Effective C# 4.0

[C#][Visual Studio][言語まとめ C#]

イディオム

■アクセス可能なデータメンバの代わりに常にプロパティを使用すること

・.NET Framework は、データメンバーを公開する場合、プロパティとして公開されることを前提としています
・プロパティはメソッドとしての言語機能をすべて備えるため、virtual キーワードを指定することも出来る
・プロパティの自動実装構文が言語によりサポートされる
  ・バッキングストアと呼ばれる private メンバ変数、アクセッサーをコンパイラが自動実装

```
public class Foo
{
    public string Name
    {
        get;
        set;
    }
}
```

・プロパティの自動実装構文と同じ構文を使用して、抽象クラスやインターフェースのメンバープロパティを実装出来る

```
public interface IHoge<T>
{
    T Value
    {
        get;
        set;
    }
}
```

型が持つデータを public または protected として公開する場合、常にプロパティを使用するべき。一連のデータやディクショナリを含む場合、インデクサを定義する。

■ const よりも readonly を利用する

```
// コンパイル時定数
public const int Millennium = 2000;
// 実行時定数
public static readonly int ThisYear = 2014;
```

・コンパイル時定数はメソッド中で定義することも出来るが、実行時定数はできない。
・コンパイル時定数は、コンパイル時にリテラル値に置き換えられる。
・コンパイル時定数はプリミティブ型の中で、コンパイル時にリテラルに置き換えられる物のみ指定できる。

```
// コンパイル時定数指定不可
public static DateTime BirthDay = new DateTime(1971, 9, 30);
```

コンパイル時に値が決まっていなければならない場合は、const 利用必須だが、まれなケース。リリースを重ねても値が不変である必要がある。そうでなければ、readonly を利用し、柔軟な処理を行えるようにすべき

■キャスト時には is あるいは as 演算子を使用すること

・ダウンキャストには 2 つの方法がある
　・as 演算子を利用する
　・C 言語以来のキャストを利用する
・できる限り as を利用するのが最良の選択

as 演算子はキャスト操作とよく似ています。 ただし、変換可能でない場合、as は、例外は発生せず、null を返します。

```
object o = GetObject();

Base b = o as Base;

if (b != null)
{
    // 変換したオブジェクトを利用する
}
else
{
    // 変換出来ない
}
```

■ #if の代わりに Conditional 属性を使用する

・#if/#endif を多用するとソースが読みにくくデバッグしにくくなる
・Conditional 属性が付与されたメソッドは、環境設定のにしたがって実行の有無が切り替えられる

```
public void Test()
{
    CheckState();
}

[Conditional("DEBUG")]
public void CheckState()
{
    Console.Write("Now Debugging.");
}
```

・Conditional 属性によって生成される中間言語は、#if/#endif で生成されるものより効率的
・メソッド単位でのみ指定可能という制限のため、条件により実行可否を変えるコードを明確に分離出来る

■ ToString() を常に実装すること

・複雑な情報を持つ型を作成する場合には、IFormattable.ToString() を実装し、書式が指定された文字列表現を取得できるようにする。
・型固有の情報を返す ToString() メソッドを実装することで、開発者、使用者の時間の節約になる。
・C#3.0 以降では、コンパイラがすべての匿名型に対し、デフォルトの ToString() メソッドを作成する。

```
var test = new {Name="Me"};
Console.WriteLine(test);
```

結果

```
{ Name = Me }
```

■さまざまな同値性メソッドの関係を把握する

■ GetHashCode の罠に注意する

■ループの代わりにクエリ構文を使用すること

■独自の API では変換演算子を定義しないこと

■メソッドのオーバーロードを最小限にするよう、オプション引数を使用する

■機能を最小限かつシンプルにすること

リソース管理

■割り当て演算子よりもメンバ初期化子を使用すること

■ static メンバは適切に初期化する

■初期化ロジックの重複を最小化する

■ using および try...finally を使用してリソースの後処理を行う

■不必要なオブジェクトの生成を避けること

■ Dispose パターンの標準的な実装

■値型と参照型の違い

■値型における 0 を正常な状態にすること

■値型は不変かつアトミックにすること

デザインの表現

■型の可視性を制限すること

■継承よりもインターフェースの定義および実装を行うこと

■インターフェースメソッドと仮想メソッドの違いを理解する

■デリゲートを利用してコールバックを実現する

・デリゲートはタイプセーフなコールバックを定義します。
・デリゲートの最も一般的な用途はイベントと組み合わせる方法だが、それだけに限らない
・互いのクラス間でデータをやり取りする必要があるが、互いのインターフェースを使用するほどには密に連携させたくない場合最善の選択肢
・対象を実行時に決定でき、複数の相手に同時に通知できる

デリゲートとはメソッドへの参照を含んだオブジェクト

・C# にはデリゲートを表現するためのラムダ式形式の構文が用意されている
・Predicate<T>,Action<>,Func<> など一般的なデリゲート形式が数多く用意されている

■イベントパターンの実装により通知を行うこと

■クラス内オブジェクトの参照を返さないようにすること

■型はできるだけ、シリアル化可能にすること

■粒度の粗いインターネットサービス API を作成する

■ジェネリックの共変性と反変性をサポートする

フレームワークの活用

■イベントハンドラよりもオーバーライドを優先すること

■ IComparable<T> と ICompare<T> を実装して順序関係をサポートする

■ ICloneable を使用しないこと

■親クラスの変更に応じる場合のみ new 修飾子を使用すること

■基本クラスに定義されたメソッドをオーバーロードしないこと

■ PLINQ が並列アルゴリズムを実装する方法

■ I/O のコストが高い処理に対して、PLINQ を使用する方法

■例外を考慮した並列アルゴリズムを構成すること

動的プログラミング

■ dynamic の利点と欠点を把握する

■ジェネリック型引数の実行時型を活用するために dynamic を使用する

■匿名型を引数にとれるよう dynamic を使用する

■ DynamicObject あるいは IDynamicMetaObjectProvidor を使用してデータ駆動の dynamic 型を作成する

■ Expression API を活用する方法を把握する

■式を使用して事前バインディングを遅延バインディングに切り替える

■公開する API では動的オブジェクトを最小限に抑えること

その他

■ボックス化、ボックス化解除を最小限に抑える

■完全にアプリケーション固有の例外クラスを作成する

■例外を強く保証すること

■安全なコードを採用すること

■ CLS **互換性のあるアセンブリを作成すること**

■より小さく凝集したアセンブリを作成すること