Diving in OOP (Day 6) : Understanding Enums in C# (A Practical Approach)

Introduction:

My article of the series "Diving in OOP" will explain enum datatype in c#. We'll learn by doing hands on lab and not only by theory. We'll explore the power of enum and will cover almost every scenario in which we can use enum. We'll follow a practical approach of learning to understand this concept. We may come across complex examples to understand the concept more deeply.

Enums (The definition):

Let's start with the definition taken from MSDN

*"The **enum** keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.*
Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience. However, an enum can also be nested within a class or struct.
By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example, in the following enumeration, Sat is 0, Sun is 1, Mon is 2, and so forth."

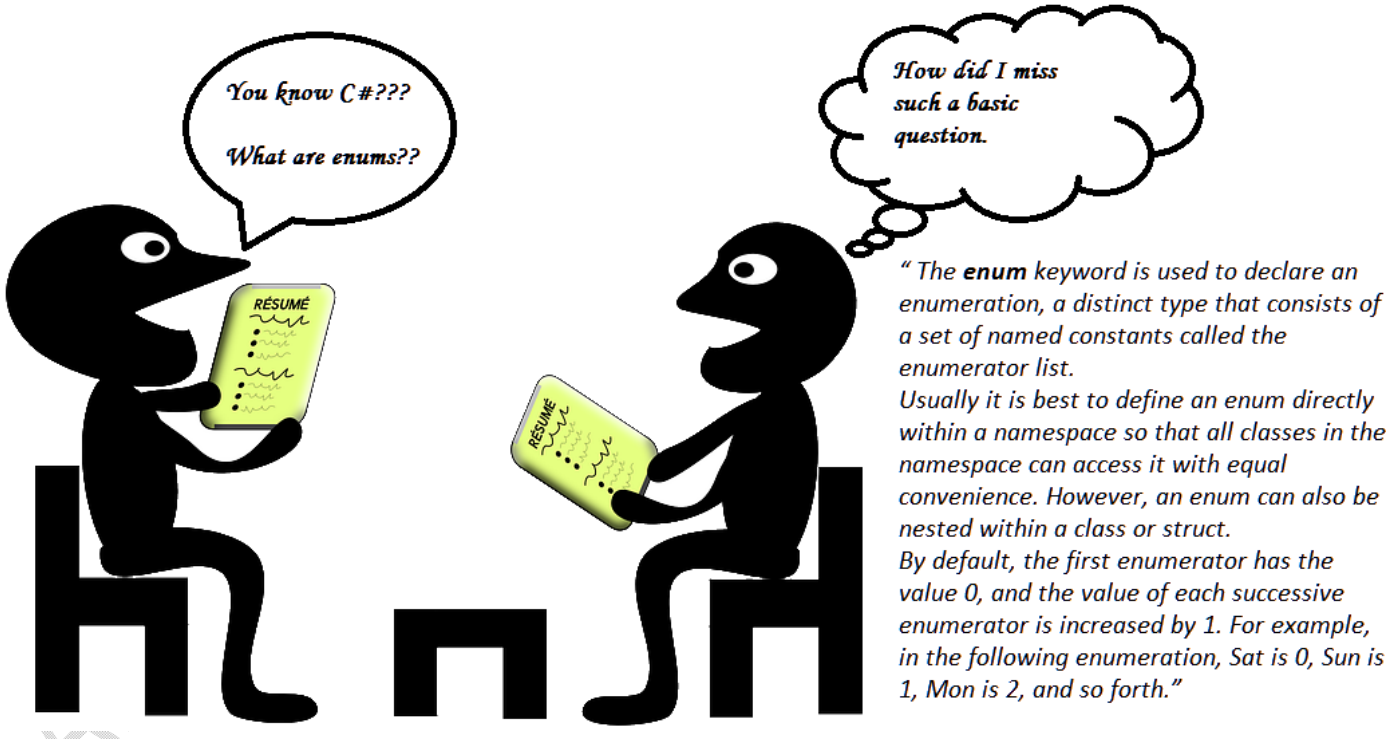

Image credit: http://pixabay.com/en/job-interview-career-conference-156130/

Pre-requisites:

I expect that my readers of this article should have very basic knowledge of C# and I always wish that my readers should enjoy while reading this article. Also keep writing programs by yourself that are given as example in this article, to get the hands on and understand the concept more deeply.

Roadmap:

Let's recall our road map,

Image credit: http://borgefagerli.com/at-a-crossroads/cross-roads/

1. Diving in OOP (Day 1): Polymorphism and Inheritance (Early Binding/Compile Time Polymorphism).
2. Diving in OOP (Day 2): Polymorphism and Inheritance (Inheritance).
3. Diving in OOP (Day 3): Polymorphism and Inheritance (Dynamic Binding/Run Time polymorphism).
4. Diving in OOP (Day 4): Polymorphism and Inheritance (All about Abstract classes in C#).
5. Diving in OOP (Day 5): All about access modifiers in C# (Public/Private/Protected/Internal/Sealed/Constants/Readonly Fields).
6. **Diving in OOP (Day 6) : Understanding Enum in C# (A Practical Approach)**

A Practical Approach:

Enum plays almost the same responsibility as the class does, i.e. creating a new data type, and it exists at the same level as class, interfaces or structs.
Just open your visual studio and add a console application named Enums. You'll get Program.cs class.

Note: Each and every code snippet in this article is tried and tested.

Declare an enum at the same level as of Program class, call it as Color.

Program.cs:

```
namespace Enums
{
   class Program
```

Akhil Mittal (codeteddy.blogspot.in)

```
    {
        static void Main(string[] args)
        {
        }
    }

    enum Color
    {
        Yellow,
        Blue,
        Brown,
        Green

    }
}
```

In the above mentioned example, we created a new datatype with the help of enum. The datatype is Color having four distinct values Yellow, Blue, Brown and Green. The text that we write inside the declared enum could be anything of your wish; it just provides a custom enumerated list to you.

Modify your main program as shown below,

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Color.Yellow);
            Console.ReadLine();
        }
    }

    enum Color
    {
        Yellow,
        Blue,
        Brown,
        Green

    }
}
```

Run the program.

Output: Yellow

Akhil Mittal (sharppulse.blogspot.in)

Now just typecast Color.Yellow to int, what do we get?

```
using System;
namespace Enums
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine((int)Color.Yellow);
         Console.ReadLine();
      }
   }

   enum Color
   {
      Yellow,
      Blue,
      Brown,
      Green

   }
}
```

Output: 0

We see that enum is called as static variables, so an enum can be considered here as static objects. Therefore other enums in the above example can be declared in the same way as Yellow, like Blue can be declared as Color.Blue. The output in above two examples we see is 0 when we type cast and Yellow without typecasting, hence we see here that its behaviour is very similar to an array where Yellow has a value 0, similarly Blue has a value 1, Brown: 2,Green:3.
Therefore when we do Color.Yellow, it's like displaying a number 0, so from this can we infer that an enum represents a constant number, therefore an enum type is a distinct type having named constants.

Point to remember: An enum represents for a constant number, and an enum type is known as a distinct type having named constants.

Underlying data type:

Program.cs:

```
using System;
namespace Enums
{
```

Akhil Mittal (csharppulse.blogspot.in)

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((byte)Color.Yellow);
            Console.WriteLine((byte)Color.Blue);
            Console.ReadLine();
        }
    }

    enum Color:byte
    {
        Yellow,
        Blue,
        Brown,
        Green

    }
}
```

Output:

0
1

Note: Each and every code snippet in this article is tried and tested.

The only change we did here is that we specified the type to the underlying enum that we declared. The default datatype for the enum is int, here we have specified the data type as byte and we get the result. There are more data types that can be specified for enum like long, ulong, short, ushort, int, uint, byte and sbyte.

Point to remember: We can't declare char as an underlying data type for enum objects because char stores Unicode characters, but enum objects data type can only be number.

Inheritance in enum:

Program.cs:

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((byte)Color.Yellow);
```

Akhil Mittal (csharppulse.blogspot.in)

```
            Console.WriteLine((byte)Color.Blue);
            Console.ReadLine();
        }
    }

    enum Color:byte
    {
        Yellow,
        Blue,
        Brown,
        Green

    }

    enum Shades:Color
    {

    }
}
```

Output:

Compile time error: Type byte, sbyte, short, ushort, int, uint, long, or ulong expected

We clearly see here enums can't be derived from any other type except that of mentioned in the error.

Point to remember: enum can't be derived from any other type except that of type byte, sbyte, short, ushort, int, uint, long, or ulong

Let's derive a class from enum, call it class **Derived**, so our code,

Program.cs:

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((byte)Color.Yellow);
            Console.WriteLine((byte)Color.Blue);
            Console.ReadLine();
        }
    }
```

Enum:

```
    enum Color:byte
    {
        Yellow,
```

Akhil Mittal (csharppulse.blogspot.in)

```
    Blue,
    Brown,
    Green

  }
```

Derived.cs:

```
class Derived:Color
  {

  }
```

Compile the code.

Output:

Compile time error: 'Enums.Derived': cannot derive from sealed type 'Enums.Color'

Image credit: https://www.flickr.com/photos/lwr/931211869/

So, **Point to remember:** By default enum is a sealed class and therefore sticks to all the rules that a sealed class follows, so no class can derive from enum i.e. a sealed type.

Can System.Enum be a base class to enum?

Program.cs:

```
using System;

namespace Enums
{
  internal enum Color: System.Enum
  {
    Yellow,
```

Akhil Mittal (sharppulse.blogspot.in)

```
        Blue
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
        }
    }
}
```

Output:

Compile time error: Type byte, sbyte, short, ushort, int, uint, long, or ulong expected

Point to remember: The enum type is implicitly derived from System.Enum and so we cannot explicitly derive it from System.Enum.

To add more, enum is also derived from three interfaces IComparable, IFormattable and IConvertible

A. IComparable :

Let's check,

Program.cs:

```
using System;

namespace Enums
{
    internal enum Color
    {
        Yellow,
        Blue,
        Green
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            Console.WriteLine(Color.Yellow.CompareTo(Color.Blue));
            Console.WriteLine(Color.Blue.CompareTo(Color.Green));
            Console.WriteLine(Color.Blue.CompareTo(Color.Yellow));
            Console.WriteLine(Color.Green.CompareTo(Color.Green));
```

Akhil Mittal (csharppulse.blogspot.in)

```
            Console.ReadLine();
        }
    }
}
```

Output:

```
-1
-1
 1
 0
```

Sometimes we may get into situations where we have large number of enums defined and we want to compare the values of enum to each other to check if they are smaller , larger or equal value to one another. Since all enum implicitly derive from Enum class that implements tha interface IComparable, so they all have a method CompareTo(), that we just used in above example. The method being non static has to be used through a member.Yellow has value 0, Blue as 1 and Green as 2.In the first statement, when Color.Yellow comared to Color.Blue, value of Yellow is smaller than Blue hence -1 returned, same applied for the second statement when Color.Blue compared to Color.Green.Green has larger value i.e. 2 than that of Color.Blue having value 1 only. In third statement i.e. vice versa of first statement, we get the result of camparison as 1, because Blue is larger than Yellow.In last statement where Color.Green compares to itself, we undoubtedly get the value 0.
So value -1 means the value is smaller, 1 means value is larger and 0 means equal values for both the enum members.

Another comparison example,
Program.cs:

```
using System;

namespace Enums
{
    enum Color
    {
        Yellow,
        Blue,
        Green
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            int myColor = 2;
            if(myColor== Color.Green)
            {
                Console.WriteLine("my color");
```

Akhil Mittal (csharppuls.blogspot.in)

```
        }
        Console.ReadLine();
      }
   }
}
```

Output:
Compile time error : Operator '==' cannot be applied to operands of type 'int' and 'Enums.Color'

In above example we tried to compare an int type to Enum type and resulted in a compile time error. Since enum acts as an individual data type so it cannot be directly compared to an int, however, we can typecast the enum type to int to perform comparison, like in below example,

Program.cs:

```
using System;

namespace Enums
{
    enum Color
   {
      Yellow,
      Blue,
      Green
   }

   internal class Program
   {
      private static void Main(string[] args)
      {
         int myColor = 2;
         if(myColor== (int)Color.Green)
         {
            Console.WriteLine("my color");
         }
        Console.ReadLine();
      }
   }
}
```

Output: my color

B. IFormattable

Program.cs:

```
using System;
```

Akhil Mittal (sharppulse.blogspot.in)

```
namespace Enums
{
   internal enum Color
   {
      Yellow,
      Blue,
      Green
   }

   internal class Program
   {
      private static void Main(string[] args)
      {
         System.Console.WriteLine(Color.Format(typeof(Color), Color.Green, "X"));
         System.Console.WriteLine(Color.Format(typeof(Color), Color.Green, "d"));
         Console.ReadLine();
      }
   }
}
```

Output:

```
00000002
2
```

Format is the method derived from IFormatter interface. It's a static method so can be used directly with the enum class defined as Color. It's first parameter is the type of the enum class, second is the member that has to be formatted and third is the format i.e. hexadecimal or decimal, like we used in above example, and we got a positive result output too.

C. IConvertible :

```
using System;

namespace Enums
{
   enum Color
   {
      Yellow,
      Blue,
      Green
   }

   internal class Program
   {
      private static void Main(string[] args)
```

Akhil Mittal (charppulse.blogspot.in)

```
    {
      string[] names;
      names = Color.GetNames(typeof (Color));
      foreach (var name in names)
      {
        Console.WriteLine(name);
      }
      Console.ReadLine();
    }
  }
}
```

Output:

```
Yellow
Blue
Green
```

Note: Each and every code snippet in this article is tried and tested.

GetNames is a static method that accepts Type i.e. instance of type as a parameter and in return gives an array of strings. Like in above example, we had array of 3 members in our enum, therefore there names is displayed one by one.

Another example:

Program.cs

```
using System;

namespace Enums
{
   enum Color
  {
    Yellow,
    Blue,
    Green
  }

  internal class Program
  {
    private static void Main(string[] args)
    {
      Console.WriteLine(Color.Blue.ToString());
      Console.WriteLine(Color.Green.ToString());
      Console.ReadLine();
    }
```

Akhil Mittal (csharppulse.blogspot.in)

```
    }
}
```

Output:

```
Blue
Green
```

As we see in above example, we converted an enum type to staring type and got an output too, so, numerous predefined conversion methods can be used to convert enum from one data type to another.

Point to remember: A numerous predefined conversion methods can be used to convert enum from one data type to another.

Duplicity, default values and initialization:

Program.cs:

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((byte)Color.Yellow);
            Console.WriteLine((byte)Color.Blue);
            Console.ReadLine();
        }
    }

    enum Color
    {
        Yellow,
        Blue,
        Brown,
        Green,
        Blue
    }
}
```

Output:

Compile time error: The type 'Enums.Color' already contains a definition for 'Blue'

Akhil Mittal (csharppulse.blogspot.in)

In the above example we just repeated the enum member Blue of Color, and we got a compile time error, hence we now know that an enum cannot contain two members having same name.By default if the first value is not specified, first member takes the value 0 and increments it by one to succeeding members.

Let's take one more example.

Program.cs:

```
using System;
namespace Enums
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine((int)Color.Yellow);
         Console.WriteLine((int)Color.Blue);
         Console.WriteLine((int)Color.Brown);
         Console.WriteLine((int)Color.Green);

         Console.ReadLine();
      }
   }

   enum Color
   {
      Yellow =2,
      Blue,
      Brown=9,
      Green,

   }
}
```

Output:

```
2
3
9
10
```

Surprised!, We can always specify the default constant value to any enum member, here we see, we specified value 2 to yellow, so as per law of enum, the value of blue will be incremented by one and gets the value 3.We again specified 9 as a default value to Brown, and so its successor Green gets incremented by one and gets that value 10.

Moving on to another example.

Akhil Mittal (csharppulse.blogspot.in)

Program.cs:

```
using System;
namespace Enums
{
   class Program
   {
      static void Main(string[] args)
      {

      }
   }

   enum Color:byte
   {
      Yellow =300 ,
      Blue,
      Brown=9,
      Green,

   }
}
```

Output:

Compile time error: Constant value '300' cannot be converted to a 'byte'

We just derived our enum from byte, we know we can do that ☺. We then changed the value of yellow from 2 to 300, and we resulted in a compile time error. Since here our underlying data type was byte , so it is as simple as that, that we can not specify the value to enum members which exceeds the range of underlying data types.The value 300 is beyond the range of byte.It is similar to assigning the beyond range value to a byte data type variable.

Another example:

Program.cs:

```
using System;
namespace Enums
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine((int)Color.Yellow);
         Console.WriteLine((int)Color.Blue);
```

Akhil Mittal (sharppulse.blogspot.in)

```
            Console.WriteLine((int)Color.Brown);
            Console.WriteLine((int)Color.Green);

            Console.ReadLine();
        }
    }

    enum Color
    {
        Yellow = 2,
        Blue,
        Brown = 9,
        Green = Yellow
    }
}
```

Output:

```
2
3
9
2
```

Here we initialized Green to Yellow, and we did not get any error, so we see, more than one enum members can be initialized a same constant value.

Point to remember: More than one enum members can be initialized a same constant value.

Program.cs:

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Color.Yellow = 3;
        }
    }

    enum Color
    {
        Yellow = 2,
        Blue,
        Brown = 9,
        Green = Yellow
    }
```

Akhil Mittal (csharppulse.blogspot.in)

}

Output:

Compile time error: The left-hand side of an assignment must be a variable, property or indexer

In the above example, we tried to initialize the enum member out of the scope of defined enum i.e. in another class, and got a compile time error.We must not forget that an enum acts as a constant, which cannot change its value.

Point to remember: An enum acts as a constant, so its value cannot be changed once initialized.

Readability:

Program.cs:

```
using System;

namespace Enums
{
   internal enum Color
   {
      Yellow,
      Blue,
      Brown,
      Green
   }

   internal class Program
   {
      private static void Main(string[] args)
      {
         Console.WriteLine(CheckColor(Color.Yellow));
         Console.WriteLine(CheckColor(Color.Brown));
         Console.WriteLine(CheckColor(Color.Green));
         Console.ReadLine();
      }

      public static string CheckColor(Color color)
      {
         switch (color)
         {
            case Color.Yellow:
               return "Yellow";
            case Color.Blue:
               return "Blue";
```

Akhil Mittal (csharppulse.blogspot.in)

```
            case Color.Brown:
               return "Brown";
            case Color.Green:
               return "Green";
            default:
               return "no color";

         }
      }
   }
}
```

Output:

```
Yellow
Brown
Green
```

Here, in above example we have declared an enum Color containing various color members.There is a class named program that contains a static method named CheckColor, that has a switch statement checking color on the basis of passed parameter to the method i.e. Enum Color.In Main method we try to access that CheckColor method, passing various parameters. We see that the the switch statement in CheckColor method can take any of the datatype passed and in return case statements use name of that type and not the plain int number to compare the result.e see that this made our program more readable. So enum plays an important role in making the [rogram more readable and structured, easy to grasp.

Circular dependency:

Program.cs:

```
using System;

namespace Enums
{
   internal enum Color
   {
      Yellow=Blue,
      Blue
   }

   internal class Program
   {
      private static void Main(string[] args)
      {
      }
   }
```

Akhil Mittal (csharpcorner.blogspot.in)

}

Output:

Compile time error: The evaluation of the constant value for 'Enums.Color.Yellow' involves a circular definition

Like constants we also cannot have circular dependency in enums. We assigned value Blue to Yellow, and Blue in turn is incremented by one as a next enum member, this results in a circular dependency of Blue to yellow, and resulted in error, c# is smart enough to catch these kind of tricks.

Diving Deep:

Let's take some complex scenarios,
Lab1:

Program.cs:

```
using System;

namespace Enums
{
    enum Color
   {

   }

   internal class Program
   {
      private static void Main(string[] args)
      {
         Color color = (Color) -1;
        Console.ReadLine();
      }
   }
}
```

Note: Each and every code snippet in this article is tried and tested.

Output:
Compile time error:
To cast a negative value, you must enclose the value in parentheses
'Enums.Color' is a 'type' but is used like a 'variable'

In the above example, we are casting a negative value to enum, but the compiler says that while casting a negative value, we must keep that in parenthesis. It's not strange, as C# knows that "-" is also a unary

Akhil Mittal (sharppulse.blogspot.in)

operator, that while using above code may create a confusion for compiler that we are using subtraction or typecasting a negative value. So always use parenthesis while typecasting negative values.

Lab2:

Program.cs:

```
using System;

namespace Enums
{
    enum Color
   {
     value__
   }

   internal class Program
   {
      private static void Main(string[] args)
      {

      }
   }
}
```

Output:

Compile time error: The enumerator name 'value__' is reserved and cannot be used

We clearly see here that we have value__ as reserved member for the enumerator. C# compiler like this keyword has large number of reserved inbuilt keywords.

Akhil Mittal (csharppulse.blogspot.in)

Image credit: www.vector.rs

It may keep this reserved keyword to keep track of the enum members internally but not sure.

Summary:

Let's recall all the points that we have to remember,

1) An enum represents for a constant number, and an enum type is known as a distinct type having named constants.
2) We can't declare char as an underlying data type for enum objects because char stores Unicode characters, but enum objects data type can only be number.
3) An enum can't be derived from any other type except that of type byte, sbyte, short, ushort, int, uint, long, or ulong.
4) By default enum is a sealed class and therefore sticks to all the rules that a sealed class follows, so no class can derive from enum i.e. a sealed type.
5) The enum type is implicitly derived from System.Enum and so we cannot explicitly derive it from System.Enum.
6) enum is also derived from three interfaces IComparable, IFormattable and IConvertible.
7) A numerous predefined conversion methods can be used to convert enum from one data type to another.
8) More than one enum members can be initialized a same constant value.
9) An enum acts as a constant, so its value cannot be changed once initialized.
10) The enumerator name 'value__' is reserved and cannot be used

Conclusion:

Akhil Mittal (sharppulse.blogspot.in)

With this article we completed almost all the scenarios related to enum. We did a lot of hands-on lab to clear our concepts. I hope my readers now know by heart about these basic concepts and will never forget them. These may also help you in cracking C# interviews.

Keep coding and enjoy reading 

Also do not forget to rate/comment/like my article if it helped you by any means, this helps me to get motivated and encourages me to write more and more.

Read more:

- **Learning MVC series.**
- **C# and Asp.Net Questions (All in one)**
- **MVC Interview Questions**
- **C# and Asp.Net Interview Questions and Answers**
- **Web Services and Windows Services Interview Questions**

For more technical articles you can reach out to **A Practical Approach**.

Akhil Mittal (sharppulse.blogspot.in)