.NET Framework 4.7 and C# 8.0

Lesson 05 : OOP using C#

Capgemini

Lesson Objectives

- On completion of this module on OOPS Concepts in C#, you will be able to explain:
  - The concept of a class
  - Different types of constructors in C#
  - Structures in C#
  - Difference between class and structures
  - Inheritance in C#
  - Properties and Indexers

Lesson Objectives

- Polymorphism in C# which includes Function Overloading and Function Overriding
- The concept of the Abstract Class and Sealed Class
- Interfaces in C#
- Difference between Interfaces in C# and the Abstract Class
- Method Parameters in C#

What are classes?

- A class is a user-defined type (UDT) that is composed of field data (member variables) and methods (member functions) that act on this data.
- In C#, classes can contain the following:
  - Constructors and destructors
  - Fields and constants
  - Methods
  - Properties
  - Indexers
  - Overloaded operators
  - Nested types
    - Classes & Structs
    - interfaces
    - Enumerations
    - Delegates
    - Event

Class Example

```
public class Employee
{
      private int employeeId ;
      private string employeeName;
      public Employee() { }   //Constructor
      //Property
      public int EmployeeId
      {
                  get  { return employeeId ;}
                  set  { employeeId = value;}
    }
}
```

Definition and Types of Constructor

- A constructor is automatically called immediately after an object is created to initialize it.
- Constructors have the same names as their class names.
- Default constructor: Default Constructor will get automatically created and invoked, if constructor is not Specified in the class and assign the instance variables with their default values.
- Static constructor: This is similar to static method. It must be parameter less and must not have an access modifier (private or public).

Constructor Example

```
public class Employee
{    static Employee()  // static constructor
     {. . .}
     public Employee() // default constructor
     {. . .}
     public Employee(string name) // parameterized constructor
     {. . .}
}
//Creating object of Class
public class Program
{          static void Main()
          {          Employee emp = new Employee();          }
}
```

Definition of Method

- A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using the following method-declaration:

```
[attributes]
[method-modifiers] return-type method-name-identifier ( [formal-parameter-list] )
{
        [statements]
}
```

- There are four kinds of parameters:
  - out
  - ref
  - params
  - value.

Example of a Class Method

```
public class Employee
{    public Employee() { . . .}
     public static void StaticMethod() { . . .}
     public void NonStaticMethod() { . . .}
}
public class Program
{       static void Main()
        {
          Employee emp = new Employee();
          emp.NonStaticMethod();
          Employee.StaticMethod();
        }
}
```

The Value Parameter

static void Mymethod(int Param1) **{** **Param1=100;** **}**	**static void Main()** **{** **int Myvalue=5;** **MyMethod(Myvalue);** **Console.WriteLine(Myvalue);** **}**

- Output would be 5.
- Though the value of the parameter Param1 is changed within MyMethod, it is not passed back to the calling part, since the value parameters are ‘input only’.

The Params Parameter

```
static int Sum(params int[] Param1)
{
    int val=0;
    foreach(int P in Param1)
    {
      val=val+P;
    }
   return val;
 }
```

```
static void Main()
{
Console.WriteLine(Sum(1,2,3));
Console.WriteLine(Sum(1,2,3,4,5);
}
```

- Output: 6 and 15

Method Overloading And Polymorphism

- In C#, two or more methods within the same class can share the same name, if their parameter declarations are different.
- In such cases, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways in which C# implements polymorphism.

Overloading Of Constructors

- Like methods, constructors can also be overloaded.
- Overloading of constructors allows you to construct objects in a variety of ways.

Creation Of Static Members

- When a member is declared static, it can be accessed even before any objects of its class are created
- It can be accessed without a reference to any object
- You can declare both methods and variables to be static
- Outside the class, to use a static member, you must specify the name of its class followed by the dot operator
- No object needs to be created

What Is A Static Constructor?

- A constructor can also be specified as static
- A static constructor is typically used to initialize attributes that apply to a class rather than an instance
- A static constructor is used to initialize aspects of a class before any objects of the class are created

Types Of Class Accessibility

- Types of class accessibilities are as follows:
- **Public:** Access is not restricted.
- **Private:** Access is limited to the containing type.
- **Protected:** Access is limited to the containing class or types derived from the containing class.
- **Internal:** Access is limited to the current assembly.
- **Protected internal:** Access is limited to the current assembly or types derived from the containing class.

Use of Properties

- Properties provide the chance to protect a field in a class by reading and writing to it through the property accessor
- Accomplished in programs by implementing the specialized getter and setter methods
- One or two code blocks are required: Those representing a get accessor and/or a set accessor
- The code block for the get accessor is executed when the property is read
- The code block for the set accessor is executed when the property is assigned a new value

Properties and Accessors

- A property without a set accessor is considered read-only
- A property without a get accessor is considered write-only
- A property that has both the accessors is read-write
- Uses of Properties
  - They can validate data before allowing a change.
  - They can transparently expose data on a class where that data is actually retrieved from some other source, such as a database.
  - They can take an action when data is changed, such as raising an event, or changing the value of other fields.

Example Properties

```
public class Date
{
    private int month;
    public int Month
    {       get
            {       return month;           }
            set
            {       if ((value > 0) && (value < 13))
                    {       month = value;          }
             }
    }
}
```

Asymmetric Accessor Accessibility

- C# 2.0 introduced a concept, called Asymmetric Accessor Accessibility
- It allows to modify the visibility of either the get accessor or set accessor on a class property that has both a getter and setter.

```
public class Customer
{    private int _customerID;
     public int ID
     {          get
                {      return _customerID;   }
                internal set
                 {     _customerID = value;  }
     }
     // ....
}
```

Use Of Auto-Implemented Properties

- Automatically implemented properties provide a more concise syntax for implementing getter setter pattern, where the C# compiler automatically generates the backing fields.

```
public class Point
{
        public int PointX { get; set; }
        public int PointY { get; set; }
}
```

Use Of Auto-Implemented Properties

- Auto-implemented properties must declare both a get and a set accessor.
- To create a ‘read only’ auto-implemented property, use a private set accessor.

```
public class Point
{
        public int X { get; private set; }  //read only
        public int Y { get; set; }
}
```

Example Auto-Implemented Properties

```
public class Customer
{
    public int CustomerID { get; private set; }  // readonly property
    public string Name { get; set; }
    public string City { get; set; }
    public override string ToString()
    {
                return Name + "\t" + City + "\t" + CustomerID;
     }
}
```

Example Auto-Implemented Properties

```
static void Main(string[] args)
{
    Customer c = new Customer();
    c.Name = "Maria Anders";
    c.City = "Berlin";
    c.CustomerID = 1;                    //should throw an error
    Console.WriteLine(c);
}
```

Demo

- Defining and using properties in C#

What are indexers?

- Indexers are ‘smart arrays’.
- Indexers permit instances of a class or struct to be indexed in the same way as arrays.
- Indexers are similar to properties except that their accessors take parameters.
- Simple declaration of indexers is as follows:

Modifier type this [formal-index-parameter-list]
{accessor-declarations}

Example indexers

```
class IntIndexer
{
        private string[] myData;
          public IntIndexer(int size)
         {
               myData=new string[size];
         }
          public string this[int pos]
          {
               get{return myData[pos];}
               set {myData[pos] = value}
         }
}
```

```
static void Main(string[] args)
{
   int size = 10;
   IntIndexer myInd = new
IntIndexer(size);
   myInd[9] = "Some Value";
   myInd[3] = "Another Value";
   myInd[5] = "Any Value";
 }
```

Demo

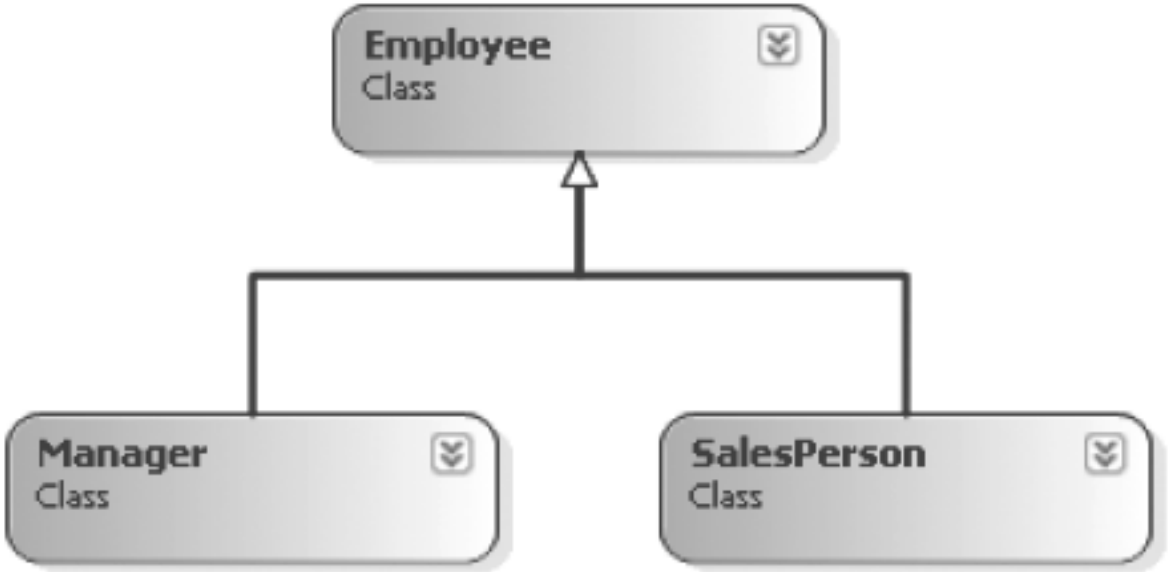
- Indexers in C#

What Is Class Inheritance?

- Inheritance is a form of software reusability in which classes are created by reusing the data and behaviors of an existing class with new capabilities
- A class inheritance hierarchy begins with a base class that defines a set of common attributes and operations that it shares with derived classes
- A derived class inherits the resources of the base class and overrides or enhances their functionality with new capabilities.
- The classes are separate, but related

Class Inheritance

- Inheritance is also called ‘is a’ relationship
- A SalesPerson ‘is-a’ Employee (as is a Manager)
- Base classes (such as Employee) are used to define general characteristics that are common to all descendents.
- Derived classes (such as SalesPerson and Manager), the general functionalities, are extended while adding more specific behaviors.



Constructors And Their Inheritance

- In a hierarchy, both the base classes and derived classes can have their own constructors.
- The constructor for the base class constructs the base class portion of the object, and the constructor for the derived class constructs the derived class part.
- A derived class can call a constructor defined in its base class by using the following:
  - An expanded form of constructor declaration of the derived class
  - The base keyword

Hiding Name Of Base Class Member

- It is possible for a derived class to define a member that has the same name as a member in its base class.
- When this happens, the member in the base class is hidden within the derived class.
- Even though this is not technically an error in C#, the compiler issues a warning message.
- If you intended to hide a base class member purposely, then to prevent this warning, the derived class member must be preceded by the **new** keyword.

Reference Of Derived Object To Base Variable

- A reference variable of a base class can be assigned a reference to an object of any class derived from the base class.

- When a reference to a derived class object is assigned to a base class reference variable, you have access only to the parts of the object that are defined by the base class.

Demo

- Inheritance in C#

Function Overriding By Polymorphism

- Polymorphism provides a way for a subclass to customize the implementation of a method defined by its base class.

```
public class Employee
{
  // GiveBonus() has a default implementation, however
  // child classes are free to override this behavior
  public virtual void GiveBonus(float amount)
  {          currPay += amount;          }
}
```

- If, in a base class, you define a method that may be overridden by a subclass, you should specify the method as virtual using the virtual modifier:

Function Overriding By Polymorphism

- A subclass uses the override keyword to redefine a virtual method:

```
public class SalesPerson : Employee
{  // A salesperson's bonus is influenced by the number of sales.
   public override void GiveBonus(float amount)
   {
        int salesBonus = 0;
        if(numberOfSales >= 0 && numberOfSales <= 100)
           salesBonus = 10;
        else if(numberOfSales >= 101 && numberOfSales <= 200)
           salesBonus = 15;
        else
           salesBonus = 20; // Anything greater than 200.
        base.GiveBonus (amount * salesBonus);
    }
    ...
}
```

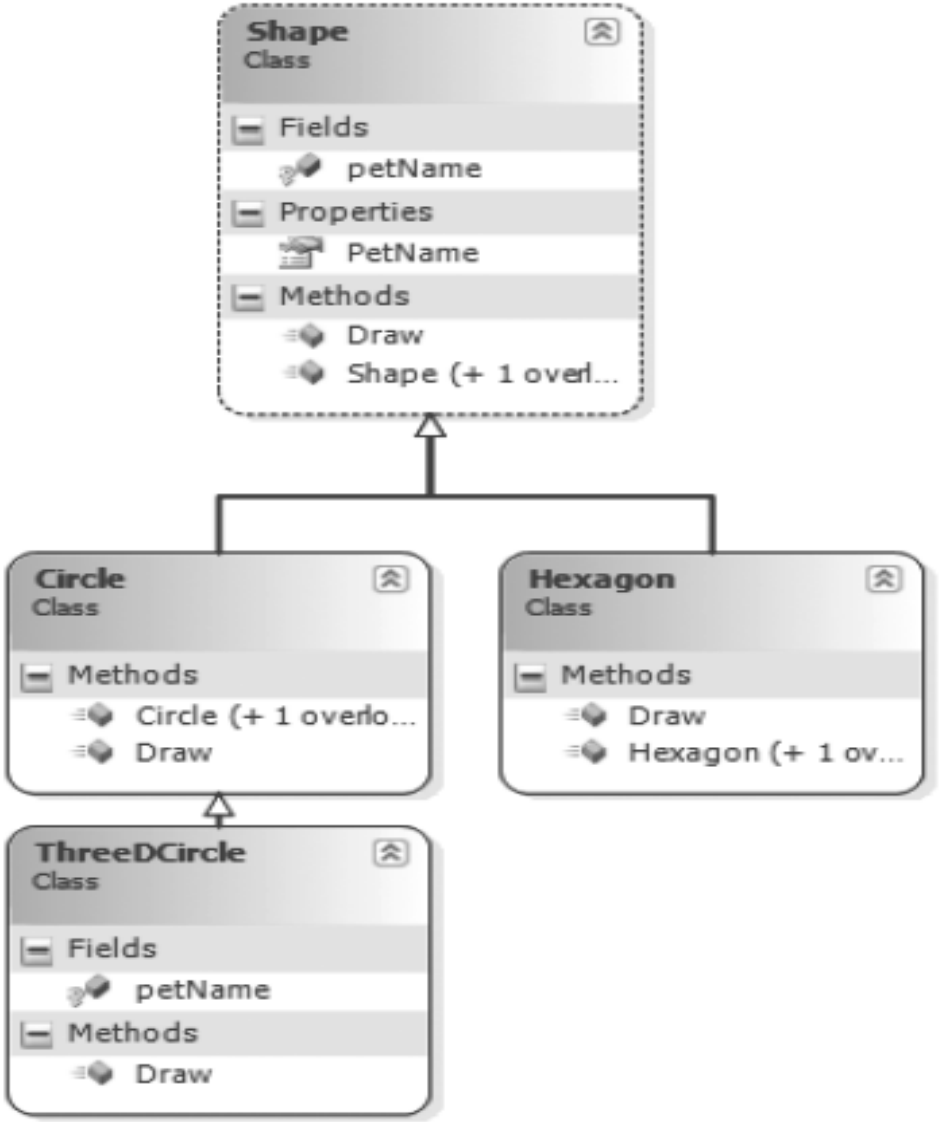
Virtual Methods

- A **virtual** method is a method that is declared as virtual in a base class and redefined in one or more derived classes.
- Each derived class can have its own version of a virtual method.
- You declare a method as virtual inside a base class by preceding its declaration with the keyword virtual.
- When a virtual method is redefined by a derived class, the **override** modifier is used.

What Is An Abstract Class?

- An abstract class is the one that cannot be instantiated
- It is intended to be used as a base class
- It may contain abstract and non-abstract function members
- It cannot be sealed

Example


Shape
Class
Fields
petName
Properties
PetName
Methods
Draw
Shape (+ 1 overl...
Circle
Class
Methods
Circle (+ 1 overlo...
Draw
Hexagon
Class
Methods
Draw
Hexagon (+ 1 ov...
ThreeDCircle
Class
Fields
petName
Methods
Draw

Characteristics of Abstract Methods

- Abstract methods do not have an implementation in the abstract base class
- Every concrete derived class must override all the base-class abstract methods and properties using the keyword override
- Abstract methods must belong to an abstract class
- These methods are intended to be implemented in a derived class

Abstract Class, Virtual and Abstract Methods

Abstract class:

```
public abstract class AbstractClass
{
        public AbstractClass()
        {
        }

         public abstract int AbstractMethod();

          public virtual int VirtualMethod()
          {
                    return 0;
          }
}
```

Derived class:

```
public class DerivedClass : AbstractClass
{
        public DerivedClass()
        {

        }
         public override int AbstractMethod()
         {
             return 0;
         }
         public override int VirtualMethod()
         {
             return base.VirtualMethod ();
         }
}
```

Demo

- Function Overriding, Abstract Class and Abstract Methods

Characteristics Of Sealed Class

- To prevent inheritance, a sealed modifier is used to define a class.
- A sealed class is the one that cannot be used as a base class. Sealed classes can’t be abstract.
- All structs are implicitly sealed.
- Many .NET Framework classes are sealed: String, StringBuilder, and so on.
- Why seal a class?
  - For prevention of unintended derivation
  - For code optimization
  - For resolution of Virtual function calls at compile-time

Sealed Class Example

```
using System;

sealed class MyClass
{
    public int x;
    public int y;
}

// class MainClass
 class MainClass: MyClass {  }  causes error
```

What are interfaces?

- An interface defines a contract.
- Interface is a purely abstract class; it has only signatures, no implementation.
- May contain methods, properties, indexers and events (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly public abstract (virtual).
- Interface members must not be static.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.

Implementation of Interfaces

- A class can inherit from a single base class, but can implement multiple interfaces.
- A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, indexer) must be implemented or inherited from a base class.
- Implemented interface methods must not be declared as override.
- Implemented interface methods can be declared as virtual or abstract (that is, an interface can be implemented by an abstract class).

Interfaces-Example

```
interface IMyInterface: IBase1, IBase2
{
   void MethodA();
   void MethodB();
}
```

Interfaces-Example

- Interfaces can be implemented by classes.
- The identifier of the implemented interface appears in the class base list.
- For example:

```
class Class1: Iface1, Iface2
{
   // class members
}
```

Interfaces-Example

- When a class base list contains a base class and interfaces, the base class is declared first in the list. For example:

```
class ClassA: BaseClass, Iface1, Iface2
{
   // class members
}
```

Interfaces-Example

- If two interfaces have the same method name, you can explicitly specify **interface + method** name to clarify their implementations.

```
interface IVersion1
{
void GetVersion();
}
interface IVersion2
{
void GetVersion();
}
```

```
interface IVersion: IVersion1,
IVersion2
{
void IVersion1.GetVersion();
void IVersion2.GetVersion();
}
```

Differences: Abstract Classes And Interface

- Abstract classes can be used to define public, private and protected state data, as well as any number of concrete methods that can be accessed by the subclasses.
- Interfaces, on the other hand, are pure protocols.
- Interfaces never define data types, and never provide a default implementation of the methods.

Demo

- Creating and using abstract classes and Interfaces in C#

Static Class

- In C#, static class is created by using static keyword
- A static class can only contain static data members, static methods, and a static constructor.
- It is not allowed to create objects of the static class.
- Static classes are sealed

Use of Structs in C#

- Classes and Structs Similarities:
- Both are user-defined types
- Both can implement multiple interfaces
- Both can contain the following
  - Data
    - Fields, constants, events, arrays
  - Functions
    - Methods, properties, indexers, operators, constructors
  - Type definitions
    - Classes, structs, enums, interfaces, delegates

Use of Structs in C#

Class	Struct
Reference type	Value type
Can inherit from any non-sealed reference type	No inheritance (inherits only from System.ValueType)
Can have a destructor	No destructor
Can have user-defined parameterless constructor	No user-defined parameterless constructor

Use of Structs in C# (Contd.)

```
public struct Point
{
    int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int X
    {
        get { return x; }
        set { x = value; }
    }
    public int Y
    {
        get { return y; }
        set { y = value; }
    }
}
```

```
Point p = new Point(2,5);
p.X += 100;
int px = p.X; // px = 102
```

Enums in C#

- An enumeration is a set of named integer constants.
- An enumerated type is declared using the enum keyword.
- C# enumerations are value data type.
- The general syntax for declaring an enumeration is :

  enum <enum_name> { enumeration list };
- By default, the first member of an enum has the value 0 and the value of each successive enum member is increased by 1.

Use and Types of Extension Method

- It is a special kind of static method.
- Allows the addition of methods to an existing class outside the class definition.
  - Without creating a new derived type
  - Without re-compiling or modifying the original type
- Called the same way regular methods are called.
  - It is declared by specifying the keyword this as a modifier.
  - It is the first parameter of the methods.
  - It can only be declared in static classes.

Use of Extension Methods - Restrictions

- Extension methods cannot be used to override existing methods.
- An extension method with the same name and signature as an instance method will not be called.
- The concept of extension methods cannot be applied to fields, properties or events.
- Extension methods should be used cautiously.

Creation of Extension Methods

```
namespace StringExtensions
{
public static class StringExtensionsClass
{
    public static string RemoveNonNumeric(this string s)
    {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < s.Length; i++) {
            if (Char.IsNumber(s[i]))
                sb.Append(s[i]);
        }
        return sb.ToString();
    }
}
```

Creation of Extension Methods

- Using the RemoveNonNumeric() method in StringExtensions using StringExtensions;

- string phone = "123-123-1234";
  - string newPhone = phone.RemoveNonNumeric();

Demo

- Defining and using methods in C#, Using various Parameters types, and Extension methods

What Are Object Initializers?

- An object initializer is used to assign values to an object fields or properties when the object is created.
- There is no need to explicitly invoke a constructor
- It combines object creation and initialization in a single step.

What Are Object Initializers?

```
public class Customer
{
        public string CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

         public Customer(int ID)
         {
              CustomerID = ID;
         }
}

Customer c = new Customer(1) { Name = "Maria Anders",
                               City = "Berlin" };
```

Anonymous Types

- Implicit type functionality for objects
- Set property values into an object without writing a class definition.
- The resulting class has no usable name
- The class name is generated by the compiler
- The created class inherits from Object
- The result is an ‘anonymous’ type that is not available at the source code level.
- It is also called as “Projections”

Use Of Anonymous Types

- Anonymous types enables developers to concisely define inline CLR types within code, without having to explicitly define a formal class declaration of the type.
- To create an anonymous type, the new operator is used with an anonymous object initializer.
  - Example:
  - var person = new { Name = “John Doe”, Age = 33 };
- C# compiler automatically creates a new type that has two properties: one called Name of type string, and another called Age having type int.

Instances Of Anonymous Types

- Two anonymous object initializers that specify a sequence of properties of the same names and types in the same order produce instances of the same anonymous type.

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

What Are Namespaces?

- A namespace defines a declarative region that provides a way to keep one set of names separate from another.
- Thus, names declared in one namespace will not conflict with the same names declared in another.
- A namespace is declared using the namespace keyword.
- The general form of namespace is shown here:

```
namespace name
{
// members
}
```

Use Of Partial Types

- Partial types allow classes, structs, and interfaces to be broken into multiple pieces stored in different source files for easier development and maintenance.
- Additionally, partial types allow separation of machine-generated and user-written parts of types so that it is easier to augment code generated by a tool.
- A new type modifier, partial, is used while defining a type in multiple parts.

Customer Class In Two Partial Classes

```
public partial class Customer
{
        private int id;
        private string name;
        private string address;
        private List<Order> orders;
        public Customer()
        {...}
}
```

```
public partial class Customer
{
    public void SubmitOrder(Order order)
        {
        orders.Add(order);
        }
        public bool HasOutstandingOrders ()
        {return orders.Count > 0;}
}
```

Customer Class After Compilation

```
public class Customer
{
      private int id;
      private string name;
      private string address;
      private List<Order> orders;

      public Customer() { }

      public void SubmitOrder(Order order)
      {          orders.Add(order);          }

      public bool HasOutstandingOrders()
      {          return orders.Count > 0;   }
 }
```

Demo

- Defining and using classes in C#

Summary

- In this lesson, you learned
  - How to create a class in C#?
  - Different Access Modifiers in C#
  - What are method parameters in C#? (ref, out and params)
  - Structures and its distinction from classes
  - How to use inheritance in C#?
  - What are properties and Indexers and how to use them?
  - Function Overriding in C#
  - Abstract Class, Abstract Method and a Sealed Class
  - What is an interface and how it is different from an abstract Class?

Review Question

- How is class different from a structure?
- Why is class called as a “is-a” relationship?
- What are the different access specifiers in C#?
- What is a Sealed Class in C#?
- Can abstract class be sealed?
- How is abstract class different from an interface?
- How is function overriding implemented in C#?
- What are the different method parameters in C#?