C# - CLASSES

When you define a class, you define a blueprint for a data type. This does not actually define any data, but it does define what the class name means. That is, what an object of the class consists of and what operations can be performed on that object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

Defining a Class

A class definition starts with the keyword class followed by the class name; and the class body enclosed by a pair of curly braces. Following is the general form of a class definition –

```
<access specifier> class  class_name {
   // member variables
   <access specifier> <data type> variable1;
   <access specifier> <data type> variable2;
   ...
   <access specifier> <data type> variableN;
   // member methods
   <access specifier> <return type> method1(parameter_list) {
      // method body
   }
   <access specifier> <return type> method2(parameter_list) {
      // method body
   }
   ...
   <access specifier> <return type> methodN(parameter_list) {
      // method body
   }
}
```

Note –

- Access specifiers specify the access rules for the members as well as the class itself. If not mentioned, then the default access specifier for a class type is **internal**. Default access for the members is **private**.
- Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
- To access the class members, you use the dot . operator.
- The dot operator links the name of an object with the name of a member.

The following example illustrates the concepts discussed so far –

Live Demo

```
using System;

namespace BoxApplication {
   class Box {
```

```
      public double length;   // Length of a box
      public double breadth;  // Breadth of a box
      public double height;   // Height of a box
   }
   class Boxtester {
      static void Main(string[] args) {
         Box Box1 = new Box();   // Declare Box1 of type Box
         Box Box2 = new Box();   // Declare Box2 of type Box
         double volume = 0.0;    // Store the volume of a box here

         // box 1 specification
         Box1.height = 5.0;
         Box1.length = 6.0;
         Box1.breadth = 7.0;

         // box 2 specification
         Box2.height = 10.0;
         Box2.length = 12.0;
         Box2.breadth = 13.0;

         // volume of box 1
         volume = Box1.height * Box1.length * Box1.breadth;
         Console.WriteLine("Volume of Box1 : {0}",  volume);

         // volume of box 2
         volume = Box2.height * Box2.length * Box2.breadth;
         Console.WriteLine("Volume of Box2 : {0}", volume);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition similar to any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are the attributes of an object *fromdesignperspective* and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Let us put above concepts to set and get the value of different class members in a class –

Live Demo

```
using System;

namespace BoxApplication {
   class Box {
      private double length;   // Length of a box
      private double breadth;  // Breadth of a box
```

```
      private double height;   // Height of a box

      public void setLength( double len ) {
         length = len;
      }
      public void setBreadth( double bre ) {
         breadth = bre;
      }
      public void setHeight( double hei ) {
         height = hei;
      }
      public double getVolume() {
         return length * breadth * height;
      }
   }
   class Boxtester {
      static void Main(string[] args) {
         Box Box1 = new Box();   // Declare Box1 of type Box
         Box Box2 = new Box();
         double volume;

         // Declare Box2 of type Box
         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}" ,volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

C# Constructors

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor has exactly the same name as that of class and it does not have any return type. Following example explains the concept of constructor –

Live Demo

```
using System;

namespace LineApplication {
   class Line {
      private double length;   // Length of a line

      public Line() {
         Console.WriteLine("Object is being created");
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
         return length;
      }

      static void Main(string[] args) {
         Line line = new Line();

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Object is being created
Length of line : 6
```

A **default constructor** does not have any parameter but if you need, a constructor can have parameters. Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation as shown in the following example –

Live Demo

```
using System;

namespace LineApplication {
   class Line {
      private double length;   // Length of a line

      public Line(double len) {  //Parameterized constructor
         Console.WriteLine("Object is being created, length = {0}", len);
         length = len;
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
```

```
         return length;
      }
      static void Main(string[] args) {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}", line.getLength());

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

C# Destructors

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope. A **destructor** has exactly the same name as that of the class with a prefixed tilde and it can neither return a value nor can it take any parameters.

Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

Following example explains the concept of destructor –

Live Demo

```
using System;

namespace LineApplication {
   class Line {
      private double length;   // Length of a line

      public Line() {   // constructor
         Console.WriteLine("Object is being created");
      }
      ~Line() {   //destructor
         Console.WriteLine("Object is being deleted");
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
         return length;
      }
      static void Main(string[] args) {
         Line line = new Line();

         // set line length
         line.setLength(6.0);
```

```
         Console.WriteLine("Length of line : {0}", line.getLength());
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Object is being created
Length of line : 6
Object is being deleted
```

Static Members of a C# Class

We can define class members as static using the **static** keyword. When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

The keyword **static** implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.

The following example demonstrates the use of **static variables** –

Live Demo

```
using System;

namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public int getNum() {
         return num;
      }
   }
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s1 = new StaticVar();
         StaticVar s2 = new StaticVar();

         s1.count();
         s1.count();
         s1.count();

         s2.count();
         s2.count();
         s2.count();

         Console.WriteLine("Variable num for s1: {0}", s1.getNum());
         Console.WriteLine("Variable num for s2: {0}", s2.getNum());
         Console.ReadKey();
      }
```

```
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Variable num for s1: 6
Variable num for s2: 6
```

You can also declare a **member function** as **static**. Such functions can access only static variables. The static functions exist even before the object is created. The following example demonstrates the use of **static functions** –

Live Demo

```
using System;

namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public static int getNum() {
         return num;
      }
   }
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s = new StaticVar();

         s.count();
         s.count();
         s.count();

         Console.WriteLine("Variable num: {0}", StaticVar.getNum());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Variable num: 3
```