とし、IEnumerable<TSource>を返す。

Guid.NewGuidを用いるとシャッフルができる：参考（[Listのシャッフル](#_Listのシャッフル)）

(compare) Sort

#### Reverse

#### Count

機能はもちろん説明不要だが、型によって少しずつ違う点があるので注意が必要。

IEnumalbeではメソッドなのでCount()としないと使えない。

ListではCountはプロパティで，Arrayでは同機能別名プロパティLengthがある。

⇒ なぜ統一しなかった？

### メソッド(List)

Listクラスにしか無いメソッドが多く存在する。しかもそのほとんどが非常に便利。

### 雑多情報

#### インスタンスの扱いに関する注意点

例えば var target = this.\_list.ToArray(); などとすると、コレクションのオブジェクトは新規生成されるが、肝心の中身の参照先は変わらないらしい。（対象オブジェクトの書き込み前に加工が必要で、書き込み後も引き続き編集する場合にうまくいかない）

⇒対策：要素のクラスでICloneableを継承し \_list.Select(x => (TestClass)x.Clone());

19.01.07

#### コレクションのlinq / クエリ式利用

コレクションであるクラス（BaseCollectionを継承？）にEnumerable.Cast メソッドを用いると、普通に Where などの拡張メソッドが利用できる。ここではDataGridViewのRowsメンバを用いる。

e.g.

var target = this.dataGridView1.Rows

.Cast<DataGridViewRow>()

.Select(x => (TestClass1)x.DataBoundItem).ToList();

#### 全要素に一括代入する方法

方法１

for (int i = 0; i < list.Count; i++) list[i].CurrentValue = null;

方法２

list.ForEach(x => x.CurrentValue = null);

方法３

list = list.Select(x => { x.CurrentValue = null; return x; }).ToList();

#### FindIndexAll

Enumerable.​Select(T, int) を用いるとFindIndexAll のような事が可能。

var indexArray = this.\_list.Select((x, i) => x.IntMember > 0 ? i : -1);

// .Where(x => x != -1);　のような事をして -1 を消す必要がある

#### Listのシャッフル

Guid.NewGuidを用いる

using System;

using System.Linq;

int[] ary1 = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int[] ary2 = ary1.OrderBy(i => Guid.NewGuid()).ToArray();

## Class

### Method

#### オプション引数

以下の様にデフォルト値を付けると、呼び出しの際に省略できる

public void Wait(int milisecond= 5000){}

### 継承

#### アップキャスト

継承クラスから基底クラスへキャストする事。通常 var a = b のように型変換すら不要。

public class BaseCls　{　public int IntMember { get; set; } 　}

public class InheritedClass : BaseCls {　public string StringMember { get; set; }　}

BaseCls bs;

bs = new InheritedCls (); // 全く問題ない

#### ダウンキャスト

基底クラスから継承クラスへキャストする事。

// 悪い例

InheritedClass inherit;

inherit = new BaseCls(); // この様にはできない。（コンパイルエラー）

as を使うと、キャストできない場合にnullになる。

var bs = new BaseCls()

var inherit = bs as InheritedCls; // null になる

継承関係にあるクラスはoperatorによるキャストが使えないので、このようにコンストラクタで対応すると良い

public class InheritedCls : BaseCls

{

public InheritedClass(){ /\* デフォルトコンストラクタが必要 \*/ }

public InheritedClass(BaseCls arg)

{

this.IntMember = arg.IntMember;

}

}

var inherit = new InheritedCls ( new BaseCls () ); // この様にする。

#### パラメータ付きのコンストラクタの継承の仕方

public NewClass(int arg) : base(arg)

{

}

### 実装、overrideすべき メソッド

#### Equals

・== と Equals メソッドの違い

簡単にまとめると、等値演算子(==)はオーバーロードされるが、Equalsメソッドはオーバーライドされる。例えば、以下の例のようにstringとobjectの比較だと、それぞれのクラスでオーバーロードされたメソッドが使われる。

⇒ **両方実装する必要がある**

string a = “hello”; string b = “hello”;

Console.WriteLine(a == b); //true

object c = a

Console.WriteLine(c == b); //false;

Enumerable.Contains メソッドなどで等価確認の際に用いられるメソッド。

注意点 ：Equals メソッド内では例外を発生させてはいけない。

public override bool Equals(object obj)

{

//objがnullか、型が違うときは、等価でない

if (obj == null || this.GetType() != obj.GetType())

{

return false;

}

//この型が継承できないクラスや構造体であれば、次のようにできる

//if (!(obj is TestClass))

//比較する

var instance = (TestClass)obj;

return this.Number == word.Number;

//return (this.Number.Equals(c.Number)); // これでも良い

}

#### GetHashCode

注意点は

// 1. オブジェクトが等しい時には同じ値を返すように実装する

// 2. オブジェクトが異なる時にはユニークな値型を返すようにする

// 3.

public override int GetHashCode()

{

return this.ID;

}

#### operator

* != も実装する必要がある。 =>!(arg1 == arg2) とする。

public static bool operator ==(TestClass arg1, TestClass arg2)

{

//nullの確認

if (object.ReferenceEquals(arg1, arg2))

return true;

//どちらかがnull

if (((object)arg1 == null) || ((object)arg2 == null))

return false;

return (arg1.Message == arg2.Message) && (c1.Number == c2.Number);

}

＝＝と！＝を実装したら bool Equal(object) を、 （参考）[Equals](#_Equals)

< と > を実装したら、IComparable を実装する。（参考）[IComparable](#_IComparable)

## Enum

### SelectedValueとSelectedItem

SelectedValue を使う：

ValueMember プロパティに値を入れないと上手く動かない。

現在の選択項目の指定にはSelectedValue に適合するobjectを代入する。恐らくEqual メソッドで判定が行われ、見つからなかった場合にはSelectedIndexに-1が入る。

SeletecedItem を使う：

ValueMemberはnullでないと上手く動かない。

⇒ 表示名と値を別々に指定する場合

DisplayMember, ValueMember と匿名クラスを利用する：18.12.03

// ※ToStringFromEnum() と言う拡張メソッドを用意しておく  
this.comboBox1.DisplayMember = ”Text”;

this.comboBox1.ValueMember = ”Value”;

this.comboBox1.DataSource = (from SomeEnum num in typeof(SomeEnum).GetEnumValues()

select new { Text = enum.ToStringFromEnum(), Value = num }).ToList();

注意点としては、SelectedItemはこの匿名メソッドのインスタンスになるので、SelectedValueで値の出し入れをする必要がある。

### Enum をループしつつ List に変換する方法

var enumList = new List<SomeEnum>(Enum.GetValues(typeof(SomeEnum)).Cast<SomeEnum >().ToList());

※DataGridViewComboBoxColumnでも全く同じ

### コンボボックスでenum を選択

this.Column\_enum.ValueType = typeof(SomeEnum);　 　// DataGridView の場合

this.comboBox1.DataSource = Enum.GetValues(typeof(SomeEnum));　// Combobox の場合

typeof(TestEnum).GetEnumValues();　　　//もしくはこちら

### キャスト

まず、文字列への変換、int(Int32)への変換は楽にできる。

#### 文字列からの変換

// 第三引数はignoreCase – 大文字と小文字の区別をなくす

var colorBlue = (Colors)Enum.Parse(typeof(Colors), “Blue”, true);

#### int からの変換

var intVal = 1;

var colorRed = (Colors)Enum.ToObject(typeof(Colors), intVal);

#### ToStringFromEnum

EnumのToString() メソッドは結構時間がかかる。ToStringFromEnumなどと言う名の拡張メソッドを作った方が、自由度も早く、処理も早くなる

static class ExtensionMethods

{

public static string ToStringFromEnum(this SomeEnum arg)

{

switch (arg)

{

case SomeEnum.A: return "列挙A";

default: return null;

}

}

}

### 関連メソッド

#### Type

##### IsEnum

##### IsEnumDefined

（情報元）<https://webbibouroku.com/Blog/Article/enum-parse>

## Interface

### ICollection

public List<WordFormat> Items

{

get => this.\_items;

private set => this.\_items = value;

}

private List<WordFormat> \_items; // ※List である必要はない

public int Count { get => this.\_items.Count; }

public void Add(WordFormat arg)

{

semaphore.Wait(); // lock でも良い

if (!this.\_items.Any(x => x == arg)) // Contains() でも良い

{

\_items.Add(arg);

}

semaphore.Release();

}

//System.Collections.ICollection の要素を System.Array にコピーします。System.Array の特定のインデックスからコピーが開始されます。

// index: コピーの開始位置となる、array の 0 から始まるインデックス。

public void CopyTo(Array array, int index)

{

foreach (int i in array)

{

array.SetValue(i, index);

index = index + 1;

}

}

// スレッド セーフである場合は true。

// 配列の処理、例えば foreach はスレッドセーフではないのでロックする必要がある。

public bool IsSynchronized { get { return false; } }

/\*

\* このように実行されるらしい。

\* lock(myCollection.SyncRoot)

\* {

\* // Some operation on the collection, which is now thread safe.

\* }

\*/

public object SyncRoot { get => this.\_items; }

public IEnumerator<WordFormat> GetEnumerator()

{

// ※パターン1、パターン2はどちらの方法で実装しても良い

/\*

// パターン1：

foreach (LanguageStringStruct item in Items)

{

yield return item;

// yield キーワード

// IEnumerator または IEnumerable インタフェースを戻り値するイテレータブロックでつかう

}

\*/

// パターン2

return Items.GetEnumerator();

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

return Items.GetEnumerator();

}

参照：[IComparable](#_IComparable)

### IComparable

Sort など、並び替えの比較で用いる。 (compare) IEquatable：Distinctなどの比較

public int CompareTo(object obj)

{

//nullより大きい

if (obj == null)

{

return 0;

}

//違う型とは比較できない

if (this.GetType() != obj.GetType())

{

throw new ArgumentException("別の型とは比較できません。", "obj");

}

//このクラスが継承されることが無い（構造体など）ならば、次のようにできる

//if (!(other is TestClass)) { }

return this.ID - ((WordFormat)obj).ID;

}

### ICloneable

クローン作製（Shallow Copy）をサポート

public object Clone()

{

return this.MemberwiseClone();

}

以下の様な型変換方法がある

var cloned\_instance = my\_class\_obj.Clone() as MyClass;

### IDisposal

MSDN による情報

<https://docs.microsoft.com/ja-jp/dotnet/standard/garbage-collection/implementing-dispose>

但し、COMオブジェクトなどのアンマネージドソースを解放しようとすると、「RaceOnRCWCleanup が発生しました。」の例外が発生する。RCWとはRuntime Callable Wrapper (ランタイム呼び出し可能ラッパー)。

結論

マネージドソースはDisoposal メソッドで解放

アンマネージドは destructor で解放。

疑問

マネージドソースはいずれGC で解放されるので、IDisposal メソッドを継承する意味ってほとんど無い？強いて言うと アンマネージドなメンバを含んでいますよ、とユーザーに伝える。

### IEquatable

Equal メソッドをoverride する必要がある。[Equals](#_Equals) を参考にする。但し、object でなく、IEquatable<TestClass>.Equals(TestClass arg)　が必要。

# NET ライブラリ

## System.Drawing

// システムカラー

System.Drawing.SystemColors

## System.Collections

### Generic

//ジェネリックのインスタンス生成方法

//this.instance = (T)Activator.CreateInstance(typeof(T));

// メンバの取得

var members = typeof(T).GetMembers(

System.Reflection.BindingFlags.Public |

// プロパティの取得

var propInfo = typeof(T).GetProperty(FieldName);

var propList = typeof(T).GetProperties()

### Dictionary

・既に存在するキーを持つ要素を追加しようとするとエラーになる。

## System.Data

### System.Data.SqlClient.SqlConnection

DbConnectionがIDisposableを継承している

### SqlClient.SqlDataReader

### System.Data.SqlClient.SqlDataAdapter

Dim command = New System.Data.SqlClient.SqlCommand(SQLString, connection)

'SQLの実行

'Dim reader = command.ExecuteReader()

Dim adapter = New System.Data.SqlClient.SqlDataAdapter(command)

'結果をDataTableクラスに格納

adapter.Fill(table)

SystemEvents.SessionEnding

## System.Reflection

// 指定クラスのTypeオブジェクトを取得する

Type t = typeof(TestClass);

// インスタンスの生成

object instance = .InvokeMember(null, System.Reflection.BindingFlags.CreateInstance,

null,

null,

new object[] { });

// フィールドの取得　※見つからなかった場合は null が返ってくる

System.Reflection.FieldInfo fd = t.GetField("FieldMember");

### PropertyInfo

プロパティの取得　※プロパティを GetField で取得できない場合はnull が返ってくる

Type t = typeof(TestClass);

System.Reflection.PropertyInfo pi = t.GetProperty("PropertyMember");

### MemberInfo

プロパティ

DeclaringType

ReflectedType

## System.IO

### DirectoryInfo

最終アクセス日時など、様々な情報が取得できる。

⇒ ディレクトリの作成、削除などを行う時、２つ以上の作業がある場合はSystem.IO.Directory（静的クラス）よりも、このDirectoryInfo クラスを用いた方が良い。

#### 一つ上のディレクトリ名を取得する場合

var dirInfo = new System.IO.DirectoryInfo(@“C:\data\src”);

string parent = dirInfo.Parent;

### Path

ファイル名取得系

GetDirectoryName() ディレクトリ名

GetFileName() ファイル名

GetTempFileName()

一意な名前を持つ 0 バイトの一時ファイルをディスク上に作成し、そのファイルの完全パスを返します。

但し拡張子などを選択できないので、あとで加工する必要があるらしい

### StreamWriter

テキストデータをストリームに直接書き込む

using (var sw = new System.IO.StreamWriter("c:\test.txt", true, System.Text.Encoding.UTF8))

{

sw.Write("hoge"); // 文字列を書き込む

sw.WriteLine("fuga"); //　\r\n 付きで書き込む

sw.Close();

}

文字コードはSystem.Text.Encoding.GetEncoding("shift\_jis") でも指定できる。

### TextWriter

テキストデータを書き込むための抽象クラス（テキストデータをストリームに直接書き込む）

#### TextWriter.Synchronized

using (var sw = new System.IO.StreamWriter(txtFilePath, true, System.Text.Encoding.UTF8))

using (var tw = System.IO.TextWriter.Synchronized(sw))

{

tw.WriteLine(DateTime.Now);

tw.Close();

}

### Ports.SerialPort

メソッド

ReadExisting 現在利用可能なバイトを取得する。

ReadLine 終端文字（ターミネータ）が出てくるまでバッファでため，完了してから返す。

## System.Windows

### Forms

#### FormClosingEventArgs

System.Windows.Forms.FormClosingEventArgs

## Microsoft.Win32

### SystemEvents

## Exception

### 雑多情報

#### 例外フィルター

catch (WebException e) when (e.Status == WebExceptionStatus.RequestCanceled)

{

}

### System

ArgumentException メソッドの引数が変な場合。ArgumentNullExceptionやArgumentOutOfRangeException以外の場合で変な時に使う。

ArrayTypeMismatchException 間違った型の要素を配列に格納しようとした場合。

ArgumentNullException 引数がnullの場合。

ArgumentOutOfRangeException メソッドの許容範囲外の値が引数として渡された場合。

ArithmeticException 算術演算によるエラーの基本クラス。OverflowException, DivideByZeroException, NotFiniteNumberException以外の算術エラーを示したければ使う。

OverflowException 算術演算やキャストでオーバーフローが起きた場合。

DivideByZeroException 0で割ったときのエラー。

FormatException 引数の書式が仕様に一致していない場合。

IndexOutOfRangeException 配列のインデックスが変な場合。

InvalidCastException 無効なキャストの場合。

InvalidOperationException 引数以外の原因でエラーが起きた場合。

ObjectDisposedException Dispose済みのオブジェクトで操作が実行される場合。

NotFiniteNumberException 浮動小数点値が無限大の場合。

NotImplementedException メソッドが未実装の場合。

NotSupportedException 呼び出されたメソッドがサポートされていない場合、または呼び出された機能を備えていないストリームに対して読み取り、シーク、書き込みが試行された場合。

NullReferenceException nullオブジェクト参照を逆参照しようとした場合。いわゆるぬるぽ。

PlatformNotSupportException 特定のプラットフォームで機能が実行されない場合。

TimeoutException 指定したタイムアウト時間が経過した場合。

### System.Collections.Generic

KeyNotFoundException コレクションに該当するキーが無い場合。

### System.IO

#### IOException

IOに関する例外の基本クラス。下に該当するのがあればそちらを優先して使う。

DirectoryNotFoundException ディレクトリが無い場合。

FileNotFoundException ファイルが無い場合。

EndOfStreamException ストリームの末尾を超えて読み込もうとしている場合。

### その他

#### System.Text.RegularExpressions.Regex

正規表現を扱う

メソッド

IsMatch

Replace

構文：Regex.Replace("{置換対象文字列}", "{正規表現パターン}", "{置換パターン}")

例：

var reg = new System.Text.RegularExpressions.Regex(@"(\s)\1{1,}");

string replaced = reg.Replace(originalStr, " ");

# NuGet

## パッケージマネージャー

Alt を使うとAlt+ T→N→O

ただ、参照から「NuGetパッケージの管理」の方が無難に思える。

# VisualStudio関連

## リモートデバッグ

Microsoftの公式ツールを使う事でリモートデバッグが可能

（補足；Visual Studioインストール時に入っている事もある）

<https://docs.microsoft.com/ja-jp/visualstudio/debugger/remote-debugging?view=vs-2019>

以下のファイルをデバッグ対象のPCに配置

C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\Common7\IDE\Remote Debugger

開発アプリに応じてx86, x64を選択して実行

補足：

ツール→オプションのTCP/IPポート番号 でポート番号は確認できる。

Visual Studioを開き、デバッグ → プロセスにアタッチ をクリックします。

接続先の欄に、[リモートPCのIPアドレス]:4022

資料

公式：<https://docs.microsoft.com/ja-jp/visualstudio/debugger/remote-debugging-csharp?view=vs-2019>

参考：<https://qiita.com/DogFortune/items/5874f845a5ad6b9ab1ef>

## .pdbファイル

Program DataBase

Microsoftが用いる、デバッガー向きの情報を格納しているファイル。

# アプリケーション関連

## Microsoft Office 関連

### Excel

・Microsoft.Office.Interop.Excel

メリット

マクロも使用できる。

デメリット

COM 参照なのでオブジェクトの解放が面倒。

Excel がインストールされている環境でしか使えない。

ClosedXML

NuGet で手に入れる 18.06.19 うまく行かない事があった

メリット

後述の NPOI よりも扱いやすい

デメリット

OpenXML を利用している為、旧型式(.xls) を扱えない。

NPOI

メリット

ClosedXML より速い

データベース

## ORM

### System.Data.SqlClient.SqlConnectionクラス

SQLクエリを直接記述する

using (SqlConnection connection = new SqlConnection())

{

connection.ConnectionString = connectionString;

connection.Open();

string selectquery = "SELECT id, name, age FROM member WHERE age >= @age ";

SqlCommand selectcommand = new SqlCommand(selectquery, connection);

selectcommand.Parameters.AddWithValue("@age", 25);

SqlDataReader reader = selectcommand.ExecuteReader();

while (reader.Read())

{

Console.WriteLine("id={0},name={1},age={2}", reader[0], reader[1], reader[2]);

}

reader.Close();

### Entity Framework

.NET標準のORM。

Visual Studioで追加→新しい項目→データ > ADO.NET Entity Data Model→「データベースからCode first」を選択。

→なんかめんどくさそう

### Dapper

どうも一番人気らしい。

### SqlKata

### fluentmigrator

## PostgreSql

Npgsql

NET 4.5.Xに対応した最新のNpgsqlを必ず用いる。古いバージョンのNpgsqlと4.5系だとバグがあるらしい。

# メール

ImapClient というライブラリがあるらしいが？

# インストーラー

## レジストリ

「プログラムの追加と削除」に表示させる為の設定

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

DisplayName ：「プログラムの追加と削除」に表示される名前を指定。

UninstallString ：アンインストールを行う実行ファイルのパスを指定

# 便利ツール

## dnSpy

.NET限定の逆コンパイラ

## Sandcastle

XMLコメントから、ライブラリのドキュメントを作成してくれる。

## C#⇔VB変換

### Webサイト

<https://converter.telerik.com/>

<https://codeconverter.icsharpcode.net/> △

### CodeConverter

Visual Studio extensionとして付け加える（らしい）  
<https://github.com/icsharpcode/CodeConverter>

# その他

## 豆知識

### VisualStudio のオフラインインストール

１　公式のインストーラーをダウンロードする。vs\_community\_\_\*\*\*\*.exe　など。

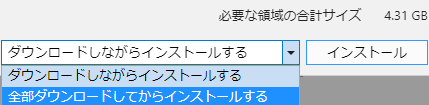
２　ダウンロードしたディレクトリでコマンドプロンプトを起動。ディレクトリでshift+右クリック　でも可能

３　コマンドを実行

# cmd

vs\_community\_\_\*\*\*\*.exe --layout c:\vslayout --lang ja-JP

などと入力。対象の言語などによってコマンドのオプションが変わる。

もしくは，webインストーラーを起動→　対象の言語などを選択 →　全部ダウンロードしてからインストールするでもいいらしい。（右図参照）

（参照）<https://docs.microsoft.com/ja-jp/visualstudio/install/~>　※めずらしく公式の情報が分かりやすい

### null と0の違い

C言語においては、nullはヌルポインタ、0 は\0でヌル文字として扱われる。

define NULL ((void\*)0) // C言語におけるNULLの定義　0と言う値への空ポインタ

但しC++では define Null となっており、同等とみなされる。

## Win10 Toast 通知機能

この通知機能はUWP(Windows Storeアプリ)向きのもの。

まずはWinRT APを使えるようにプロジェクトを編集：

<https://ufcpp.wordpress.com/2012/09/18/デスクトップ-アプリからのwinrt-api利用/>

通知自体は簡単：

var toastNotifier = Windows.UI.Notifications.ToastNotificationManager.createToastNotifier();

但し、xmlで引数を渡すので少しめんどくさい。

方法は：

・自力でxmlを書く

・Windows.UI.Notifications.ToastNotificationManager.GetTemplateContent(); を使う

（参考）<https://stackoverflow.com/questions/37541923/how-to-create-informative-toast-notification-in-uwp-app>

関連事項

Microsoft.Toolkit.Uwp.Notifications

機能：システムへの通知機能

→ 対象ライブラリ数がやたらと多いのでできれば避けたい

System.Windows.Forms.NotifyIcon()　でも通知できるらしいが？

## 別プログラムの実行

System.Diagnostics.Process やSystem.Diagnostics.ProcessInfoを使う。

var pInfo = new ProcessStartInfo();

app.FileName = "test.cmd";

app.UseShellExecute = true;

## アカウント情報

user: h.yaezawa@m-tec-m.co.jp

pass: Mtec6248257

プロダクトキーはVolume Licensing Center で確認できる。

※この処理の際に一時コードが必要となり、八重沢さんのアカウントにメールで送信される。

ライセンスID: 68388331

Key: X7PNM-MQ9BY-8YDXH-RDVJC-76D68

マルチスレッド処理に関するプロパティは foreach を避けた方が良い。

UserControl を継承したクラスで、デザイナを用いた際に自動生成される InitializeComponent に注意！

すべての public property を初期値で生成するコードが自動で書かれる。

その為、例えば private List<T> \_item = new List<T> のようにフィールドでインスタンス生成したメンバがあっても、

その \_item にアクセスする public property があると、InitializeComponent で初期値が入ってしまい null が代入される