Enumeration

An **enum** is a custom data type of name-value pairs. The enumeration defines named constants, corresponding to discrete numerical values. By default, the first element is set to the value zero (0), followed by an n+1 progression. You are free to change the initial value as you see fit.

```
enum EmpType
{
        Manager,     // = 0
        Grunt,       // = 1
        Contractor   // = 2
}
```

Enumerations do not necessarily need to follow a sequential ordering and do not need to have unique values.

```
enum EmpType
{
        Manager = 10,
        Grunt = 1,
        Contractor = 21
}
```

By default, the storage type used to hold the values of an enumeration is a System.Int32 (the C# int); however, you are free to change this to your liking. C# enumerations can be defined in a similar manner for any of the core system types (byte, short, int, or long). For example, if you want to set the underlying storage value of EmpType to be a byte rather than an int, you can write the following:

```
enum EmpType : byte
{
        Manager = 10,
        Grunt = 1,
        Contractor = 100
}
```

Because enumerations are nothing more than a user-defined data type, you are able to use them as function return values, method parameters, local variables, and so forth.

```
static void Main(string[] args)
{
        EmpType emp = EmpType.Grunt;
        Console.WriteLine(emp);       // Grunt
        Console.WriteLine((byte)emp);      //  1
        Console.ReadKey();
}
```

The System.Enum Type

The interesting thing about .NET enumerations is that they gain functionality from the System.Enum class type. This class defines a number of methods that allow you to interrogate and transform a given enumeration. One helpful method is the static *Enum.GetUnderlyingType()*, which, as the name implies, returns the data type used to store the values of the enumerated type. To do so you need to know that the Enum. GetUnderlyingType() method requires to pass in a System.Type as the first parameter.

```
Enum.GetUnderlyingType(emp.GetType());
```

Structure

Structure types are well suited for modeling mathematical, geometrical, and other "atomic" entities in your application. A **structure** is a user-defined type; however, structures are not simply a collection of name-value pairs. Rather, structures are types that can contain any number of data fields and members that operate on these fields.

If we look from the OOP's point of view we can think of a structure as a "lightweight class type," given that structures provide a way to define a type that supports encapsulation but cannot be used to build a family of related types. When we need to build a family of related types through inheritance, we will need to make use of class types.

```
struct Point
{
        public int X;
        public int Y;
        public void Increment() { X++; Y++; }
        public void Decrement() { X--; Y--; }
        public void Display()
        {
                Console.WriteLine($"X = {X}, Y = {Y}");
        }
 }
static void Main(string[] args)
{
        // Create an initial Point.
        Point myPoint;
        myPoint.X = 349;
        myPoint.Y = 76;
        myPoint.Display();

        // Adjust the X and Y values.
        myPoint.Increment();
        myPoint.Display();

        Console.ReadKey();
}
```

If you do not assign each piece of public field data (X and Y in this case) before using the structure, you will receive a compiler error.

As an alternative, you can create structure variables using the C# new keyword, which will invoke the structure's default constructor. By definition, a default constructor does not take any arguments. The benefit of invoking the default constructor of a structure is that each piece of field data is automatically set to its default value.

```
// Set all fields to default values
// using the default constructor.
Point p1 = new Point();

// Prints X=0, Y=0.
p1.Display();
```

It is also possible to design a structure with a custom constructor. This allows you to specify the values of field data upon variable creation, rather than having to set each data member field by field.

```
struct Point
{
        // Fields of the structure.
        public int X;
        public int Y;

        // A custom constructor.
        public Point(int XPos, int YPos)
        {
                X = XPos;
                Y = YPos;
        }
        ...
}
```

With this, you could now create Point variables, as follows:

```
// Call custom constructor.
Point p2 = new Point(50, 60);

// Prints X=50,Y=60.
p2.Display();
```