CHAPTER 15

Enumerated Types and Bit Flags

In this chapter:

In this chapter, I'll discuss enumerated types and bit flags. Because Windows and many programming languages have used these constructs for so many years, I'm sure that many of you are already familiar with how to use enumerated types and bit flags. However, the common language runtime (CLR) and the Framework Class Library (FCL) work together to make enumerated types and bit flags real object-oriented types that offer cool features that I suspect most developers aren't familiar with. It's amazing to me how these features, which are the focus of this chapter, make developing application code so much easier.

Enumerated Types

An *enumerated type* is a type that defines a set of symbolic name and value pairs. For example, the `Color` type shown here defines a set of symbols, with each symbol identifying a single color.

```
internal enum Color {
   White,       // Assigned a value of 0
   Red,         // Assigned a value of 1
   Green,       // Assigned a value of 2
   Blue,        // Assigned a value of 3
   Orange       // Assigned a value of 4
}
```

Of course, programmers can always write a program using 0 to represent white, 1 to represent red, and so on. However, programmers shouldn't hard-code numbers into their code and should use an enumerated type instead, for at least two reasons:

- Enumerated types make the program much easier to write, read, and maintain. With enumerated types, the symbolic name is used throughout the code, and the programmer doesn't have to mentally map the meaning of each hard-coded value (for example, white is 0 or vice versa). Also, should a symbol's numeric value change, the code can simply be recompiled without requiring any changes to the source code. In addition, documentation tools and other utilities, such as a debugger, can show meaningful symbolic names to the programmer.

- Enumerated types are strongly typed. For example, the compiler will report an error if I attempt to pass `Color.Orange` as a value to a method requiring a `Fruit` enumerated type as a parameter.

In the Microsoft .NET Framework, enumerated types are more than just symbols that the compiler cares about. Enumerated types are treated as first-class citizens in the type system, which allows for very powerful operations that simply can't be done with enumerated types in other environments (such as in unmanaged C++, for example).

Every enumerated type is derived directly from `System.Enum`, which is derived from `System.ValueType`, which in turn is derived from `System.Object`. So enumerated types are value types (described in Chapter 5, "Primitive, Reference, and Value Types") and can be represented in unboxed and boxed forms. However, unlike other value types, an enumerated type can't define any methods, properties, or events. However, you can use C#'s *extension methods* feature to simulate adding methods to an enumerated type. See the "Adding Methods to Enumerated Types" section at the end of this chapter for an example of this.

When an enumerated type is compiled, the C# compiler turns each symbol into a constant field of the type. For example, the compiler treats the `Color` enumeration shown earlier as if you had written code similar to the following.

```
internal struct Color : System.Enum {
   // Following are public constants defining Color's symbols and values
   public const Color White  = (Color) 0;
   public const Color Red    = (Color) 1;
   public const Color Green  = (Color) 2;
   public const Color Blue   = (Color) 3;
   public const Color Orange = (Color) 4;

   // Following is a public instance field containing a Color variable's value
   // You cannot write code that references this instance field directly
   public Int32 value__;
}
```

The C# compiler won't actually compile this code because it forbids you from defining a type derived from the special `System.Enum` type. However, this pseudo-type definition shows you what's happening internally. Basically, an enumerated type is just a structure with a bunch of constant fields defined in it and one instance field. The constant fields are emitted to the assembly's metadata and can be accessed via reflection. This means that you can get all of the symbols and their values associated with an enumerated type at run time. It also means that you can convert a string symbol into its equivalent numeric value. These operations are made available to you by the `System.Enum` base type, which offers several static and instance methods that can be performed on an instance of an enumerated type, saving you the trouble of having to use reflection. I'll discuss some of these operations next.

Important Symbols defined by an enumerated type are constant values. So when a compiler sees code that references an enumerated type's symbol, the compiler substitutes the symbol's numeric value at compile time, and this code no longer references the enumerated type that defined the symbol. This means that the assembly that defines the enumerated type may not be required at run time; it was required only when compiling. If you have code that references the enumerated type—rather than just having references to symbols defined by the type—the assembly containing the enumerated type's definition will be required at run time. Some versioning issues arise because enumerated type symbols are constants instead of read-only values. I explained these issues in the "Constants" section of Chapter 7, "Constants and Fields."

For example, the `System.Enum` type has a static method called `GetUnderlyingType`, and the `System.Type` type has an instance method called `GetEnumUnderlyingType`.

```
public static Type GetUnderlyingType(Type enumType);       // Defined in System.Enum
public        Type GetEnumUnderlyingType();                // Defined in System.Type
```

These methods return the core type used to hold an enumerated type's value. Every enumerated type has an underlying type, which can be a `byte`, `sbyte`, `short`, `ushort`, `int` (the most common type and what C# chooses by default), `uint`, `long`, or `ulong`. Of course, these C# primitive types correspond to FCL types. However, to make the implementation of the compiler itself simpler, the C# compiler requires you to specify a primitive type name here; using an FCL type name (such as `Int32`) generates the following message: `error CS1008: Type byte, sbyte, short, ushort, int, uint, long, or ulong expected.` The following code shows how to declare an enumerated type with an underlying type of `byte` (`System.Byte`).

```
internal enum Color : byte {
   White,
   Red,
   Green,
   Blue,
   Orange
}
```

With the `Color` enumerated type defined in this way, the following code shows what `Get-UnderlyingType` will return.

```
// The following line displays "System.Byte".
Console.WriteLine(Enum.GetUnderlyingType(typeof(Color)));
```

The C# compiler treats enumerated types as primitive types. As such, you can use many of the familiar operators (==, !=, <, >, <=, >=, +, −, ^, &, |, ~, ++, and --) to manipulate enumerated type instances. All of these operators actually work on the `value__` instance field inside each enumerated type instance. Furthermore, the C# compiler allows you to explicitly cast instances of an enumerated type to a different enumerated type. You can also explicitly cast an enumerated type instance to a numeric type.

Given an instance of an enumerated type, it's possible to map that value to one of several string representations by calling the `ToString` method inherited from `System.Enum`.

```
Color c = Color.Blue;
Console.WriteLine(c);               // "Blue" (General format)
Console.WriteLine(c.ToString());    // "Blue" (General format)
Console.WriteLine(c.ToString("G")); // "Blue" (General format)
Console.WriteLine(c.ToString("D")); // "3"    (Decimal format)
Console.WriteLine(c.ToString("X")); // "03"   (Hex format)
```

Note When using hex formatting, `ToString` always outputs uppercase letters. In addition, the number of digits output depends on the enum's underlying type: 2 digits for `byte/sbyte`, 4 digits for `short/ushort`, 8 digits for `int/uint`, and 16 digits for `long/ulong`. Leading zeros are output if necessary.

In addition to the `ToString` method, the `System.Enum` type offers a static `Format` method that you can call to format an enumerated type's value.

```
public static String Format(Type enumType, Object value, String format);
```

Generally, I prefer to call the `ToString` method because it requires less code and it's easier to call. But using `Format` has one advantage over `ToString`: `Format` lets you pass a numeric value for the value parameter; you don't have to have an instance of the enumerated type. For example, the following code will display "Blue".

```
// The following line displays "Blue".
Console.WriteLine(Enum.Format(typeof(Color), (Byte)3, "G"));
```

Note It's possible to declare an enumerated type that has multiple symbols, all with the same numeric value. When converting a numeric value to a symbol by using general formatting, `Enum`'s methods return one of the symbols. However, there's no guarantee of which symbol name is returned. Also, if no symbol is defined for the numeric value you're looking up, a string containing the numeric value is returned.

It's also possible to call `System.Enum`'s static `GetValues` method or `System.Type`'s instance `GetEnumValues` method to obtain an array that contains one element for each symbolic name in an enumerated type; each element contains the symbolic name's numeric value.

```
public static Array GetValues(Type enumType);      // Defined in System.Enum
public        Array GetEnumValues();               // Defined in System.Type
```

Using this method along with the ToString method, you can display all of an enumerated type's symbolic and numeric values, like the following.

```
Color[] colors = (Color[]) Enum.GetValues(typeof(Color));
Console.WriteLine("Number of symbols defined: " + colors.Length);
Console.WriteLine("Value\tSymbol\n-----\t------");
foreach (Color c in colors) {
   // Display each symbol in Decimal and General format.
   Console.WriteLine("{0,5:D}\t{0:G}", c);
}
```

The previous code produces the following output.

```
Number of symbols defined: 5
Value   Symbol
-----   ------
    0   White
    1   Red
    2   Green
    3   Blue
    4   Orange
```

Personally, I don't like the GetValues and GetEnumValues methods because they both return an Array, which I then have to cast to the appropriate array type. So, I always define my own method.

```
public static TEnum[] GetEnumValues<TEnum>() where TEnum : struct {
   return (TEnum[])Enum.GetValues(typeof(TEnum));
}
```

With my generic GetEnumValues method, I can get better compile-time type-safety and simplify the first line of code in the previous example to the following.

```
Color[] colors = GetEnumValues<Color>();
```

This discussion shows some of the cool operations that can be performed on enumerated types. I suspect that the ToString method with the general format will be used quite frequently to show symbolic names in a program's user interface elements (list boxes, combo boxes, and the like), as long as the strings don't need to be localized (because enumerated types offer no support for localization). In addition to the GetValues method, the System.Enum type and the System.Type type also offer the following methods that return an enumerated type's symbols.

```
// Returns a String representation for the numeric value
public static String GetName(Type enumType, Object value); // Defined in System.Enum
public        String GetEnumName(Object value);            // Defined in System.Type


// Returns an array of Strings: one per symbol defined in the enum
public static String[] GetNames(Type enumType);            // Defined in System.Enum
public        String[] GetEnumNames();                     // Defined in System.Type
```

I've discussed a lot of methods that you can use to look up an enumerated type's symbol. But you also need a method that can look up a symbol's equivalent value, an operation that could be used to convert a symbol that a user enters into a text box, for example. Converting a symbol to an instance of an enumerated type is easily accomplished by using one of `Enum`'s static `Parse` and `TryParse` methods.

```
public static Object Parse(Type enumType, String value);
public static Object Parse(Type enumType, String value, Boolean ignoreCase);
public static Boolean TryParse<TEnum>(String value, out TEnum result) where TEnum: struct;
public static Boolean TryParse<TEnum>(String value, Boolean ignoreCase, out TEnum result)
   where TEnum : struct;
```

Here's some code demonstrating how to use this method.

```
// Because Orange is defined as 4, 'c' is initialized to 4.
Color c = (Color) Enum.Parse(typeof(Color), "orange", true);

// Because Brown isn't defined, an ArgumentException is thrown.
c = (Color) Enum.Parse(typeof(Color), "Brown", false);

// Creates an instance of the Color enum with a value of 1
Enum.TryParse<Color>("1", false, out c);

// Creates an instance of the Color enum with a value of 23
Enum.TryParse<Color>("23", false, out c);
```

Finally, using the following `Enum`'s static `IsDefined` method and `Type`'s `IsEnumDefined` method:

```
public static Boolean IsDefined(Type enumType, Object value);     // Defined in System.Enum
public        Boolean IsEnumDefined(Object value);                // Defined in System.Type
```

you can determine whether a numeric value is legal for an enumerated type.

```
// Displays "True" because Color defines Red as 1
Console.WriteLine(Enum.IsDefined(typeof(Color),  (Byte)1));

// Displays "True" because Color defines White as 0
Console.WriteLine(Enum.IsDefined(typeof(Color), "White"));

// Displays "False" because a case-sensitive check is performed
Console.WriteLine(Enum.IsDefined(typeof(Color), "white"));

// Displays "False" because Color doesn't have a symbol of value 10
Console.WriteLine(Enum.IsDefined(typeof(Color),  (Byte)10));
```

The `IsDefined` method is frequently used for parameter validation. Here's an example.

```
public void SetColor(Color c) {
   if (!Enum.IsDefined(typeof(Color), c)) {
      throw(new ArgumentOutOfRangeException("c", c, "Invalid Color value."));
   }
   // Set color to White, Red, Green, Blue, or Orange
   ...
}
```

The parameter validation is useful because someone could call SetColor like the following.

```
SetColor((Color) 547);
```

Because no symbol has a corresponding value of 547, the SetColor method will throw an ArgumentOutOfRangeException exception, indicating which parameter is invalid and why.

Important The IsDefined method is very convenient, but you must use it with caution. First, IsDefined always does a case-sensitive search, and there is no way to get it to perform a case-insensitive search. Second, IsDefined is pretty slow because it uses reflection internally; if you wrote code to manually check each possible value, your application's performance would most certainly be better. Third, you should really use IsDefined only if the enum type itself is defined in the same assembly that is calling IsDefined. Here's why: Let's say the Color enum is defined in one assembly and the SetColor method is defined in another assembly. The SetColor method calls IsDefined, and if the color is White, Red, Green, Blue, or Orange, SetColor performs its work. However, if the Color enum changes in the future to include Purple, SetColor will now allow Purple, which it never expected before, and the method might execute with unpredictable results.

Finally, the System.Enum type offers a set of static ToObject methods that convert an instance of a Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, or UInt64 to an instance of an enumerated type.

Enumerated types are always used in conjunction with some other type. Typically, they're used for the type's method parameters or return type, properties, and fields. A common question that arises is whether to define the enumerated type nested within the type that requires it or to define the enumerated type at the same level as the type that requires it. If you examine the FCL, you'll see that an enumerated type is usually defined at the same level as the class that requires it. The reason is simply to make the developer's life a little easier by reducing the amount of typing required. So you should define your enumerated type at the same level unless you're concerned about name conflicts.

Bit Flags

Programmers frequently work with sets of bit flags. When you call the System.IO.File type's GetAttributes method, it returns an instance of a FileAttributes type. A FileAttributes type is an instance of an Int32-based enumerated type, in which each bit reflects a single attribute of the file. The FileAttributes type is defined in the FCL as follows.

```
[Flags, Serializable]
public enum FileAttributes {
   ReadOnly            = 0x00001,
   Hidden              = 0x00002,
   System              = 0x00004,
   Directory           = 0x00010,
   Archive             = 0x00020,
   Device            = 0x00040,
   Normal            = 0x00080,
   Temporary         = 0x00100,
   SparseFile        = 0x00200,
   ReparsePoint      = 0x00400,
   Compressed        = 0x00800,
   Offline           = 0x01000,
   NotContentIndexed = 0x02000,
   Encrypted         = 0x04000,
   IntegrityStream   = 0x08000,
   NoScrubData       = 0x20000
}
```

To determine whether a file is hidden, you would execute code like the following.

```
String file = Assembly.GetEntryAssembly().Location;
FileAttributes attributes = File.GetAttributes(file);
Console.WriteLine("Is {0} hidden? {1}", file, (attributes & FileAttributes.Hidden) != 0);
```

Note The `Enum` class defines a `HasFlag` method defined as follows.

```
public Boolean HasFlag(Enum flag);
```

Using this method, you could rewrite the call to `Console.WriteLine` like the following.

```
Console.WriteLine("Is {0} hidden? {1}", file,
   attributes.HasFlag(FileAttributes.Hidden));
```

However, I recommend that you avoid the `HasFlag` method for this reason: Because it takes a parameter of type `Enum`, any value you pass to it must be boxed, requiring a memory allocation.

And here's code demonstrating how to change a file's attributes to read-only and hidden.

```
File.SetAttributes(file, FileAttributes.ReadOnly | FileAttributes.Hidden);
```

As the `FileAttributes` type shows, it's common to use enumerated types to express the set of bit flags that can be combined. However, although enumerated types and bit flags are similar, they don't have exactly the same semantics. For example, enumerated types represent single numeric values, and bit flags represent a set of bits, some of which are on, and some of which are off.

When defining an enumerated type that is to be used to identify bit flags, you should, of course, explicitly assign a numeric value to each symbol. Usually, each symbol will have an individual bit turned on. It is also common to see a symbol called `None` defined with a value of 0, and you can also define symbols that represent commonly used combinations (see the following `ReadWrite` symbol). It's also highly recommended that you apply the `System.FlagsAttribute` custom attribute type to the enumerated type, as shown here:

```
[Flags]    // The C# compiler allows either "Flags" or "FlagsAttribute".
internal enum Actions {
   None      = 0,
   Read      = 0x0001,
   Write     = 0x0002,
   ReadWrite = Actions.Read | Actions.Write,
   Delete    = 0x0004,
   Query     = 0x0008,
   Sync      = 0x0010
}
```

Because Actions is an enumerated type, you can use all of the methods described in the previous section when working with bit-flag enumerated types. However, it would be nice if some of those functions behaved a little differently. For example, let's say you had the following code.

```
Actions actions = Actions.Read | Actions.Delete; // 0x0005
Console.WriteLine(actions.ToString());           // "Read, Delete"
```

When ToString is called, it attempts to translate the numeric value into its symbolic equivalent. The numeric value is 0x0005, which has no symbolic equivalent. However, the ToString method detects the existence of the [Flags] attribute on the Actions type, and ToString now treats the numeric value not as a single value but as a set of bit flags. Because the 0x0001 and 0x0004 bits are set, ToString generates the following string: "Read, Delete". If you remove the [Flags] attribute from the Actions type, ToString would return "5".

I discussed the ToString method in the previous section, and I showed that it offered three ways to format the output: "G" (general), "D" (decimal), and "X" (hex). When you're formatting an instance of an enumerated type by using the general format, the type is first checked to see if the [Flags] attribute is applied to it. If this attribute is not applied, a symbol matching the numeric value is looked up and returned. If the [Flags] attribute is applied, ToString works like this:

1. The set of numeric values defined by the enumerated type is obtained, and the numbers are sorted in descending order.
2. Each numeric value is bitwise-ANDed with the value in the enum instance, and if the result equals the numeric value, the string associated with the numeric value is appended to the output string, and the bits are considered accounted for and are turned off. This step is repeated until all numeric values have been checked or until the enum instance has all of its bits turned off.
3. If, after all the numeric values have been checked, the enum instance is still not 0, the enum instance has some bits turned on that do not correspond to any defined symbols. In this case, ToString returns the original number in the enum instance as a string.
4. If the enum instance's original value wasn't 0, the string with the comma-separated set of symbols is returned.
5. If the enum instance's original value was 0 and if the enumerated type has a symbol defined with a corresponding value of 0, the symbol is returned.
6. If we reach this step, "0" is returned.

If you prefer, you could define the `Actions` type without the `[Flags]` attribute and still get the correct string by using the "F" format.

```
// [Flags]    // Commented out now
internal enum Actions {
   None      = 0
   Read      = 0x0001,
   Write     = 0x0002,
   ReadWrite = Actions.Read | Actions.Write,
   Delete    = 0x0004,
   Query     = 0x0008,
   Sync      = 0x0010
}

Actions actions = Actions.Read | Actions.Delete; // 0x0005
Console.WriteLine(actions.ToString("F"));        // "Read, Delete"
```

If the numeric value has a bit that cannot be mapped to a symbol, the returned string will contain just a decimal number indicating the original numeric value; no symbols will appear in the string.

Note that the symbols you define in your enumerated type don't have to be pure powers of 2. For example, the `Actions` type could define a symbol called `All` with a value of 0x001F. If an instance of the `Actions` type has a value of 0x001F, formatting the instance will produce a string that contains "All". The other symbol strings won't appear.

So far, I've discussed how to convert numeric values into a string of flags. It's also possible to convert a string of comma-delimited symbols into a numeric value by calling `Enum`'s static `Parse` and `TryParse` method. Here's some code demonstrating how to use this method.

```
// Because Query is defined as 8, 'a' is initialized to 8.
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query", true);
Console.WriteLine(a.ToString());  // "Query"

// Because Query and Read are defined, 'a' is initialized to 9.
Enum.TryParse<Actions>("Query, Read", false, out a);
Console.WriteLine(a.ToString());  // "Read, Query"

// Creates an instance of the Actions enum with a value of 28
a = (Actions) Enum.Parse(typeof(Actions), "28", false);
Console.WriteLine(a.ToString());  // "Delete, Query, Sync"
```

When `Parse` and `TryParse` are called, the following actions are performed internally:

1. It removes all whitespace characters from the start and end of the string.
2. If the first character of the string is a digit, plus sign (+), or minus sign (-), the string is assumed to be a number, and an enum instance is returned whose numeric value is equal to the string converted to its numeric equivalent.
3. The passed string is split into a set of tokens (separated by commas), and all white space is trimmed away from each token.

4. Each token string is looked up in the enum type's defined symbols. If the symbol is not found, `Parse` throws a `System.ArgumentException` and `TryParse` returns `false`. If the symbol is found, bitwise-OR its numeric value into a running result, and then look up the next token.
5. If all tokens have been sought and found, return the running result.

You should never use the `IsDefined` method with bit flag–enumerated types. It won't work for two reasons:

- If you pass a string to `IsDefined`, it doesn't split the string into separate tokens to look up; it will attempt to look up the string as through it were one big symbol with commas in it. Because you can't define an enum with a symbol that has commas in it, the symbol will never be found.
- If you pass a numeric value to `IsDefined`, it checks whether the enumerated type defines a single symbol whose numeric value matches the passed-in number. Because this is unlikely for bit flags, `IsDefined` will usually return `false`.

Adding Methods to Enumerated Types

Earlier in this chapter, I mentioned that you cannot define a method as part of an enumerated type. And, for many years, this has saddened me because there are many occasions when I would love to have been able to supply some methods to my enumerated type. Fortunately, I can use C#'s extension method feature (discussed in Chapter 8, "Methods") to simulate adding methods to an enumerated type.

If I want to add some methods to the `FileAttributes` enumerated type, I can define a static class with extension methods as follows.

```
internal static class FileAttributesExtensionMethods {
   public static Boolean IsSet(this FileAttributes flags, FileAttributes flagToTest) {
      if (flagToTest == 0)
         throw new ArgumentOutOfRangeException("flagToTest", "Value must not be 0");
      return (flags & flagToTest) == flagToTest;
   }

   public static Boolean IsClear(this FileAttributes flags, FileAttributes flagToTest) {
      if (flagToTest == 0)
         throw new ArgumentOutOfRangeException("flagToTest", "Value must not be 0");
      return !IsSet(flags, flagToTest);
   }

   public static Boolean AnyFlagsSet(this FileAttributes flags, FileAttributes testFlags) {
      return ((flags & testFlags) != 0);
   }

   public static FileAttributes Set(this FileAttributes flags, FileAttributes setFlags) {
      return flags | setFlags;
   }
   public static FileAttributes Clear(this FileAttributes flags,
      FileAttributes clearFlags) {
      return flags & ~clearFlags;
   }

   public static void ForEach(this FileAttributes flags,
      Action<FileAttributes> processFlag) {
      if (processFlag == null) throw new ArgumentNullException("processFlag");
      for (UInt32 bit = 1; bit != 0; bit <<= 1) {
         UInt32 temp = ((UInt32)flags) & bit;
         if (temp != 0) processFlag((FileAttributes)temp);
      }
   }
}
```

And here is some code that demonstrates calling some of these methods. As you can see, the code looks as if I'm calling methods on the enumerated type.

```
FileAttributes fa = FileAttributes.System;
fa = fa.Set(FileAttributes.ReadOnly);
fa = fa.Clear(FileAttributes.System);
fa.ForEach(f => Console.WriteLine(f));
```