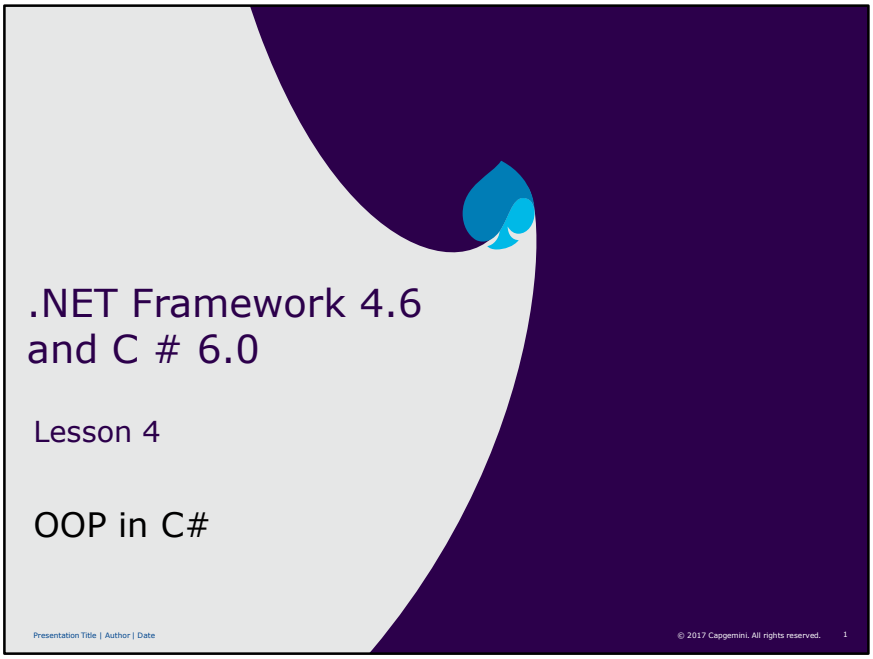

.NET Framework 4.6
and C # 6.0
Lesson 4
OOP in C#
Presentation Title | Author | Date
© 2017 Capgemini. All rights reserved.
1

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 a the Abstract Class and Sealed Class
- Interfaces in C#
- Difference between Interfaces in C# and the Abstract Class
- Method Parameters in C#

4.1: Classes

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

Classes:

- **Template**: A class is a template that defines the form of an object. It specifies both the data and the code that will operate on that data. C# uses a class specification to construct objects. Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object.
- **Logical Construction**: A class is a logical abstraction. Only after an object of that class has been created, a physical representation of that class exists in memory.
- **The Data and Code**: When you define a class, you declare the data that it contains and the code that operates on it. While very simple classes might contain only code or only data, most real-world classes contain both. In general terms, data is contained in data members defined by the class, and code is contained in function members. It is important to state at the outset that C# defines several specific types of data and function members.
- For example, data members (also called fields) include: instance variables and static variables.
- Function members include:
  - Methods
  - Constructors
  - Destructors
  - Indexers
  - Events
  - Operators
  - Properties

Class Example

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

Classes (Contd.):

- A class is created using the keyword **class**. Given here is the general form of a simple class definition that contains only instance variables and methods.
- Notice that each declaration of variable or method is preceded with access. Here, **access** is an access specifier, such as **public**, which specifies how the member can be accessed. The access specifier determines what type of access is allowed. The access specifier is optional and if absent, then the member is private to the class. Members with private access can be used only by other members of their class.

4.2: Constructors

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).

Constructors:

A constructor initializes an object when it is created. Constructor has the same name as its class and is syntactically similar to a method. However, constructor has no explicit return type. The general form of constructor is shown here:

access class-name( ) { // constructor code }

Typically, you use a constructor to give initial values to the instance variables defined by the class. Also, you use it to perform any other startup procedures required to create a fully formed object. Usually, access is **public** because constructors are normally called from outside their class.

All classes have constructors, whether you define one or not. The reason is, C# automatically provides a default constructor that initializes all member variables to **zero** (for value types) or **null** (for reference types). However, once you define your own constructor, the default constructor is no longer used.

4.2: Constructors

Constructor Example

```
public class Employee
{
      static Employee()  // static constructor
      {. . .}
      public Employee() // default constructor
      {. . .}
      public Employee(string name) // parameterized
constructor
      {. . .}
}
```

Example of a Constructor:

Static constructor is called only once before any object of that class gets instantiated.

Constructor Example

```
//Creating object of Class
public class Program
{
        static void Main()
        {
           Employee emp = new Employee();
        }
}
```

4.3: Methods

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:

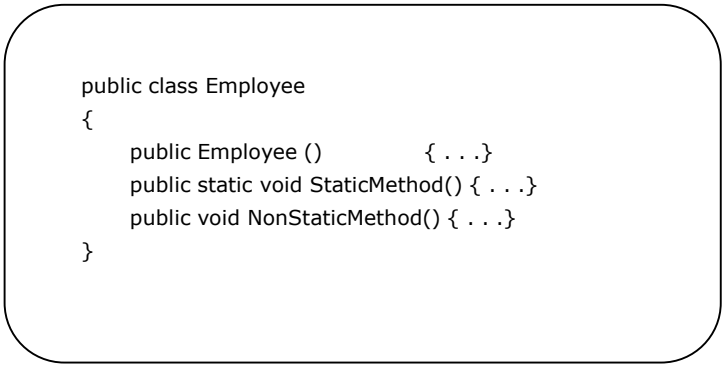
[attributes]
[method-modifiers] return-type method-name-identifier ([formal-parameter-list])
{
[statements]
}

There are four kinds of parameters:

- out
- ref
- params
- value.

value is a default parameter

Example of a Class Method



```
public class Employee
{
      public Employee ()            { . . .}
      public static void StaticMethod() { . . .}
      public void NonStaticMethod() { . . .}
}
```

Class Method - Example :

Methods are subroutines that manipulate the data defined by a class; and, in many cases, provide access to that data. Typically, other parts of your program interacts with a class through its methods.

The general form of a method is shown here:

```
access ret-type name(parameter-list)
 {
 // body of method
 }
```

A method contains one or more statements. In a well-written C# code, each method performs only one task.

Each method has a name, and this name is used for calling the method.

Example of a Class Method

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

4.3: Methods

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 Value Parameters:

The **value** parameter is the default parameter type in C#.

If a parameter does not have any modifier, it is the **value** parameter by default.

When you use the **value** parameter, the actual value is passed to the function.

This means, changes made to the parameter are local to the function and are not passed back to the calling part.

The Ref Parameter

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

- Output of the above program would be 105, since the ref parameter acts as both input and output.

The Ref Parameter:

The **ref** parameters are input/output parameters.

That means, they can be used for passing a value to a function as well as for getting back a value from a function.

We create a **ref** parameter by preceding the parameter data type with a **ref** modifier.

Whenever, a **ref** parameter is passed, a reference is passed to the function.

The Out Parameter

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

- Output of the above program is 100, since the value of the out parameter is passed back to the calling part.

The Out Parameter:

The **out** parameters are 'output only' parameters. That means, they can only return a value from a function.

We create an **out** parameter by preceding the parameter data type with an **out** modifier.

Whenever an **out** parameter is passed only an unassigned reference is passed to the function.

The **out** modifier should precede the parameter being passed even in the calling part.

The **out** parameters cannot be used within the function before assigning a value to it.

A value should be assigned to the **out** parameter before the method returns.

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

The Params Parameter:

The value passed for a **params** parameter can be either a comma-separated value list or a single dimensional array.

The **params** parameters are 'input only'.

4.4: Method Overloading

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.

Method Overloading and Polymorphism:

In general, to overload a method, we declare different versions of it. The compiler takes care of the rest.

We must observe one important restriction: the type and/or number of the parameters of each overloaded method must be unique.

It is not sufficient for two methods to differ only in their return types. They must differ in the types or numbers of their parameters. (Return types do not provide sufficient information in all cases for C# to decide which method to use.)

Of course, overloaded methods may differ in their return types also.

When an overloaded method is called, the version of the method executed is the one which has parameters that match the arguments.

Overloading Of Constructors

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

4.6: Static Members

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

Creation of Static Members:

Sometimes, we need to define a class member that can be used independent of any object of that class.

Normally, a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific instance.

To create such a member, precede its declaration with the keyword **static**.

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

What is a static constructor?:

A static constructor is called automatically; it is called before the instance constructor is called.

In all the cases, the static constructor is executed before any instance constructor.

Furthermore, static constructors cannot have access modifiers (thus, they use default access) and cannot be called by your program.

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.

4.9: Properties

Example Properties

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

Example of Properties:

In this example, **month** is declared as a property so that the **set** accessor can make sure that the **month** value is set between **1** and **12**. The **month** property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property.

4.9: Properties

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;  }
        }
 // ....
}
```

Asymmetric Accessor Accessibility:

There are a few restrictions to this asymmetric accessor accessibility feature in C# 2.0:

- You can only set a different visibility on one of the two accessors, not both. Trying to restrict visibility on both **get** and **set** to **internal** gives an error:
- You cannot use the feature on properties that do not have BOTH **get** and **set** accessors. Removing the **get** accessor from the example above gives you error:
- The accessibility modifier used on a **get** or **set** accessor can only restrict visibility not increase it.

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:

Often, property accessors (**get** and **set**) have trivial implementations and follow the pattern that simply get (return) a private field and set the private field to the value passed.

In the following example, the **Point** class contains two properties:

```
public class Point
{
   private int x;
   private int y;
   public int X { get { return x; } set { x = value; } }
   public int Y { get { return y; } set { y = value; } }
}
```

To simplify this, let the compiler generate the field and you can simply specify the property name as shown below:

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

4.9: Properties

Use Of Auto-Implemented Properties

- Auto-implemented properties must declare both a get and a set accessor.

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

- To create a 'read only' auto-implemented property, use a private set accessor.

Use of Auto-Implemented Properties (Contd.):

Another example:

```
class LightweightCustomer
{
        public double TotalPurchases { get; set; }
        public string Name { get; private set; } // read-only
        public int CustomerID { get; private set; } // read-only
}
```

Example Auto-Implemented Properties

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

In the above example, an error occurs when you attempt to set the **CustomerID** property directly. This happens due to the private modifier on set. The **CustomerID** property now behaves as if it were read-only.

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#

4.11: Class Inheritance

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

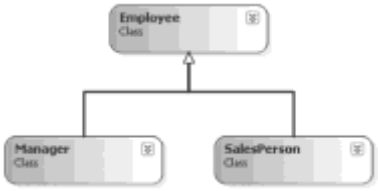
What is class inheritance?

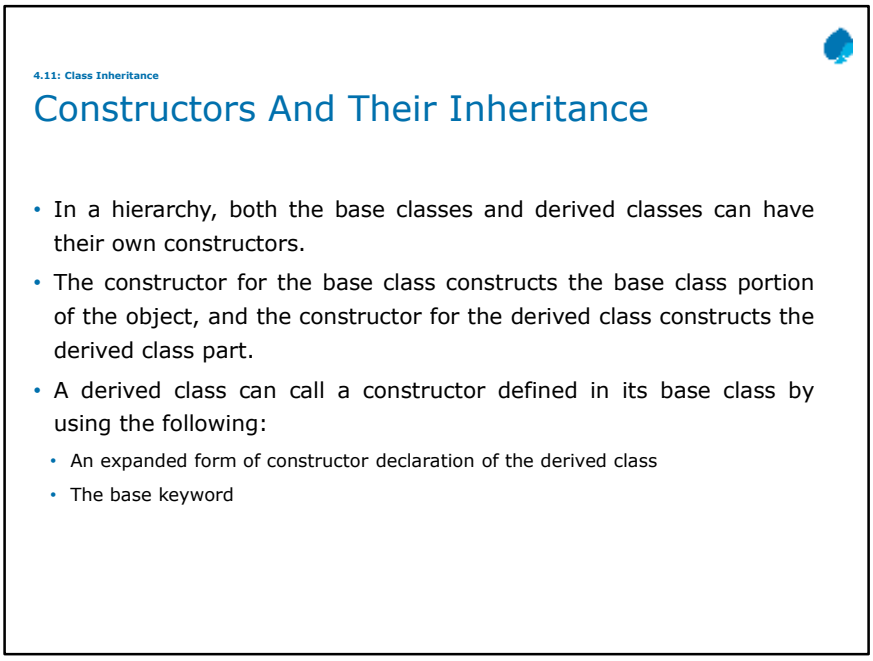
Inheritance is one of the three foundational principles of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding the things that are
unique to it.

In the language of C#, a class that is inherited is called a base class. The class that performs the inheriting is called a derived class. Therefore, a derived class is a specialized version of a base class. It inherits all of the variables, methods, properties, operators, and indexers defined by the base class and adds its own unique elements.

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. This raises an important question: which constructor is responsible for building an object of the derived class? The one in the base class, the one in the derived class, or both? The answer is: 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. The reason is, the base class has no knowledge of or access to any element in a derived class. Thus, their construction is separate.

A derived class can call a constructor defined in its base class by using an expanded form of the derived class' constructor declaration and the base keyword. The general form of this expanded declaration is as follows:

```
derived-constructor(parameter-list) : base(arg-list) {
 // body of constructor
}
```

Here, arg-list specifies the arguments that are needed by the constructor defined in the base class.

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.

Hiding the Name of Base Class Member:

Understand that this use of the **new** keyword is separate and distinct from its use while creating an object instance.

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.

Reference of Derived Object to Base Variable:

C# is a strongly typed language. The standard conversions and automatic promotions apply to all its value types. However, values are essentially compatible with their class types (type compatibility). Therefore, a reference variable for one class type cannot normally refer to an object of another class type.

There is, however, an important exception to C#'s strict type enforcement. A reference variable of a base class can be assigned a reference to an object of any class derived from that 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);
  } ... }
```

4.13: Virtual Methods

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.

Virtual Methods -Declaration and Redefinition:

Virtual methods are interesting because of what happens when one is called through a base class reference. In this situation, C# determines which version of the method to call based upon the type of the object referred to by the reference—and this determination is made at runtime. Thus, when different types of objects are referred to, different versions of the virtual method are executed.

4.14: Abstract Classes

What Is An Abstract Class?

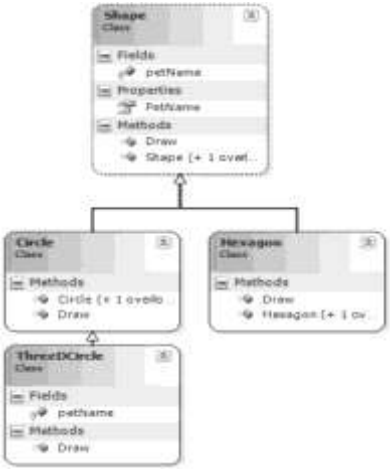
- An abstract class is the one that cannot be instantiated
- It is intended to be used as a base class
- It may contain abstract and non-abstract function members
- It cannot be sealed

What is an abstract class?

Sometimes, you may require to create a base class that defines only a generalized form that can be shared by all of its derived classes, each derived class filling in its specific details. Such a class determines the nature of the methods that the derived classes must implement, but does not, by itself, provide an implementation of one or more of these methods. Such classes are declared as the abstract classes.

4.14: Abstract Classes

Example



4.14: Abstract Classes

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

Characteristics of Abstract Methods:

When a class has been defined as an abstract base class, it may define any number of abstract members (which is analogous to a C++ pure virtual function).

Abstract methods can be used whenever you wish to define a method that does not supply a default implementation.

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

4.15:: Sealed Class

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

Characteristics of Sealed Class:

Inheritance is powerful and useful; however, sometimes you require to prevent it. For example, you might have a class that encapsulates the initialization sequence of some specialized hardware device, such as a medical monitor. In this case, you don't want users of your class to be able to change the way the monitor is initialized, setting the device incorrectly. Whatever the reason, in C#, it is easy to prevent a class from being inherited by using the keyword **sealed**.

Sealed Class Example

```
using System;
sealed class MyClass
   {   public int x;    public int y;}
// class MainClass
 class MainClass: MyClass {  }  causes
error
```

4.16:Interfaces

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.

What are interfaces?:

In object-oriented programming, it is sometimes helpful to define what a class must do, but not how it will do it. You have already seen an example, the abstract method. While abstract classes and methods are useful, it is possible to take this concept a step further. In C#, you can fully separate specification of interface of a class from its implementation by using the keyword **interface**.

Interfaces are syntactically similar to abstract classes. However, in an interface, no method can include a body. That is, an interface provides no implementation whatsoever. It specifies what must be done, but not how. Once an interface is defined, any number of classes can implement it. Also, one class can implement any number of interfaces.

4.16: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).

Implementation of Interfaces:

To implement an interface, a class must provide bodies (implementations) for the methods described by the interface. Each class is free to determine the details of its own implementation. Thus, two classes might implement the same interface in different ways, but each class still supports the same set of methods. Therefore, code that has knowledge of the interface can use objects of either class since the interface to those objects is the same. By providing the interface, C# allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.

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();
}
```

4.16:Interfaces

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.

Differences: Abstract Classes and Interface

One of the most challenging parts of C# programming involves ascertaining when to create an interface and when to use an abstract class when you want to describe functionality but not implementation. The general rule is: When you can fully describe the concept in terms of "what it does" without needing to specify any "how it does it," then you should use an interface. If you need to include some implementation details, then you need to represent your concept using an abstract class.

Demo

- Creating and using abstract classes and Interfaces in C#

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
```

4.18:Extension Methods

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.

The Extension Methods:

Extension methods are static methods that can be invoked using instance method syntax. In effect, extension methods make it possible to extend existing types and constructed types with additional methods.

Extension methods allow existing classes to be extended without relying on inheritance or having to change the class's source code. This means that if you want to add some methods into the existing String class you can do it quite easily.

Declaring Extension Methods:

Extension methods are declared by specifying the keyword this as a modifier on the first parameter of the methods. Extension methods can only be declared in static classes.

4.18:Extension Methods

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.

```
public static class Utility
  {

     public static int WordCount(this string sentence)
   {

       return sentence.Split(new char[] { ' ' }).Length;

   }
  }

  class UtilityTest
  {
    static void Main(string[] args)
    {
      string testString = "CAPGEMINI LnD";
      Console.WriteLine(testString.WordCount());
    }
  }
```

Use of Extension Methods – Restrictions

The following are a couple of rules to consider when deciding on whether or not to use extension methods:

- 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.
- Use extension methods sparingly.

4.18:Extension Methods

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 Method - Example :

The following is an example of creating an extension method in C# that adds a **RemoveNonNumeric()** method to a **String** class. Notice that the class is defined as static as well as the extension method itself. The **this** keyword in the parameter signature tells the compiler to add the extension method to the **String** class since **String** follows the keyword.

```
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();
                }
        }
}
```

4.18:Extension Methods

Creation of Extension Methods

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

string phone = "123-123-1234";

- string newPhone = phone.RemoveNonNumeric();

Use of Extension Methods - Example:

This is an example of how an extension method can be used. You'll see that the namespace for the extension method class is imported. From there, the compiler treats the **RemoveNonNumeric()** method as if it was originally a part of the standard **System.String** class.

```
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

4.19: Object Initializers

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?

From C# 3.0 onwards, when declaring an object or collection, you may include an initializer that specifies the initial values of the members of the newly created object or collection. This new syntax combines object creation and initialization in a single step.

Using Object Initializers

An object initializer consists of a sequence of member initializers, enclosed by **{and}** tokens and separated by commas. Each member initializer must name an accessible field or property of the object being initialized, followed by an 'equal to' (=) sign and an expression or an object or collection initializer. It is an error for an object initializer to include more than one member initializer for the same field or property. For example

```
public class Point
{
   public int X { get; set; }
   public int Y { get; set; }
}
```

4.19: Object Initializers

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" };
```

Example - 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;
        }
        public override string ToString()
        {
                return Name + "\t" + City + "\t" +
      CustomerID;
        }
}
```

4.20: Anonymous Types

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"

Anonymous Types:

From C# 3.0 onwards the new operator can be used with an anonymous object initializer to create an object of an anonymous type.

Specifically, an anonymous object initializer of the following form declares an anonymous type of the form.

new { p1 = e1 , p2 = e2 , … pn = en }

To facilitate the creation of classes from data values, C# 3.0 and higher versions provide the ability to easily declare an anonymous type and return an instance of that type. To create an anonymous type, the new operator is used with an anonymous object initializer. For example, when presented with the following declaration, the C# compiler automatically creates a new type that has two properties: one, called **Name** of the type string, and another called **Age** having the **int** type:

var person = new { Name = "John Doe", Age = 33 };

Each member of the anonymous type is a property inferred from the

object initializer. The name of the anonymous type is automatically generated by the compiler and cannot be referenced from the user code.

4.20: Anonymous Types

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.

When to use Anonymous Types

•Need a temporary object to hold related data
•Don't need methods
•When there is a need for different set of properties for each declaration
•When there is a need to change the order of the properties for each declaration.

When Not to use Anonymous Types

•There is a need to define a method
•There is a need to define another variable.
•There is a need to share data across methods

4.20: Anonymous Types

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;
```

Instances of Anonymous Types:

Within the same program, 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.

In the example,

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

The assignment on the last line is permitted because p1 and p2 are of the same anonymous type.

4.20: Namespaces

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
}
```

Namespaces:

Namespaces are important because there has been an explosion of variable, method, property, and class names over the past few years. These include library routines, third-party codes, and your own code. Without namespaces, all of these names would compete for slots in the global namespace and conflicts would arise. For example, if your program defines a class called **Finder**, it can conflict with another class called **Finder** supplied by a third-party library that your program uses. Fortunately, namespaces prevent this type of problem, because a namespace localizes the visibility of names declared within it.

A namespace is declared using the **namespace** keyword. The general form of namespace is shown below:

namespace name { // members }

Here, **name** is the name of the namespace. Anything defined within a namespace is said to be within the scope of that namespace. Thus, namespace defines a scope. Within a namespace, you can declare classes, structures, delegates, enumerations, interfaces, or another namespace.

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#

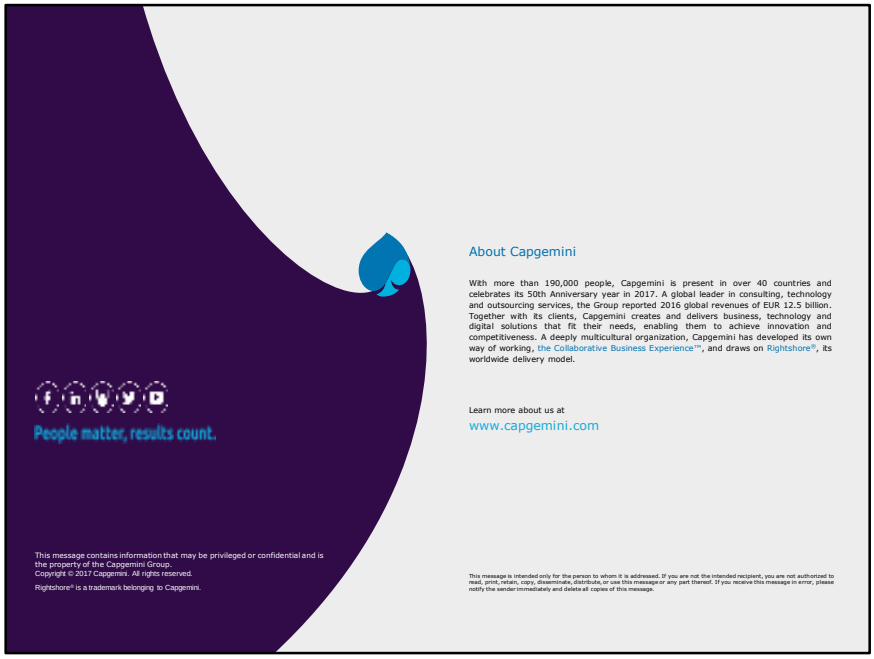
Summary

- 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

1. How is class different from a structure?
2. Why is class called as a "is-a" relationship?
3. What are the different access specifiers in C#?
4. What is a Sealed Class in C#?
5. Can abstract class be sealed?
6. How is abstract class different from an interface?
7. How is function overriding implemented in C#?
8. What are the different method parameters in C#?

1. Class is a reference type where as struct is a value type. Class can inherit from any non sealed reference type where as struct can inherit only from System.ValueType. Class can have default constructor, parameterized constructor and destructor. Struct can have only parameterized constructor and destructor.
2. Because of relationship between base class and derived class.
3. public, private, protected, internal, protected internal.
4. A class that cannot be inherited.
5. No.
6. 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.
7. Declare method as virtual in base class.
   Provide own implementation to the method in derived class by using
   the keyword override.
8. ref, out and params.


People matter, results count.
This message contains information that may be privileged or confidential and is the property of the Capgemini Group.
Copyright © 2017 Capgemini. All rights reserved.
Rightshore® is a trademark belonging to Capgemini.
About Capgemini
With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.
Learn more about us at
www.capgemini.com
This message is intended only for the person to whom it is addressed. If you are not the intended recipient, you are not authorized to read, print, retain, copy, disseminate, distribute, or use this message or any part thereof. If you receive this message in error, please notify the sender immediately and delete all copies of this message.