Structure, Enum and Interface

Struct

Classes define types that support inheritance and polymorphism. They enable you to create sophisticated behaviours based on hierarchies of derived classes. By contrast, `struct` types are simpler types whose primary purpose is to store data values. Structs can't declare a base type; they implicitly derive from `System.ValueType`. You can't derive other struct types from a struct type. They're implicitly sealed.

A structure type (or `struct` type) is a value type that can encapsulate data and related functionality. You use the struct keyword to define a structure type:

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Structure types have value semantics. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. In the case of a structure-type variable, an instance of the type is copied.

Typically, you use structure types to design small data-centric types that provide little or no behavior. For example, .NET uses structure types to represent a number (both integer and real), a Boolean value, a Unicode character, a time instance. If you're focused on the behavior of a type, consider defining a class. Class types have reference semantics. That is, a variable of a class type contains a reference to an instance of the type, not the instance itself.

Limitations with the design of a structure type

When you design a structure type, you have the same capabilities as with a class type, with the following exceptions:

- You can't declare a parameterless constructor. Every structure type already provides an implicit parameterless constructor that produces the default value of the type.
- You can't initialize an instance field or property at its declaration. However, you can initialize a static or const field or a static property at its declaration.
- A constructor of a structure type must initialize all instance fields of the type.
- A structure type can't inherit from other class or structure type and it can't be the base of a class. However, a structure type can implement interfaces.

- You can't declare a finalizer within a structure type.

Interface

An `interface` defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface typically doesn't provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement. An interface may define static methods, which must have an implementation. Beginning with C# 8.0, an interface may define a default implementation for members. An interface may not declare instance data such as fields, auto-implemented properties, or property-like events.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the `interface` keyword as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of an interface must be a valid C# identifier name. By convention, interface names begin with a capital `I`.

Any class or struct that implements the `IEquatable<T>` interface must contain a definition for an `Equals` method that matches the signature that the interface specifies. As a result, you can count on a class that implements `IEquatable<T>` to contain an `Equals` method with which an instance of the class can determine whether it's equal to another instance of the same class. The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`. A `class` or `struct` can implement multiple interfaces, but a class can only inherit from a single class.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member. When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface declares but doesn't provide a default implementation for. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the `IEquatable<T>` interface. The implementing class, `Car`, must provide an implementation of the `Equals` method.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }
    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}
```

Interfaces can inherit from one or more interfaces. The derived interface inherits the members from its base interfaces. A class that implements a derived interface must implement all members in the derived interface, including all members of the derived interface's base interfaces.

Enum

An *enumeration type* (or enum type) is a value type defined by a set of named constants of the underlying integral numeric type. To define an enumeration type, use the `enum` keyword and specify the names of enum members:

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

By default, the associated constant values of enum members are of type `int`; they start with zero and increase by one following the definition text order. You can explicitly specify any other integral numeric type as an underlying type of an enumeration type. You can also explicitly specify the associated constant values, as the following example shows:

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

You can also define an `enum` to be used in combination as **`flags`**. The following declaration declares a set of flags for the four seasons. Any combination of the seasons may be applied, including an All value that includes all seasons:

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

The following example shows declarations of both the preceding enums:

```
var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

For any enumeration type, there exist explicit conversions between the enumeration type and its underlying integral type. If you cast an enum value to its underlying type, the result is the associated integral value of an enum member.

Example:

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output: 
Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```