OBJECT ORIENTED PROGRAMMING

LAB MANUAL

TABLE OF CONTENTS

TOPIC NO. 1

DOT NET ARCHITECTURE AND OBJECT ORIENTED PROGRAMMING

What is the .NET Framework?

The .NET Framework is a new and revolutionary platform created by Microsoft for developing applications.

It is a platform for application developers.

It is a Framework that supports Multiple Language and Cross language integration.

It has an IDE (Integrated Development Environment).

Framework is a set of utilities or can say building blocks of your application system.

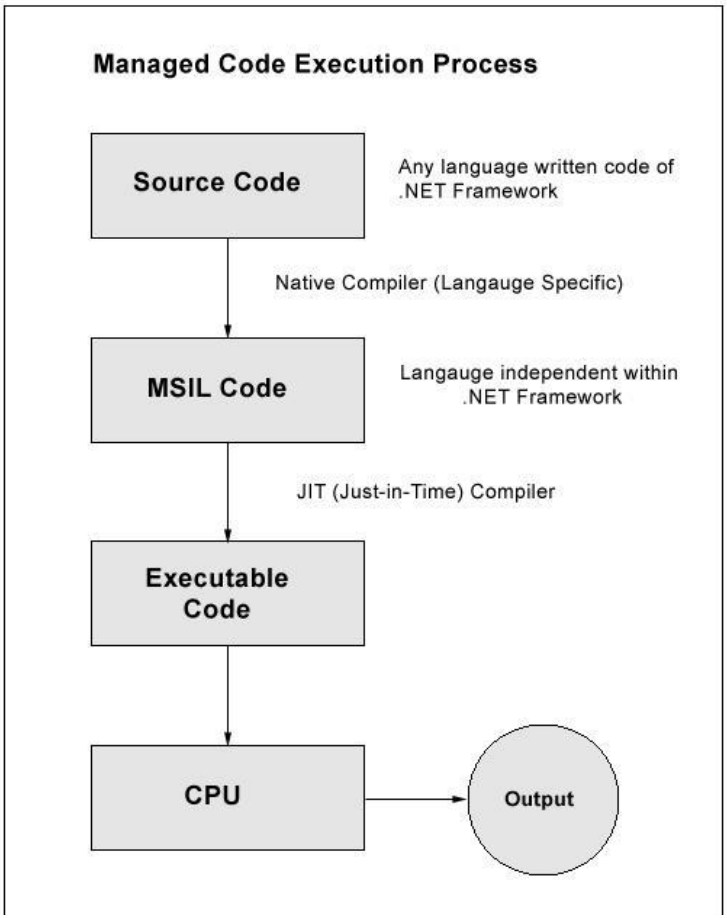
.NET Framework provides GUI in a GUI manner.

.NET is a platform independent but with help of Mono Compilation System (MCS). MCS is a middle level interface.

.NET Framework provides interoperability between languages i.e. Common Type System (CTS).

.NET Framework also includes the .NET Common Language Runtime (CLR), which is responsible for maintaining the execution of all applications developed using the .NET library.

The .NET Framework consists primarily of a gigantic library of code.



Definition: A programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications and Web services.

Object Oriented Programming (OOP): Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on a computer desktop (such as buttons and scroll bars).

The first step in OOP is to identify all the objects the programmer wants to manipulate and how they relate to each other, an exercise often known as data modeling. Once an object has been identified, it is generalized as a class of objects (think of Plato's concept of the "ideal" chair that stands for all chairs) which defines the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method. Objects communicate with well-defined interfaces called *messages*.

The concepts and rules used in object-oriented programming provide these important benefits:

The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Called inheritance, this property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding.

Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding provides greater system security and avoids unintended data corruption.

The definition of a class is reusable not only by the program for which it is initially created but also by other object-oriented programs (and, for this reason, can be more easily distributed for use in networks).

The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.

TOPIC NO. 2
CLASSES AND OBJECTS

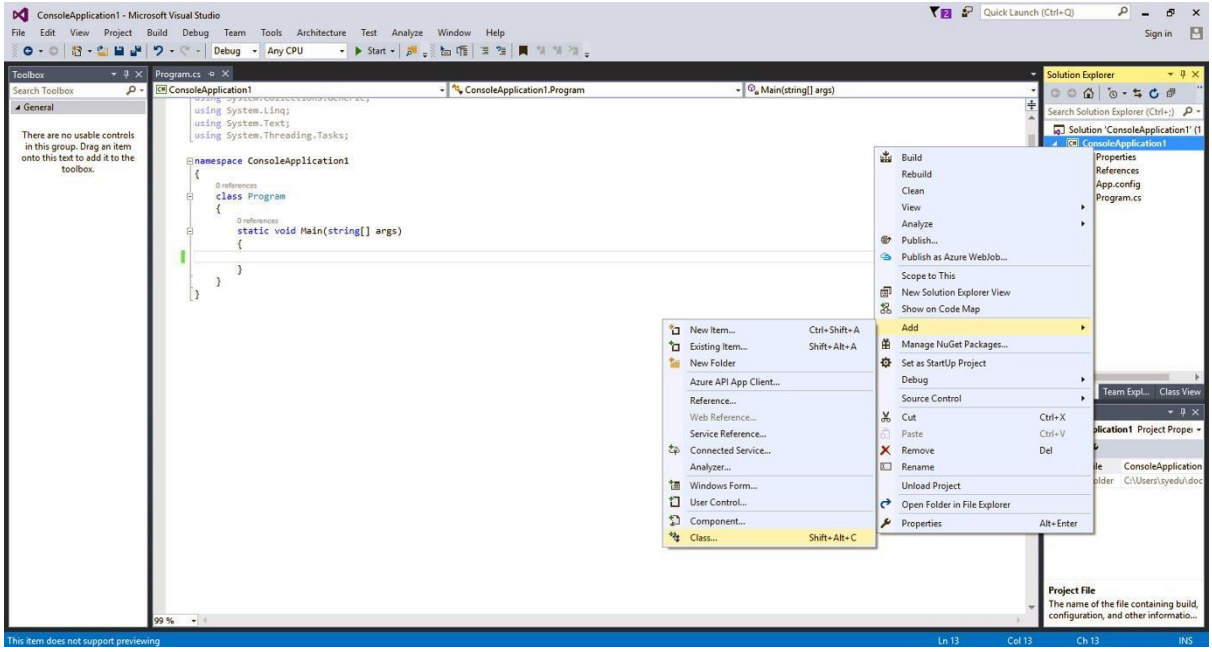
What is a Class?

A class is simply an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on data (methods) and accessors to data (properties). For example, there is a class String in the System namespace of .Net Framework Class Library (FCL). This class contains an array of characters (data) and provide different operations (methods) that can be applied to its data like ToLowerCase(), Trim(), Substring(), etc. It also has some properties like Length (used to find the length of the string). A class in C# is declared using the keyword class and its members are enclosed in parenthesis.

How a Class is created in Visual Studio?

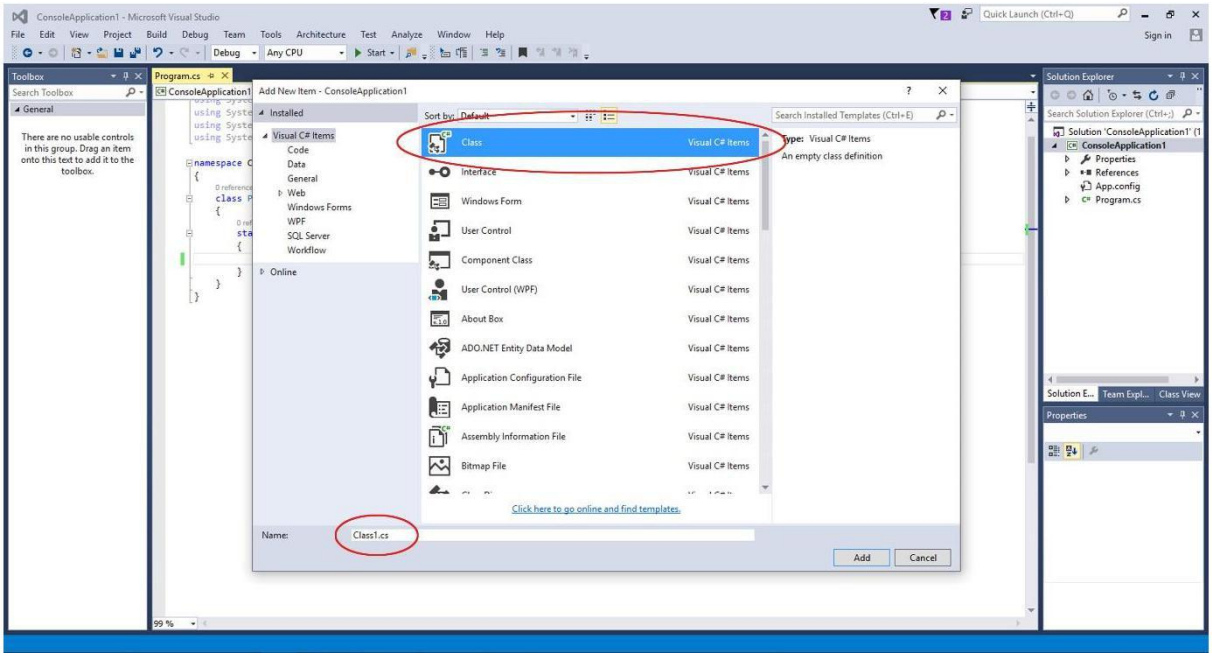
Open a new project/program. Here in this example it's named with 'ConsoleApplication1'. Right click on the named application (let say ConsoleApplication1) as shown in the solution explorer on the right panel. Then click on 'Add' and click on 'Class…'

Or either you can create a new class with a short cut key (Shift+Alt+A).

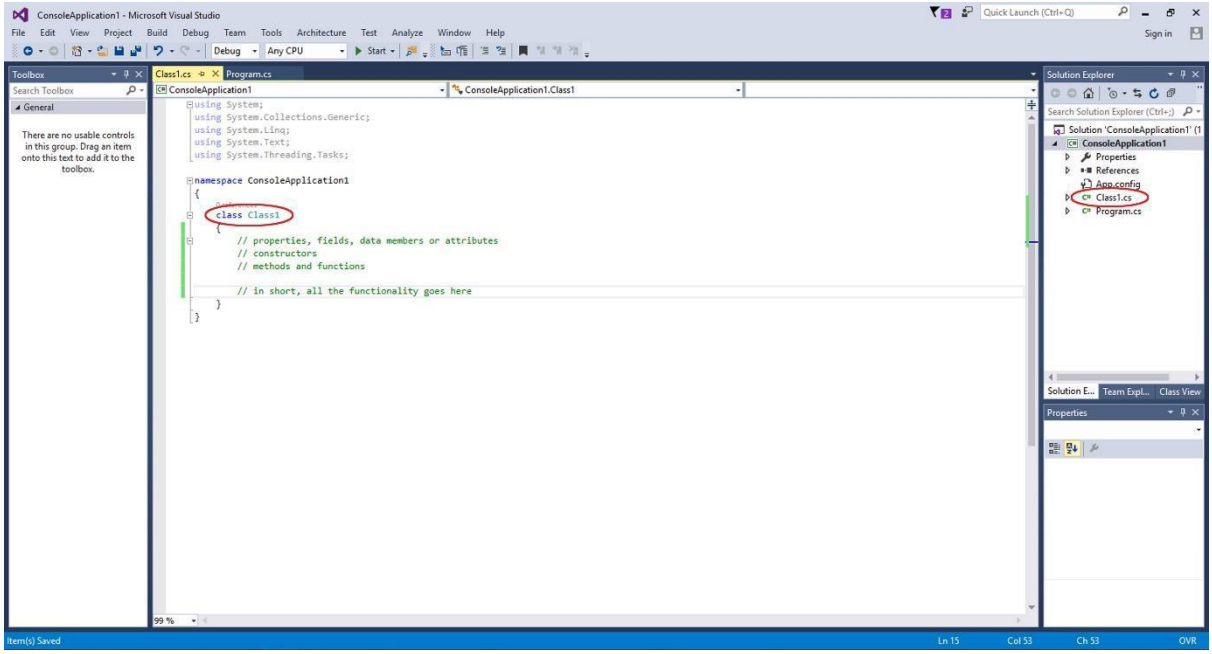


How to name a Class in Visual Studio?

When you click on the 'Class…' option, a new window will appear. On the bottom panel, by default Class1.cs would be written. You can rename it as according to your requirement. Image is shown on the next page.



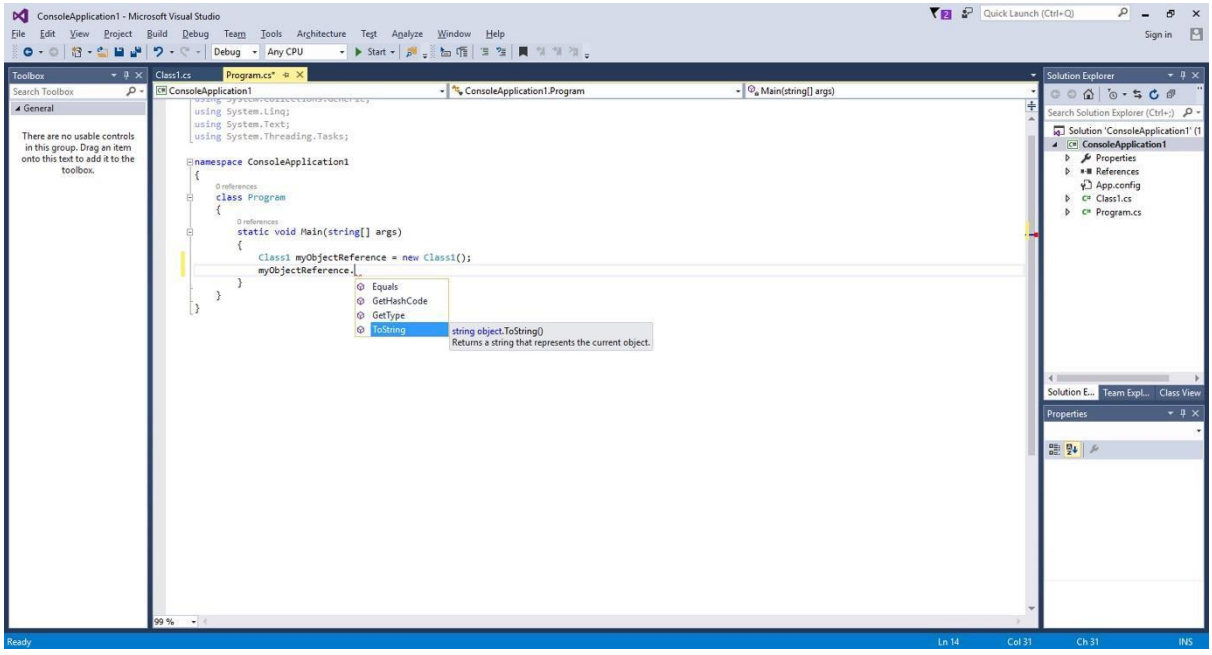
Hence, a new class would be added in the same namespace / project or application.



Here Class1 is the name of this class in this scenario. All the functionality goes in this class related to a particular class. As mentioned above, a class contains (fields, properties, attributes, data members), (methods and functions) and (constructors).

What is an Object?

As mentioned above, a class IS an abstract model. An object is the concrete realization or instance built on the model specified by the class. An object is created in the memory using the keyword 'new' and is referenced by an Identifier called a "reference". A picture is shown below in which object/instance of Class1 is created in Program class. By using the instance, we can use the entire functionality of Class1 in Program class.




```
Class1 myObjectReference = new Class1();
```

In the line above, we made an object of type Class1 which is referenced by an identifier **myObjectReference.**

The difference between classes and implicit data types is that objects are reference types (passed by reference) while implicit data types are value type (passed by making a copy). Also, objects are created at the heap while implicit data types are stored on stack.

* The object can be created anywhere i.e. in any class in the same namespace. All the functionality of a particular class can be used anywhere in the same namespace by using the object of a particular class.

TOPIC NO. 3

METHODS / FUNCTIONS / SUBROUTINES

What is a Method / Function or a Subroutine?

In object-oriented programming, a method is a subroutine (or procedure or function) associated with a class.

A method, function or a subroutine is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

To use a method, you need to:

Define the method

Call the method

Defining Methods in C#

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows:

```
<Access Specifier> <Return Type> <Method Name> ( Parameter List )
{
  Method Body
}
```

Following are the various elements of a method:

Access Specifier: This determines the visibility of a variable or a method from another class.

Return type: A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.

Method name: Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.

Parameter list: Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

Method body: This contains the set of instructions needed to complete the required activity i.e. the entire code for functionality of that particular method.

For instance:

Following code snippet shows a function ***FindMax*** that takes two integer values and

returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator
{
  public int FindMax(int num1, int num2)
  {
    int result;


    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
  }
  .....
}
```

Calling Methods in C#

You can call a method using the name of the method. The following example illustrates this:

```
using System;
namespace CalculatorApplication
{
  class Program
  {
     static void Main(string[] args)
    {
      int a = 100;
      int b = 200;
      int result;


      NumberManipulator nm = new NumberManipulator();


      //calling the FindMax method
      result = nm.FindMax(a, b);
      Console.WriteLine("Max value is : {0}", result);
      Console.ReadLine();
    }
  }
}
```

TOPIC NO. 4
CONSTRUCTORS AND ITS TYPES

What is a Constructor?

A constructor is responsible for preparing the object for action, and in particular establishing initial values for all its data, i.e. its data members. Although it plays a special role, the constructor is just another member function, and in particular can be passed information via its argument list that can be used to initialize it. The name of the constructor function is the name of the class.

OR

A special method of the class that will be automatically invoked when an instance of the class is created is called a constructor. The main use of constructors is to initialize private fields of the class while creating an instance for the class. When you have not created a constructor in the class, the compiler will automatically create a default constructor in the class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

Some of the key points regarding the Constructor are:

A class can have any number of constructors.

A constructor doesn't have any return type, not even void.

A static constructor cannot be a parametrized constructor.

Within a class you can create only one static constructor.

TYPES OF CONSTRUCTORS

Constructors can be divided into 5 types:

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

1. DEFAULT CONSTRUCTOR

A constructor without any parameters is called a default constructor; in other words this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class to different values. The default constructor initializes:

All numeric fields in the class to zero.

All string and object fields to null.

Example

```
using System;
namespace DefaultConstructor
{
   class addition
   {
      int a;
      int b;

      //default contructor
      public addition( )
      {
         a = 100;
         b =
         175;
      }

      public static void Main( )
      {
         //an object is created , constructor is called
         addition obj = new addition( );
         Console.WriteLine(obj.a);
         Console.WriteLine(obj.b);
         Console.Read( );
      }
   }
}
```

2. PARAMETERIZED CONSTRUCTOR

A constructor with at least one parameter is called a parametrized constructor. The advantage of a parametrized constructor is that you can initialize each instance of the class to different values.

Example

```
using System;
namespace Constructor
{
   class ParaConstructor
   {
     public int a;
     public int b;

     public ParaConstructor ( int x, int y )
     {
         a = x;
         b = y;
     }
 }
```

```
  class MainClass
  {
     static void Main( )
     {
        ParaConstructor pc = new ParaConstructor(100, 175);
        Console.WriteLine("Value of a = " + pc.a );
        Console.WriteLine("Value of b = " + pc.b);
        Console.Read( );
     }
  }
}
```

3. COPY CONSTRUCTOR

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

Syntax

```
public employee(employee emp)
{
      name = emp.name;
      age = emp.age;
}
```

The copy constructor is invoked by instantiating an object of type employee and adding it to the object to be copied.

Example

employee emp1 = new employee (emp2);
Now, emp1 is a copy of emp2.

So let us see its practical implementation.

```
using System;
namespace CopyConstructor
{
  class employee
  {
     private string name;
     private int age;
     public employee(employee emp)  // declaring Copy constructor.
     {
        name = emp.name;
        age = emp.age;
     }
     public employee(string name, int age) // Instance constructor.
     {
        this.name = name;
        this.age = age;
     }
```

```
    public string Details  // Get deatils of employee
    {
      get
      {
        return " The age of " + name + " is " + age.ToString( );
      }
    }
  }
  class empdetail
  {
    static void Main( )
    {
      employee emp1 = new employee("", 22); // Create a new employee object. employee
      emp2 = new employee(emp1); // here is emp1 details is copied to emp2.
      Console.WriteLine(emp2.Details);
      Console.ReadLine( );
    }
  }
}
```

4. STATIC CONSTRUCTOR

When a constructor is created as static, it will be invoked only once for all of instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Some key points of a static constructor is:

A static constructor does not take access modifiers or have parameters.

A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.

A static constructor cannot be called directly.

The user has no control on when the static constructor is executed in the program.

A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

Syntax

```
class employee
{   //   Static   constructor
        static employee( )
        {
        }
 }
```

Example

```
using System;
namespace StaticConstructor
{
    public class employee
    {
        static employee( )
    }
    public static void Slary( )
    {
        Console.WriteLine("The Salary method");
    }
}
class details
{
  static void Main( )
  {
    employee.Salary( );
  }
 }
}
```

5. PRIVATE CONSTRUCTOR

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in classes that contain static members only.

Some key points of a private constructor are:

One use of a private constructor is when we have only static members.

It provides an implementation of a singleton class pattern

Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.

Example

```
using System;
namespace DefaultConstructor
{
  public class Counter
  {
    private Counter( )  //private constructor declaration
    {
    }
    public static int currentview;
```

```
        public static int visitedCount( )
        {
            return ++ currentview;
        }
    }
    class viewCountedetails
    {
        static void Main( )
        {
            Counter.currentview = 500;
            Counter.visitedCount( );
            Console.WriteLine("Now the view count is: {0}", Counter.currentview);
            Console.ReadLine();
        }
    }
}
```

TOPIC NO. 5
CONSTRUCTOR OVERLOADING

What is Constructor Overloading?

Multiple constructors with the same class name but different number and types of parameters is referred as Constructor Overloading.

OR

When there are more than one or multiple constructors with the same name is defined inside the same class but their parameter list is different from one another, these constructors are called overloaded and this situation is called constructors overloading.

Some Key points:

In constructors overloading each constructor's parameter list must be different from one another by their type, number or sequence of parameters.

Overload constructors allow us to create many different forms / shapes of a same type object. These objects will be same in type but totally different from one another's.

Overload constructors make a class flexible for creating multiple objects.

Example

```
using System;
namespace ConsoleApplication1
{
    class Shape
    {
        double  lenght;
        double  bredth;
        double  height;
        double radius;

        public Shape()
        {
        }

        public Shape(double _radius)
        {
            this.radius = _radius;
        }

        public Shape(double _length, double _bredth)
        {
            this.lenght  =  _length;
            this.bredth = _bredth;
        }

        public Shape(double _length, double _height, double _bredth)
        {
            this.lenght  =  _length;
            this.height  =  _height;
            this.bredth = _bredth;
        }
    }
```

```
}
```

TOPIC NO. 6
ACCESS MODIFIERS

What is an Access Modifier?

Access modifiers are keywords used to specify the declared accessibility of a member or a type.

Why to use access modifiers?

Access modifiers are an integral part of object-oriented programming. They support the concept of encapsulation, which promotes the idea of hiding functionality. Access modifiers allow you to define who does or doesn't have access to certain features.

In C# there are 5 different types of Access Modifiers.

MODIFIER	DESCRIPTION
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified
protected	Access is limited to within the class definition and any class that inherits from the class
internal	Access is limited exclusively to classes defined within the current project assembly
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.

PUBLIC ACCESS SPECIFIER

Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

PRIVATE ACCESS SPECIFIER

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

PROTECTED ACCESS SPECIFIER

Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance.

INTERNAL ACCESS SPECIFIER

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

PROTECTED INTERNAL ACCESS SPECIFIER

The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

**** Furthermore, they are elaborated in 'Encapsulation' topic with examples.***

TOPIC NO. 7
ENCAPSULATION

What is Encapsulation?

Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'. Encapsulation, in object oriented programming methodology, prevents access to implementation details.

Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.

What is the need for Encapsulation?

The need of encapsulation is to protect or prevent the code (data) from accidental corruption due to the silly little errors that we are all prone to make. In Object oriented programming data is treated as a critical element in the program development and data is packed closely to the functions that operate on it and protects it from accidental modification from outside functions.

Encapsulation provides a way to protect data from accidental corruption. Rather than defining the data in the form of public, we can declare those fields as private.

Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifiers:

1. Public
2. Private
3. Protected
4. Internal
5. Protected internal

Public Access Modifier for Encapsulation:

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      public double length;
      public double width;

      public double GetArea()
      {
         return length * width;
      }
      public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}",
         GetArea());
      }
   }
```

```
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Rectangle rec = new Rectangle();
         rec.length = 4.5;
         rec.width = 3.5;
         rec.Display();
         Console.ReadLine();
      }
   }
}
```

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

Length: 4.5
Width: 3.5
Area: 15.75

In the preceding example, the member variables length and width are declared **public**, so they can be accessed from the function Main() using an instance of the Rectangle class, named **rec**.

The member function *Display()* and *GetArea()* can also access these variables directly without using any instance of the class.

The member functions *Display()* is also declared **public**, so it can also be accessed from *Main()* using an instance of the Rectangle class, named **rec**.

Private Access Modifier for Encapsulation:

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      private double length;
      private double width;

      public void Acceptdetails()
      {
         Console.Write("Enter Length: ");
         length = Convert.ToDouble(Console.ReadLine());
         Console.Write("Enter Width: ");
         width = Convert.ToDouble(Console.ReadLine());
      }
      private double GetArea()
      {
         return length * width;
      }
      public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}",
         GetArea());
      }
   }
```

```
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Rectangle rec = new Rectangle();
         rec.Acceptdetails();
         rec.Display();
         Console.ReadLine();
      }
   }
}
```

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

```
Enter Length: 4.4
Enter Width: 3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

In the preceding example, the member variables length and width and a method GetArea( ) are declared private, so they cannot be accessed from the function Main(). The member functions AcceptDetails( ) and Display() can access these variables. Since the member functions AcceptDetails( ) and Display() are declared public, they can be accessed from Main() using an instance of the Rectangle class, named rec.

TOPIC NO. 8
GETTER SETTERS

What are Getter and Setter Properties/Methods?

Getters and Setters are just means of helping encapsulation. When you make a class you have several class variables that perhaps you want to expose to other classes to allow them to get a glimpse of some of the data you store. While just making the variables public to begin with may seem like an acceptable alternative, in the long run you will regret letting other classes manipulate your class member variables directly. If you force them to do it through a setter, you can add logic to ensure no strange values ever occur, and you can always change that logic in the future without effecting things already manipulating this class. For such purpose Getter and Setter are used.

Overview of Properties

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In other languages, this is often accomplished by programs implementing specialized getter and setter methods. C# properties enable this type of protection while also letting you access the property just like it was a field.

Another benefit of properties over fields is that you can change their internal implementation over time. With a public field, the underlying data type must always be the same because calling code depends on the field being the same. However, with a property, you can change the implementation. For example, if a customer has an ID that is originally stored as an **int**, you might have a requirements change that made you perform a validation to ensure that calling code could never set the ID to a negative value. If it was a field, you would never be able to do this, but a property allows you to make such a change without breaking code. Now, let's see how to use properties.

Example-1:

```
using System;
public class Customer
{
    private int m_id = -1;

    public int GetID()
    {
        return m_id;
    }

    public void SetID(int id)
    {
        m_id = id;
    }

    private string m_name = string.Empty;

    public string GetName()
    {
        return m_name;
    }
```

```
    public void SetName(string name)
    {
        m_name = name;
    }
}

public class Program
{
    public static void Main()
    {
        Customer cust = new Customer();
        cust.SetID(6750);
        cust.SetName("");
        Console.WriteLine("ID:{0},Name:{1}",cust.GetID(), cust.GetName());
        Console.ReadKey();
    }
}
```

Example-2:

```
using System;
public class Student
{
    public int ID { get; set; }
    public string Name { get; set; }
}

public class Program
{
    static void Main()
    {
        Student std = new Student();
        std.ID = 6750;
        std.Name = "";
        Console.WriteLine("ID: {0}, Name: {1}", std.ID, std.Name);
        Console.ReadKey();
    }
}
```

TOPIC NO. 9
BOXING AND UNBOXING

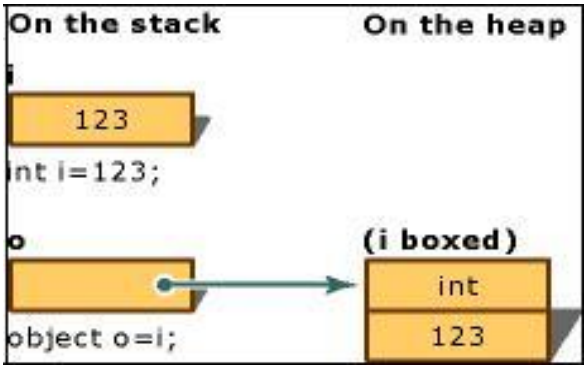
What is Boxing?

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a value type to the type object or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

Example:

```
int i = 123;
object o = num;
```

The result of this statement is creating an object reference 'o', on the stack, that references a value of the type **int**, on the heap. This value is a copy of the value-type value assigned to the variable 'i'. The difference between the two variables, 'i' and 'o', is illustrated in the following figure.



Boxing Conversion

What is Unboxing?

Unboxing is an explicit conversion from the type **object** to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

Checking the object instance to make sure that it is a boxed value of the given value type.

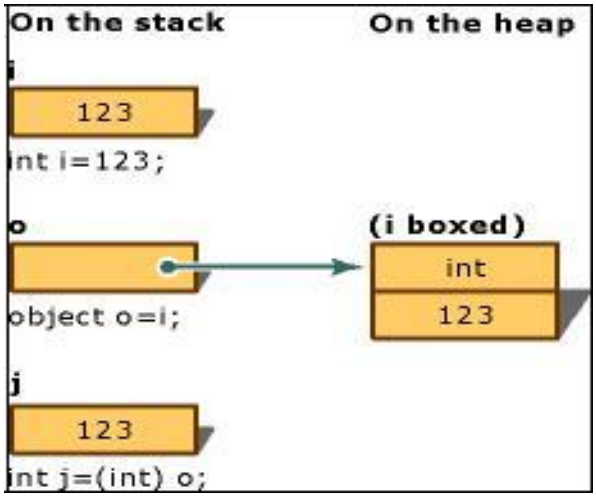
Copying the value from the instance into the value-type variable.

Example:

```
int i = 123;       // a value type
object o = i;      // boxing
int j = (int)o;    // unboxing
```

For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type. Attempting to unbox **null** causes

a NullReferenceException. Attempting to unbox a reference to an incompatible value type causes an InvalidCastException.



TOPIC NO. 10

STATIC VS. NON-STATIC

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, we cannot use the new keyword to create a variable of the class type. Because there is no instance variable, we access the members of a static class by using the class name itself.

C# fields must be declared inside a class. However, if we declare a method or a field as static, we can call the method or access the field using the name of the class. No instance is required. We can also use the static keyword when defining a field. With this feature, we can create a single field that is shared among all objects created from a single class. Non-static fields are local to each instance of an object.

When you define a static method or field, it does not have access to any instance fields defined for the class; it can use only fields that are marked as static. Furthermore, it can directly invoke only other methods in the class that are marked as static; non-static (instance) methods or fields first require creation of an object on which to call them.

Many locations exist in computer memory. A static thing stands alone. It occupies just one location. The static modifier is used on methods, classes, and variables/property. A keyword, **static** denotes things that are singular. They are part of no instance. Static often improves performance, but makes programs less flexible.

Static Methods Example:

```
using System; class
Program
{
    static void MethodA()
    {
     Console.WriteLine("Method-A is a Static method");
    }

    void MethodB()
    {
      Console.WriteLine("Method-B is an Instance method");
    }

    static string MethodC()
    {
      string C = “Method-C is a Static Method”;
      return 'C';
    }

    string MethodD()
    {
      string D = “Method-D is an Instance Method”;
      return 'D';
    }
```

```
    static void Main()
    {
      Program.MethodA();
      Console.WriteLine(Program.MethodC());
      Console.WriteLine();

      Program programInstance = new Program();

      programInstance.MethodB();
      Console.WriteLine(programInstance.MethodD());
      Console.WriteLine();
    }
}
```

Static Class Example:

```
using System;
static class Student
{
    public static int sid = 6750;

    public static string GetName()
    {
        string name = “”; return name;
    }
}

class Program
{
    static void Main()
    {
        Console.WriteLine(Student.sid);
        Console.WriteLine(Student.GetName());
    }
}
```

Note:

A static class must contain static properties and methods but a non-static class may or may not contain static methods.

TOPIC NO. 11
INHERITANCE

One of the most important concepts in object-oriented programming is inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS-A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.

Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>
{
   ...
}
class <derived_class> : <base_class>
{
   ...
}
```

Example:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      protected int width;
      protected int height;

      public void setWidth(int w)
      {
         width = w;
      }

      public void setHeight(int h)
      {
         height = h;
      }
   }
```

```
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         Console.WriteLine("Total area: {0}", Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

Multiple Inheritance in C#

C# does not support multiple inheritance. However, you can use interfaces to implement multiple inheritance. That inheritance is referred as Multi Level Inheritance. The following program demonstrates this:

```
using System;
namespace MultiLevelInheritance
{
    class Shape
    {
        protected double width;
        protected double height;

        public double setWidth(double _width)
        {
            width = _width;
            return width;
        }

        public double setHeight(double _height)
        {
            height = _height;
            return height;
        }
    }
}
```

```
using System;
namespace MultiLevelInheritance
{
    interface PaintCost
    {
        double getCost(double area, double rate);
    }
}
```

```
using System;
namespace MultiLevelInheritance
{
    class Rectangle : Shape, PaintCost
    {
        public double getArea()
        {
            double area = width * height;
            return area;
        }

        public double getCost(double area, double rate)
        {
            double amount = area * rate;
            return amount;
        }
    }
}
```

```
using System;
namespace MultiLevelInheritance
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter the Width of Rectangle: ");
            double width = double.Parse(Console.ReadLine());
            Console.Write("Enter the Height of Rectangle: ");
            double height = double.Parse(Console.ReadLine());
            Console.WriteLine("-----------------------------------------------------");

            Rectangle rec = new Rectangle();
            rec.setWidth(width);
            rec.setHeight(height);
            double area = rec.getArea();

            Console.WriteLine("Area of Rectangle is: " + area);
            Console.WriteLine();

            Console.Write("Enter Cost for Paint per Square Feet: " );
            double rate = double.Parse(Console.ReadLine());
            Console.WriteLine("-----------------------------------------------------");
            Console.WriteLine("Total Cost for the Painting in Rectangle is Rs. " +
rec.getCost(area, rate));
            Console.ReadLine();
        }
    }
}
```

TOPIC NO. 12
POLYMORPHISM

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'. Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. It includes Method/Function Overloading. In **dynamic polymorphism**, it is decided at run-time. It includes Method/Function Overriding.

METHOD / FUNCTION OVERLOADING (STATIC POLYMORPHISM)

In C# we are allowed to have functions with the same name, but having different data types parameters. The advantage is that we call the function by the same name as by passing different parameters, a different function gets called. This feature is called function overloading.

Example:

```
namespace MethodOverloading
{
    public class Test
    {
        public void Add(string a1, string a2)
        {
            Console.WriteLine("Adding Two String : " + a1 + a2);
        }

        public void Add(int a1, int a2)
        {
            Console.WriteLine("Adding Two Integer : " +  a1 + a2);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Test obj = new Test();
            obj.Add("Syed Umair " , "Ali");
            obj.Add(6750, 08);
            Console.ReadLine();
        }
    }
}
```

METHOD / FUNCTION OVERRIDING (DYNAMIC POLYMORPHISM)

The **override** modifier/keyword is required to extend or modify the abstract or **virtual** implementation of an inherited method, property, indexer, or event.

Method overriding refers to the change in implementation of a particular method/function that exist in the parent class. The method which is to be overridden is kept virtual using the 'virtual' keyword after the access modifier of a particular method and is overridden in the child class i.e. the implementation of the method is changed in the child class as according to the need.

Example:

```
namespace MethodOverriding
{
    class Program
    {
        public class Base
        {
            public virtual void Show()
            {
                Console.WriteLine("Show From Base Class.");
            }
        }

        public class Derived : Base
        {
            public override void Show()
            {
                Console.WriteLine("Show From Derived Class.");
            }
        }

        static void Main(string[] args)
        {
            Base objBase = new Base();
            objBase.Show();

            objBase = new Derived();
            objBase.Show();

            Console.ReadLine();
        }
    }
}
```

TOPIC NO. 13
ABSTRACTION

What is Abstraction?

The word abstract means a concept or an idea not associated with any specific instance. In programming we apply the same meaning of abstraction by making classes not associated with any specific instance.

The abstraction is done when we need to only inherit from a certain class, but not need to instantiate objects of that class. In such case the base class can be regarded as Incomplete". Such classes are known as an "Abstract Base Class".

Abstract Base Class:

There are some important points about Abstract Base Class:

1. An Abstract Base class cannot be instantiated; it means the object of that class cannot be created.
2. Class having abstract keyword and having, abstract keyword with some of its methods (not all) is known as an Abstract Base Class.
3. Class having Abstract keyword and having abstract keyword with all of its methods is known as pure Abstract Base Class.
4. The method of abstract class that has no implementation is known as "operation". It can be defined as abstract void method ();
5. An abstract class holds the methods but the actual implementation of those methods is made in derived class.

Example:

```
using System;
namespace Bank
{
    abstract class Account
    {
        protected double balance = 1000;

        public abstract string getDeposit(double deposit);

        public abstract string getWithdraw(double withdraw);

    }
}

using System;
namespace Bank
{
    class CurrentAccount : Account
    {
        public override string getDeposit(double deposit)
        {
            balance = balance + deposit;
            return "Your Total Balance is Rs. " + balance;
        }
```

```
        public override string getWithdraw(double withdraw)
        {
            if (balance > withdraw)
            {
                balance = balance - withdraw;
                return "Your Total Balance is Rs. " + balance;
            }
            else
            {
                return "Sorry, cannot withdraw requested amount. Insufficient Funds";
            }
        }
    }
}
```

```
using System;
namespace Bank
{
    class SavingAccount : Account
    {
        double interest = 100;

        public override string getDeposit(double deposit)
        {
            balance = balance + deposit;
            return "Your Total Balance is Rs. " + balance;
        }

        public override string getWithdraw(double withdraw)
        {
            if (balance > withdraw)
            {
                balance = balance - withdraw;
                return "Your Total Balance is Rs. " + (balance + interest);
            }
            else
            {
                return "Sorry, cannot withdraw requested amount. Insufficient Funds";
            }
        }
    }
}
```

```
using System;
namespace Bank
{
    class Program
    {
        static void Main(string[] args)
        {
            SavingAccount sa = new SavingAccount();
            Console.Write("Enter Amount to Withdraw: "); double
            amount = double.Parse(Console.ReadLine());
            Console.WriteLine(sa.getWithdraw(amount));
        }
    }
}
```

TOPIC NO. 14
INTERFACES

What are Interfaces?

An interface is defined as a syntactical contract that all the classes inheriting the interface should follow. The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.

Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

Properties of Interfaces:

An interface has the following properties:

An interface is like an abstract base class. Any class that implements the interface must implement all its members.

An interface can't be instantiated directly. Its members are implemented by any class that implements the interface.

Interfaces can contain events, indexers, methods, and properties.

Interfaces contain no implementation of methods.

A class can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceApplication
{
    interface ITransaction
    {
        void showTransaction();
        double getAmount();
    }

    class Transaction : ITransaction
    {
        string TransactionCode;
        string Date;
        double Amount;

        public Transaction()
        {
        }
```

```
        public Transaction(string _TransactionCode, string _Date, double _Amount)
        {
            TransactionCode =
            _TransactionCode; Date = _Date;
            Amount = _Amount;
        }

        public void showTransaction()
        {
            Console.WriteLine("Transaction Code: {0}", TransactionCode);
            Console.WriteLine("Transaction Date: {0}", Date);
            Console.WriteLine("Transaction Amount: {0}", getAmount());
        }

        public double getAmount()
        {
            return Amount;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Transaction tran = new Transaction("BS-67795301", "05-Mar-2016", 5055.25);
            tran.showTransaction();
        }
    }
}
```

TOPIC NO. 15
RELATIONSHIPS

What are Relationships in Object Oriented Programming?

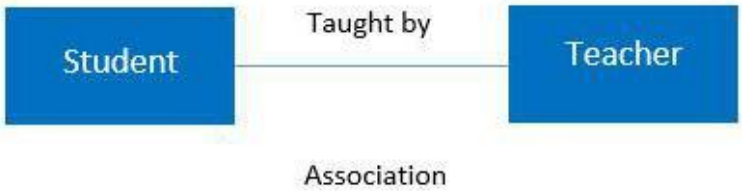
A relationship defines the connection between objects. This explains how objects are connected to each other's and how they will behave. Basically there are three (3) types of relationship exist in Object Oriented Programming.

1. Association
2. Aggregation
3. Composition

ASSOCIATION

It represents a relationship between two or more objects where all objects have their own lifecycle and there is no owner. The name of an association specifies the nature of relationship between objects. This is represented by a solid line.

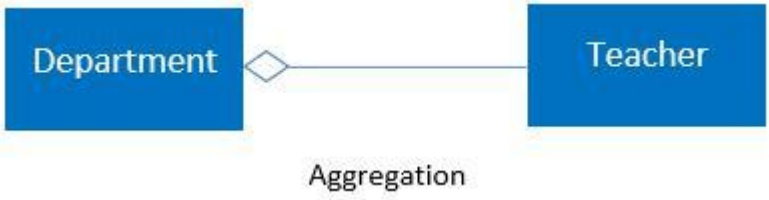
Let's take an example of relationship between Teacher and Student. Multiple students can associate with a single teacher and a single student can associate with multiple teachers. But there is no ownership between the objects and both have their own lifecycle. Both can be created and deleted independently.



AGGREGATION

It is a specialized form of Association where all object have their own lifecycle but there is ownership. This represents "whole-part or a-part-of" relationship. This is represented by a hollow diamond followed by a line.

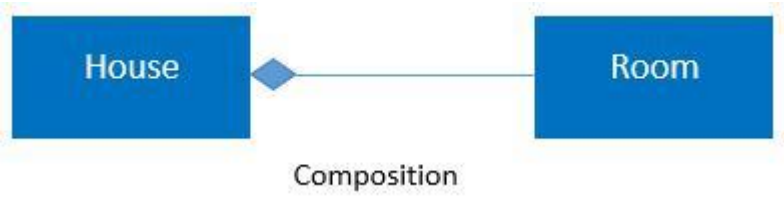
Let's take an example of relationship between Department and Teacher. A Teacher may belongs to multiple departments. Hence Teacher is a part of multiple departments. But if we delete a Department, Teacher Object will not destroy.



COMPOSITION

It is a specialized form of Aggregation. It is a strong type of Aggregation. In this relationship child objects does not have their lifecycle without Parent object. If a parent object is deleted, all its child objects will also be deleted. This represents "death" relationship. This is represented by a solid diamond followed by a line.

Let's take an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room cannot belongs to two different house if we delete the house room will automatically delete.



Association Code Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Association
{
    class Account
    {
        public string AccountNo { set; get; }
        public string OpeningDate { set; get;
        } private double Balance { set; get; }

        public Account()
        {
            Balance = 1000.00;
        }

        public Account(string _AccountNo, string _OpeningDate)
        {
            AccountNo = _AccountNo;
            OpeningDate = _OpeningDate;
        }

        public void Deposit(double _amount)
        {
            Balance = Balance + _amount;
            Console.WriteLine("Total Amount Deposited in the Account is Rs. " +
_amount);
            Console.WriteLine("Total Amount of Balance in the Acount is Rs. " +
Balance);
        }

        public void Withdraw(double _amount)
        {
            if (Balance > _amount)
            {
```

```
                Balance = Balance - _amount;
                Console.WriteLine("Total Amount Withdrawn from the Account is Rs. " +
_amount);
                Console.WriteLine("Total Amount of Balance in the Acount is Rs. " +
Balance);
            }
            else
            {
                Console.WriteLine("Insufficient Funds in the Account");
            }
        }
    }

    class Customer
    {
        public string CustomerName { set; get; }
        public string CustomerEmail { set; get; }
        public Account BankAccount { set; get; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Account account = new Account();
            account.AccountNo = "BI-67506739-B";
            account.OpeningDate = "05-Mar-2016";

            Customer customer = new Customer();
            customer.CustomerName = ""; customer.CustomerEmail
            = "syedumoali@gmail.com"; customer.BankAccount =
            account;

            customer.BankAccount.Deposit(5005.25);
            customer.BankAccount.Withdraw(1050.75);
        }
    }
}
```

Aggregation Code Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Aggregation
{
    class Employee
    {
        string Name;
        int Age;
        string Designation;

        public Employee()
        {
        }

        public Employee(string _name, int _age, string _designation)
        {
            Name = _name;
            Age = _age;
            Designation = _designation;
```

```
        }

        public string PrintInfo()
        {
            string output = "Name: " + Name + "\n" + "Age: " + Age + "\n" +
"Designation: " + Designation;
            return output;
        }
    }

    class Department
    {
        string DepartmentName;
        string InstituteName;
        Employee emp;

        public Department()
        {
        }

        public Department(string _departmentName, string _instituteName)
        {
            DepartmentName = _departmentName;
            InstituteName = _instituteName;
        }

        public string GetDepartmentInfo()
        {
            emp = new Employee("Arsalan Majeed", 22, "Program Manager");
            string output = "Department Name: " + DepartmentName + "\n" + "Institute
Name: " + InstituteName + "\n" + emp.PrintInfo();
            return output;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Department dep = new Department("Human Resource",
            "PAF-KIET"); Console.WriteLine(dep.GetDepartmentInfo());
        }
    }
}
```

Composition Code Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Composition
{
    class Engine
    {
        string   EngineName;
        string   EngineType;
        string EnginePower;

        public void SetEngineName(string _engineName)
        {
            EngineName = _engineName;
```

```
        }
        public string GetEngineName()
        {
            return EngineName;
        }

        public void SetEngineType(string _engineType)
        {
            EngineType = _engineType;
        }
        public string GetEngineType()
        {
            return EngineType;
        }

        public void SetEnginePower(string _enginePower)
        {
            EnginePower = _enginePower;
        }
        public string GetEnginePower()
        {
            return EnginePower;
        }
    }

    class Car
    {
        string color;
        string company;
        int model;
        Engine eng;

        public Car()
        {
        }

        public Car(string _color, string _company, int _model, string _engineName,
string _engineType, string _enginePower)
        {
            eng = new Engine();
            color = _color;
            company = _company;
            model = _model;
            eng.SetEngineName(_engineName);
            eng.SetEngineType(_engineType);
            eng.SetEnginePower(_enginePower);
        }

        public string GetEngineName()
        {
            return eng.GetEngineName();
        }

        public string GetEngineType()
        {
            return eng.GetEngineType();
        }

        public string GetEnginePower()
        {
            return eng.GetEnginePower();
        }
```

```
        public string GetCarInfo()
        {
            string output = "Car Colour: " + color + "\n" + "Car Company: " + company
+ "\n" + "Car Model: " + model;
            return output;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Car car = new Car("Blue", "Honda", 2012, "Honda", "V8-Classic", "320-
BHP");
            Console.WriteLine(car.GetCarInfo());
            Console.WriteLine("Car Engine Name: " + car.GetEngineName());
            Console.WriteLine("Car Engine Type: " + car.GetEngineType());
            Console.WriteLine("Car Engine Power: " + car.GetEnginePower());
        }
    }
}
```