属性

ユーザー定義宣言情報

C# には、C++ や Java の特性を受け継ぎ、その機能を拡張するだけではなく
従来の言語にはない強力な新機能を搭載しています
その機能のひとつとして上げられるもののの一つが**属性**です

これまで、ユーザーが作成するクラスや構造体、メソッドやフィールドなどには
静的かインスタンスか、公開か非公開かなどを指定する修飾子を用いました
これらの修飾子は、全て言語設計者が作り出したものです

属性は、これら修飾子に加えて C# 型と情報を関連付ける手段として提供されます
従来の言語に慣れすぎている人は、この新しい概念は理解に苦しむかもしれません
しかし、この機能によって本来はリソースや定数として扱っていた情報を
直接クラスに関連付けられるので、XML や Web との融合を目的とする .NET の開発に適します

属性とは、属性クラスのことをあらわします
属性クラスとは **System.Attribute** クラスの派生クラスを指します
私たちが属性を定義する時は、この Attribute クラスを継承します

public abstract class Attribute

このクラスを継承したクラスは、属性クラスとなります
そう、属性の実体は単純に Attribute を継承したクラスなのです
例えば、次のように属性クラスを宣言したとしましょう

```
using System;
enum KittyName { RENA , YUKI , MIMI }

class KittyAttribute : Attribute {
 public readonly KittyName name;
 public KittyAttribute(KittyName name) {
  this.name = name;
 }
}
```

KittyAttribute クラスは Attribute クラスを継承しているので属性クラスです
このクラスは宣言情報として KittyName をコンストラクタから受け取ります

さて、これで新しい宣言情報となる属性クラスを定義することができました
属性をクラスやメンバに指定する場合、それらの宣言構文を思い出してください

[attributes] [modifiers] class identifier [:base-list] { class-body }[;]

これは、クラスの宣言構文です
最初の attributes が属性宣言子群を指定する場所です
ここに、属性クラスのコンストラクタを指定することでクラスと関連付けます

```
[KittyAttribute(KittyName.RENA)] class Kitty {}
```

例えば、これは Kitty クラスの宣言に KittyAttribute 属性を用いて情報を追加しています
追加する情報の形は、属性クラスを定義するあなたの手に委ねられています
このように、型やメンバに属性を指定することを**アタッチする**といいます

属性で追加した宣言情報は、実行時に取り出すことが可能です
型の属性に問い合わせるには、リフレクション API を用います
属性も型情報の一つなので、リフレクション で問い合わせるという考えは理解できるでしょう

型の属性を取得するには MemberInfo クラスの **GetCustomAttributes()** を使います
このメソッドは、型が保有する属性のインスタンスを生成して返します

public abstract object[] GetCustomAttributes(bool inherit);

inghrit は、型の継承チェーンをさかのぼって、基底クラスの属性も検索するかどうかを指定します
true を指定すれば、メソッドは基底クラスの属性も検索して属性を返します
Type クラスや MethodInfo、FieldInfo などは全て MemberInfo の派生クラスです
各クラスは GetCustomAttributes() を再定義しています

GetCustomAttributes() が返した object 型の配列にクラスの各種属性が含まれます
これを、as 演算子などで目的の型に変換すれば、後はそのクラスの仕様にあわせ
その属性がもっている、型情報を引き出すことができます

```
using System;

enum KittyName { RENA , YUKI , MIMI }

class KittyAttribute : Attribute {
 public readonly KittyName name;
 public KittyAttribute(KittyName name) {
  this.name = name;
 }
}
```

```
[KittyAttribute(KittyName.RENA)] class Kitty {}

class Test {
 static void Main(string[] args) {
  Type t = typeof(Kitty);
  foreach(Object tmp in t.GetCustomAttributes(false)) {
   KittyAttribute attrKitty = tmp as KittyAttribute;
   if (attrKitty != null)
    Console.WriteLine("名前 : " + attrKitty.name);
  }
 }
}
```

これが、先ほどの属性クラスを用いたプログラムの完成版です
クラス Kitty は KittyAttribute 属性を宣言情報として保有しています
Main() メソッドでは typeof 演算子で Kitty クラスの型情報を取得し
GetCustomAttributes() メソッドを用いて Kitty クラスの属性情報を引き出しています

属性名のショートカット

属性クラスの名前は、最後に Attribute と付けるのが一般的です
これは言語規定では定められていませんが、.NET の暗黙のルールです

C# では、属性クラスのコンストラクタを指定するとき
クラス名の **Attributeを省略** することができます
例えば、次のプログラムは有効です

class XAttribute : System.Attribute {...
[X] class C {...

クラスに属性をアタッチするとき、属性クラスの Attribute を省略していますね
このとき、C# コンパイラは最初に X という名前の属性クラスを探します
そして、X という名前の属性クラスが見つからなければ
コンパイラは名前の末尾に Attribute という文字列を付加して再度検索するのです

```
using System;

enum KittyName { RENA , YUKI , MIMI }

class KittyAttribute : Attribute {
 public readonly KittyName name;
 public KittyAttribute(KittyName name) {
  this.name = name;
 }
}

[Kitty(KittyName.RENA)] class Kitty {}

class Test {
 static void Main(string[] args) {
  Type t = typeof(Kitty);
  foreach(Object tmp in t.GetCustomAttributes(false)) {
   KittyAttribute attrKitty = tmp as KittyAttribute;
   if (attrKitty != null)
    Console.WriteLine("名前 : " + attrKitty.name);
  }
 }
}
```

Kitty という名前の属性クラスは存在しませんし Kitty クラスと名前が衝突しています
しかし、このプログラムは正常にコンパイルすることができるでしょう
Kitty という名前の属性クラスが見つからなければ、コンパイラは KittyAttribute を探します

言語規定で属性は Attribute という名前を末尾につけなければならないわけではないので
当然、属性に Attribute が付いていないからといってショートカットとは限りません
Attribute が付いていない状態でも、その名前の属性クラスと一致した場合
コンパイラはフルネームで一致する属性クラスを優先して選択します

```
using System;

class KittyAttribute : Attribute {}
class Kitty : Attribute {}

[Kitty()] class Kitty1 {}
[KittyAttribute()] class Kitty2 {}

class Test {
 static void Main(string[] args) {
  CheckAttribute(typeof(Kitty1));
  CheckAttribute(typeof(Kitty2));
 }
 static void CheckAttribute(Type t) {
  foreach(Object tmp in t.GetCustomAttributes(false)) {
   if (tmp is KittyAttribute)
    Console.WriteLine(t + " have KittyAttribute");
   else if (tmp is Kitty)
    Console.WriteLine(t + " have Kitty");
  }
 }
}
```

Kitty という属性と KittyAttribute という属性があります
この場合 Kitty() と指定すると Kitty 属性クラスをあらわします
プログラムを実行した結果、次のようになりました

Kitty1 have Kitty
Kitty2 have KittyAttribute

Kitty 属性をアタッチしたとき、Kitty 属性が検索されたため
ショートカット機能は適応されなかったことがわかります

メンバ属性

属性は、クラスだけではなく、メンバに付けることも可能です
この場合も、同様にリフレクション API から属性を得ることができます
属性の指定方法は、クラスの場合と同様です

クラスのメンバの MemberInfo クラスの取得方法は、前章で紹介しましたね
ただし、以下のサンプルでは Type クラスの **GetMethod()** メソッドを使います

public MethodInfo GetMethod(string name);

name には、この型が保有する公開メソッドの名前を指定します
メソッドが成功すれば MethodInfo 型の参照が、失敗すれば null が返ります

```
using System;

enum KittyName { RENA , YUKI , MIMI }

class KittyAttribute : Attribute {
 public readonly KittyName name;
 public KittyAttribute(KittyName name) {
  this.name = name;
 }
}

class Kitty {
 [Kitty(KittyName.RENA)] public void KittyMethod() {}
}

class Test {
 public static void Main() {
  Type t = typeof(Kitty);
  System.Reflection.MethodInfo mi =
   t.GetMethod("KittyMethod");

  foreach(Object tmp in mi.GetCustomAttributes(false)) {
   KittyAttribute attrKitty = tmp as KittyAttribute;
   if (attrKitty != null)
    Console.WriteLine("名前 : " + attrKitty.name);
  }
 }
}
```

このプログラムの Kitty クラスの KittyMethod() メソッドは
KittyAttribute 属性を持っています
Main() メソッドでは Type クラスの GetMethod() メソッドで MethodInfo を取得し
ここから、メソッドが保有する KittyAttribute のインスタンスを取得しています

この原理は、プロパティやフィールドなどでも同様です
そのメンバに対する何らかの追加情報を、属性を使って明示できるのです

前のページへ　戻る　次のページへ