列挙型

順列の定数

コンポーネントの開発では、頻繁に定数を利用する必要があります
定数はコンポーネントの利用者が、コンポーネントの動作を「選択」する手段に用いられます

例えば、Win32 API の MessageBox() 関数は、ダイアログに表示するボタンを定数で受け取ります
MB_OK ならば OK ボタンのダイアログを表示するというような仕様になっています
このように、動作が決まっているならその動作を定数で通知するという方法があります

この場合、定数の内容よりも定数自体の興味があります
さらに、選択するべき動作が大量にある場合、定数を宣言するだけでも大変です
そこで、このような定数の列挙が必要な場合は**列挙型**を用います

列挙型は、基本的に C 言語の列挙と同じ概念です
名前付きの定数を持つ独立した型で、クラスと同じ場所で宣言することができます
列挙形の宣言は **enum** キーワードで行います

[attributes] [modifiers] enum identifier [:base-type] {enumerator-list};

attributes は属性、modifiers は列挙型の修飾子を指定します
identifier は列挙型の識別子、base-type は列挙の基底
enumerator-list は列挙の本文を指定します

列挙の本文とは**列挙メンバ宣言子群**です
列挙メンバは常に属性群と識別子、及びそれの初期化子で構成されます
列挙メンバはデフォルトで int 型の 0 からの昇順の定数となります

列挙のメンバ型は base-type で指定します
base-type は必ず数値型でなければならず char は指定できません
また、修飾子に abstract と sealed を指定することもできません
列挙は抽象化することはできず、派生も許されません

```
enum Kitty : uint { RENA , YUKI , MIMI }

class Test {
 static void Main() {
  GetKitty(Kitty.RENA);
  GetKitty(Kitty.YUKI);
  GetKitty((Kitty)2);
 }
 static void GetKitty(Kitty em) {
  switch(em) {
  case Kitty.RENA:
   System.Console.WriteLine("This is RENA");
   break;
  case Kitty.YUKI:
   System.Console.WriteLine("This is YUKI");
   break;
  case Kitty.MIMI:
   System.Console.WriteLine("This is MIMI");
   break;
  }
 }
}
```

このプログラムには、いくつか注目してほしい重要なポイントがあります
まず一つは、C# の列挙型は独立した「型」であり、C のように曖昧ではありません

Kitty は uint を基底とする列挙型です
GetKitty() メソッドは、Kitty 型の値を受け取りそれを評価します
そして、その評価に応じて適切な処理を行うメソッドをエミュレートしています

列挙型は、定数値を明示的に指定しない場合は**暗黙的に定められ**
最初のメンバを 0 として、その後のメンバは前のメンバに 1 加算した値となります
それぞれのメンバは列挙型として独立していますが、内部的な数値は uint 型です
面白いことに GetKitty((Kitty)2) は Kitty.MIMI に等しい結果となるのです
最初のメンバ RENA = 0 であり、続いて YUKI = 1、MIMI = 2 というように割り当てられているからです

ここで問題なのが、**列挙型の正体**です
C# は他言語のような単純なビット単位の型は存在しません
つまり、あらゆる型は Object を継承したクラス（参照）または構造体（値）なのです

実は、**System.Enum** クラスを基底とした型を「列挙型」と呼びます
列挙型の全てのメンバはこのクラスを基底としています
さらに、巧みな .NET の構造が列挙と数値型の相互変換を可能にしています
先ほどのプログラムでは、数値を列挙に変換するという特殊なキャストが行えました

単純に列挙型を式中に含めれば、仕様上 ToString() が呼び出されます
列挙型の ToString() は単純にそのメンバの名前を返します
列挙型の内部的数値を見たい場合は、その数値型にキャストします

```
enum Kitty : int { RENA , YUKI , MIMI }
```

```
class Test {
 static void Main() {
  System.Console.WriteLine("RENA = " + (int)Kitty.RENA);
  System.Console.WriteLine("YUKI = " + (int)Kitty.YUKI);
  System.Console.WriteLine("MIMI = " + (int)Kitty.MIMI);
 }
}
```

このプログラムを実行すると、次のような結果になります

RENA = 0
YUKI = 1
MIMI = 2

先ほど説明したように昇順に数値が割り当てられていることを確認できます
さらに、列挙は明示的に数値を指定することができます
ある地点で定数を指定した場合、それ以降のメンバは指定した定数から加算されます

```
enum Kitty : int { RENA , YUKI = 10 , MIMI }

class Test {
 static void Main() {
  System.Console.WriteLine("RENA = " + (int)Kitty.RENA);
  System.Console.WriteLine("YUKI = " + (int)Kitty.YUKI);
  System.Console.WriteLine("MIMI = " + (int)Kitty.MIMI);
 }
}
```

このプログラムでは YUKI を明示的に 10 という定数にしています
これによって、YUKI の値に暗黙的に依存する MIMI の値も変更されます
これを実行すると、次のような結果になりました

RENA = 0
YUKI = 10
MIMI = 11

MIMI はその前のメンバ YUKI の値に 1 加算した値となります
YUKI の値を明示的に指定したことによって MIMI の値も変化したのです
当然、MIMI 以降にメンバを指定していれば 12 , 13 ... と列挙されることになります

列挙の定数式は他の列挙メンバに依存することも可能です
つまり、ある地点のメンバは他のメンバの値に等しいとすることができます

```
enum Kitty : uint { RENA , YUKI , MIMI , IWAO = YUKI }

class Test {
 static void Main() {
  System.Console.WriteLine("RENA = " + (int)Kitty.RENA);
  System.Console.WriteLine("YUKI = " + (int)Kitty.YUKI);
  System.Console.WriteLine("MIMI = " + (int)Kitty.MIMI);
  System.Console.WriteLine("岩男潤子 = " + (int)Kitty.IWAO);
 }
}
```

Kitty のメンバ IWAO は同一の列挙メンバ YUKI に等しいです
ただし、これは互いに依存する再帰的な関係が生まれた場合はエラーとなります
互いに影響し合う定義を**循環定義**と呼びます

```
enum Kitty : uint { TARUTO = TAI , TAI }

class Test {
 static void Main() {
  System.Console.WriteLine("TARUTO = " + (int)Kitty.TARUTO);
  System.Console.WriteLine("TAI = " + (int)Kitty.TAI);
 }
}
```

列挙メンバ TARUTO は明示的に TAI に依存し
かつ TAI は暗黙的に TARUTO に依存する循環定義の関係があります
そのため、このプログラムはコンパイルすることができません

前のページへ　戻る　次のページへ