5.16 Structs

Structs are composed of several pieces of data, possibly of different types. They have data members and function members. They enable you to define your own types of variables based on this structure. Structs are declared outside of the main body of the code. Structs are defined using the struct keyword as follows:

```
struct StructName
{
      MemberDeclarations
}
```

The `MemberDeclarations` *section contains declarations of variables (called the data members of the struct) in almost the same format as usual. Each member declaration takes the following form:*

```
< accessibility > < type > < name >;
```

For example, the following code declares a struct named Point (which are demonstrated in Fig. 5.16) . It has two public fields, named X and Y. In Main, three variables of struct type Point are declared, and their values are assigned and printed out.

```
1    // Fig. 5.16 : Points.cs
2    // Declare Point with two fields (x,y) as structure
3    struct  Point
4    {
5      public int  X;
6      public int  Y;
7    }
8    class  Program
9    {
10     static void  Main()
11     {
12         Point  first, second, third;
13         first.X  = 10;  first.Y = 10;
14         second.X = 20;  second.Y = 20;
15         third.X  = first.X + second.X;
16         third.Y  = first.Y + second.Y;
17         Console.WriteLine("first: {0}, {1}", first.X,  first.Y);
18         Console.WriteLine("second:{0}, {1}", second.X, second.Y);
19         Console.WriteLine("third: {0}, {1}", third.X,  third.Y);
20     }
   }
```

Fig . 5.16 | Declare Point with two field (x, y) using `struct` Structure.

The output of the previous program is:

```
first: 10, 10
second: 20, 20
third: 30, 30
```

Structures with Constructors

Structs can have instance constructors, but destructors are not allowed. The language implicitly supplies a parameterless constructor for every struct. This constructor sets each of the struct's members to the default value for that type. Value members are set to their default values. Reference members are set to null.

The predefined parameterless constructor exists for every struct—and you cannot delete or redefine it. You can, however, create additional constructors, as long as they have parameters. For example, the following code declares a simple struct with a constructor that takes two int parameters (which are demonstrated in Fig. 5.24). Main creates two instances of the struct—one using the implicit parameterless constructor and the second with the declared two-parameter constructor.

```
1   // Fig. 5.24: ConPoints.cs
2   // Declare Point with two fields (x,y) as structure with constuctor
3   struct Simple
4   {
5     public int  X;
6     public int  Y;
7
8     public Simple(int a,  int b) // Constructor  with  parameters
9     {
10       X  = a; Y  = b;
11    }
12  }
13  class  Program
14  {
15    static void  Main()
16    {
17       Simple s1 = new  Simple();
18       Simple s2 = new  Simple(5,  10);
19       Console.WriteLine("{0},{1}", s1.X, s1.Y);
20       Console.WriteLine("{0},{1}", s2.X, s2.Y);
21    }
22  }
```

Fig . 5.17 | Declare Point with two field (x, y) using **`struct`** Structure with Constructor.

5.17 Enumerations

An enumeration, or enum, is a programmer-defined type, such as a struct, enumerations like, structs, are declared outside of the main body of the code. Like structs, enums are value types and therefore store their data directly, rather than separately, with a reference and data. Enums have only one type of member: named constants with integer values. Enums can be defined using the enum keyword as follows:

```
enum typeName
{
  value1 ,
  value2 ,
  value3 ,
  ...
  valueN
}
```

The following code shows an example of the declaration of a new enum type called TrafficLight, which contains three members. Notice that the list of member declarations is a comma-separated list; there are no semicolons in an enum declaration.

Every enum type has an underlying integer type, which by default is int, and is assigned a constant value of the underlying type. By default, the compiler assigns 0 to the first member and assigns each subsequent member the value one more than the previous member.For example, in the TrafficLight type, the compiler assigns the int values 0, 1, and 2 to members Green, Yellow, and Red, respectively. In the output of the following code, you can see the underlying member values by casting them to type int.

```
TrafficLight t1  = TrafficLight.Green;
TrafficLight t2  = TrafficLight.Yellow;
TrafficLight t3  = TrafficLight.Red;
```

```
Console.WriteLine("{0},\t{1}",    t1, (int) t1);
Console.WriteLine("{0},\t{1}",    t2, (int) t2);
Console.WriteLine("{0},\t{1}\n", t3, (int) t3);
```

This code produces the following output:

```
Green,  0
Yellow, 1
Red, 2
```

5.17.1 Setting the Underlying Type and Explicit Values

You can use an integer type other than int by placing a colon and the type name after the enum name. The type can be any integer type. All the member constants are of the enum's underlying type.

The values of the member constants can be any values of the underlying type. To explicitly set the value of a member, use an initializer after its name in the enum declaration. There can be duplicate values, although not duplicate names, as shown here:

```
enum  TrafficLight
{
    Green  = 10,
    Yellow = 15, // Duplicate  values
    Red=15       // Duplicate  values
}
```

For example, the code in Figure 16-25 shows two equivalent declarations of enum TrafficLight. The code on the left accepts the default type and numbering. The code on the right explicitly sets the underlying type to int and the members to values corresponding to the default values.

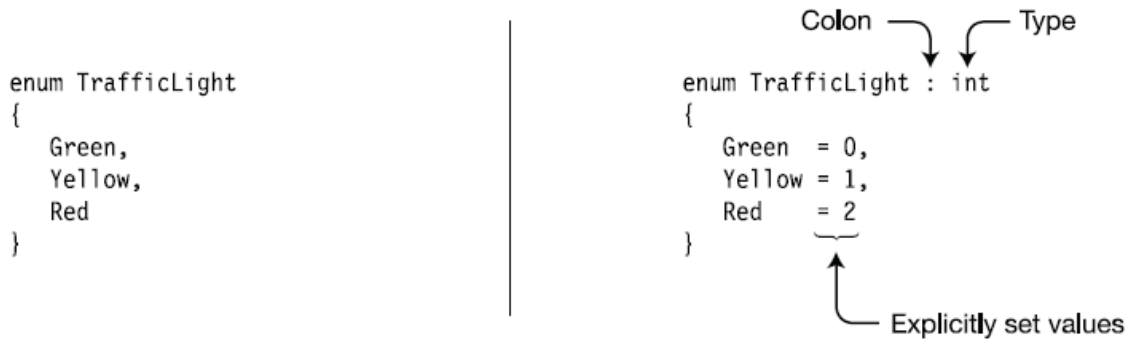

Fig . 5.18 | Equivalent enum declarations

5.17.2 Implicit Member Numbering

You can explicitly assign the values for any of the member constants. If you don't initialize a member constant, the compiler implicitly assigns it a value. For example, the following code declares two enumerations. CardSuit accepts the implicit numbering of the members, as shown in the comments. FaceCards sets some members explicitly and accepts implicit numbering of the others.

```
enum CardSuit
{
    Hearts,                 // 0  - Since this is first
    Clubs,                  // 1  - One more than the previous one
    Diamonds,               // 2  - One more than the previous one
    Spades,                 // 3  - One more than the previous one
    MaxSuits                // 4  - A common way to assign a constant
}                           //      to the number of listed items

enum FaceCards
{
    // Member                  // Value assigned
    Jack      = 11,             // 11 - Explicitly set
    Queen,                      // 12 - One more than the previous one
    King,                       // 13 - One more than the previous one
    Ace,                        // 14 - One more than the previous one
    NumberOfFaceCards = 4,      // 4  - Explicitly set
    SomeOtherValue,             // 5  - One more than the previous one
    HighestFaceCard   = Ace     // 14 - Ace is defined above
}
```

More About Enums

An enum is a distinct type. Comparing enum members of different enum types results in a compile- time error. For example, the following code declares two different enum types with the exact same structure and member names.

```
1       // Fig. 5.19: TwoEnum.cs
2       // Declare two enumeration and some operation on them
3       enum  FirstEnum          // First enum  type
4       {
5            Mem1,
6            Mem2
7       }
8       enum  SecondEnum          // Second  enum type
9       {
10          Mem1,
11          Mem2
12      }
```

```
13      class  Program
14      {
15          static void  Main()
16          {
17               // OK--members  of  same  enum  type
18
19               if (FirstEnum.Mem1 < FirstEnum.Mem2)
20                   Console.WriteLine("True");
21
22               // Error--different enum  types
23               if (FirstEnum.Mem1 < SecondEnum.Mem1)
24                      Console.WriteLine("True");
25          }
26      }
```

Fig . 5.19 | Equivalent enum declarations

The first if statement is fine because it compares different members from the same enum type. The second if statement produces an error because it attempts to compare members from different enum types. This error occurs even though the structures and member names are exactly the same.