**Dot Net Training 7th Feb 2022**

**Link:**[*https://catalys.portal.nttdataservices.com/course/view.php?id=11544*](https://catalys.portal.nttdataservices.com/course/view.php?id=11544)*)*

1)Basic Concepts of C#

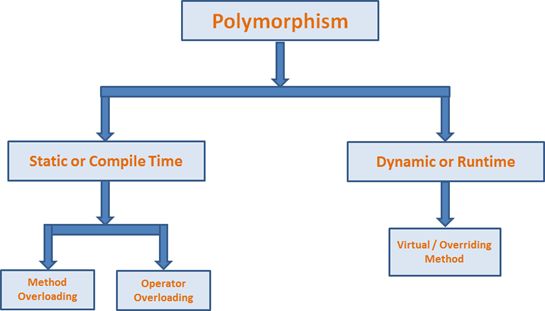
using System;  
namespace sai.CS  
{  
    class FirstCSProgram  
    {  
        static void Main()  
        {  
            Console.WriteLine("This Is Our First CSharp Program.");  
             Console.ReadLine();  
             return;  
        }  
    }  
}

Like the C++ language, C# is an Object Oriented programming language. Generally many people spell C# as C#.Net (C Sharp dot Net), but here Microsoft developed the .Net environment mainly for distributed applications (the sharing of processing between client and server) and in C#.Net "Net" indicates that C# is used to develop only Distributed Applications but using C# we can develop any kind of software applications including Windows applications.

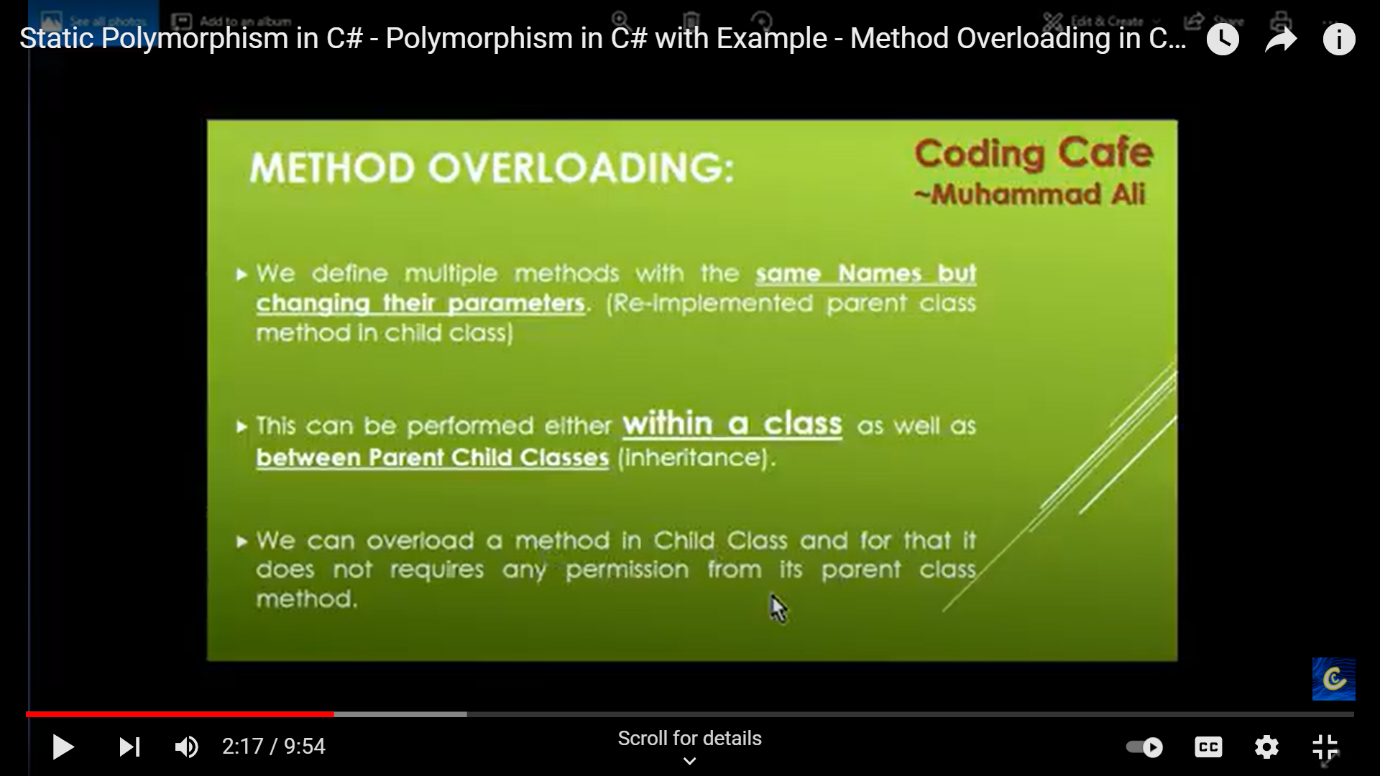
# 2)C# - Polymorphism

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. In **dynamic polymorphism**, it is decided at run-time.



## **static polymorphism(method overloading)/compile time**



The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

* Function overloading
* Operator overloading

## **Function Overloading**

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types

using System;

namespace PolymorphismApplication {

class Printdata {

void print(int i) {

Console.WriteLine("Printing int: {0}", i );

}

void print(double f) {

Console.WriteLine("Printing float: {0}" , f);

}

void print(string s) {

Console.WriteLine("Printing string: {0}", s);

}

static void Main(string[] args) {

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

p.print("Hello C++");

Console.ReadKey();

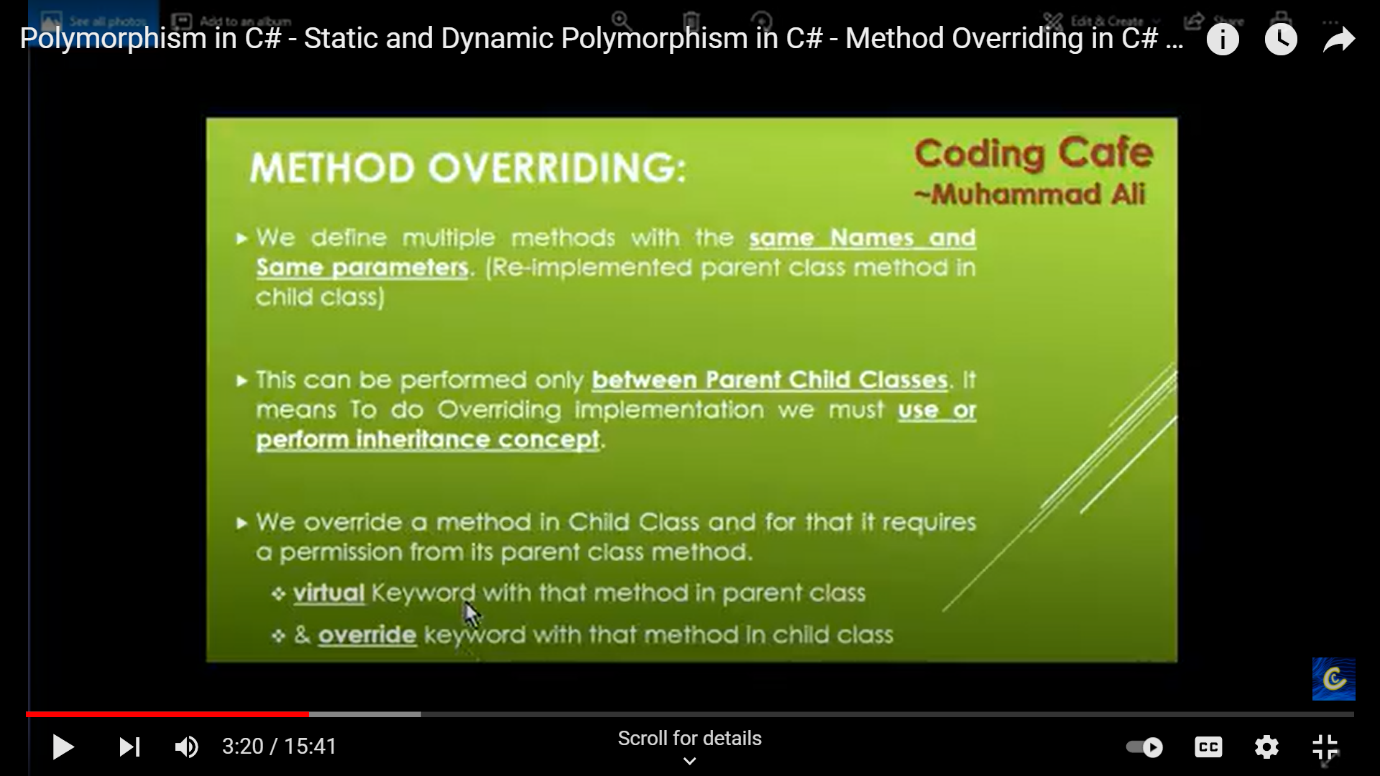
}

}

}

## **Dynamic Polymorphism(Method Overriding)/Run Time**

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.



Here are the rules about abstract classes −

* You cannot create an instance of an abstract class
* You cannot declare an abstract method outside an abstract class
* When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed

using System;

namespace PolymorphismApplication {

abstract class Shape {

public abstract int area();

}

class Rectangle: Shape {

private int length;

private int width;

public Rectangle( int a = 0, int b = 0) {

length = a;

width = b;

}

public override int area () {

Console.WriteLine("Rectangle class area :");

return (width \* length);

}

}

class RectangleTester {

static void Main(string[] args) {

Rectangle r = new Rectangle(10, 7);

double a = r.area();

Console.WriteLine("Area: {0}",a);

Console.ReadKey();

}

}

}

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

using System;

namespace PolymorphismApplication {

class Shape {

protected int width, height;

public Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

public virtual int area() {

Console.WriteLine("Parent class area :");

return 0;

}

}

class Rectangle: Shape {

public Rectangle( int a = 0, int b = 0): base(a, b) {

}

public override int area () {

Console.WriteLine("Rectangle class area :");

return (width \* height);

}

}

class Triangle: Shape {

public Triangle(int a = 0, int b = 0): base(a, b) {

}

public override int area() {

Console.WriteLine("Triangle class area :");

return (width \* height / 2);

}

}

class Caller {

public void CallArea(Shape sh) {

int a;

a = sh.area();

Console.WriteLine("Area: {0}", a);

}

}

class Tester {

static void Main(string[] args) {

Caller c = new Caller();

Rectangle r = new Rectangle(10, 7);

Triangle t = new Triangle(10, 5);

c.CallArea(r);

c.CallArea(t);

Console.ReadKey();

}

}

}

# 3)C# | Method Overloading

***Method Overloading*** is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. C# can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.

* Overloaded methods are differentiated based on the number and type of the parameters passed as arguments to the methods.
* You can not define more than one method with the same name, Order and the type of the arguments. It would be compiler error.
* The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error. If both methods have the same parameter types, but different return type, then it is not possible.

**Different ways of doing overloading methods-**  
Method overloading can be done by changing:

1. The number of parameters in two methods.
2. The data types of the parameters of methods.
3. The Order of the parameters of methods

**1)By changing the Number of Parameters**

// C# program to demonstrate the function

// overloading by changing the Number

// of parameters

using System;

class GFG {

    // adding two integer values.

    public int Add(int a, int b)

    {

        int sum = a + b;

        return sum;

    }

    // adding three integer values.

    public int Add(int a, int b, int c)

    {

        int sum = a + b + c;

        return sum;

    }

    // Main Method

    public static void Main(String[] args)

    {

        // Creating Object

        GFG ob = new GFG();

        int sum1 = ob.Add(1, 2);

        Console.WriteLine("sum of the two "

                          + "integer value : " + sum1);

        int sum2 = ob.Add(1, 2, 3);

        Console.WriteLine("sum of the three "

                          + "integer value : " + sum2);

    }

}

**Output:**

sum of the two integer value : 3

sum of the three integer value : 6

Name2 : Abby, Id2 : 2

# 4) C# | Constructor Overloading

It is quite similar to the [Method Overloading](https://www.geeksforgeeks.org/c-method-overloading/). It is the ability to redefine a Constructor in more than one form. A user can implement constructor overloading by defining two or more constructors in a class sharing the same name. C# can distinguish the constructors with different signatures. i.e. the constructor must have the same name but with different parameters list.

We can overload constructors in different ways as follows:

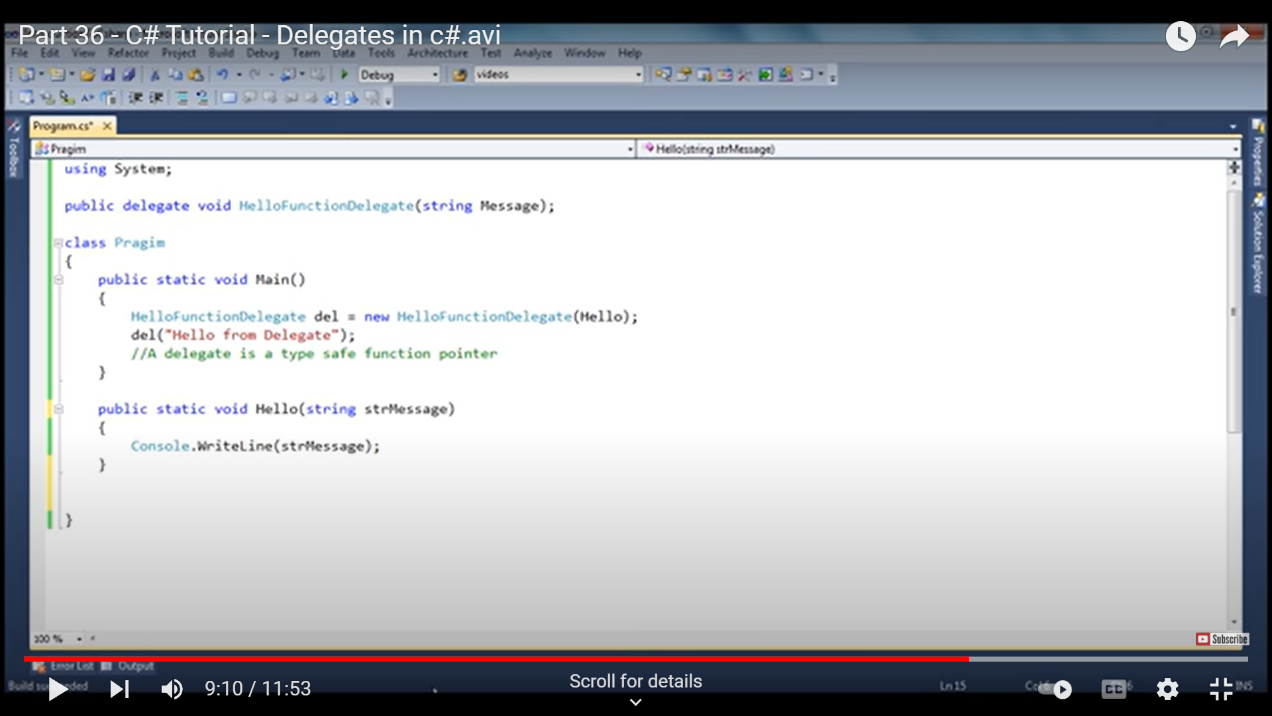
* By using different **type of arguments**
* By using different **number of arguments**
* By using different **order of arguments**

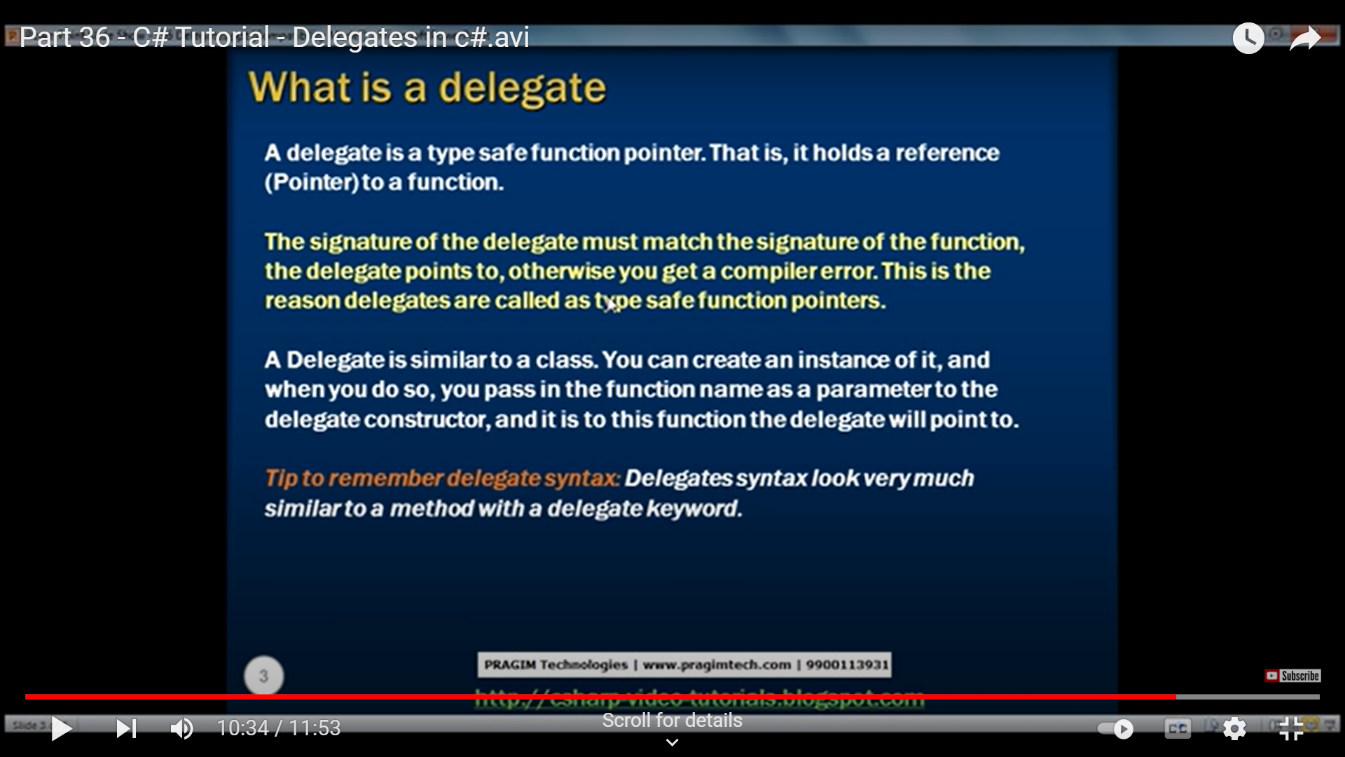
# 5)Delegates

Def: A delegate is a type safe function pointer.

>signature of the delegates must match with the signature of functions, the delegate points to.

>Delegates syntax look similar to a method with a delegate keyword.





# 6) Implicit and Explicit Interface Implementation in C#

**Implicit interface implementation**

This is the most regular or obvious way  to implement members of an interface. Here we don't specify the interface name of the members and implement implicitly. The method can be declared at any interface (s) the class implements.

For example:

1. **interface** ITest
2. {
3. **void** TestMethod();
4. }
5. **class** TestClass: ITest
6. {
7. **public** **void** TestMethod()
8. {
9. Console.WriteLine("Implicit Interface Implementation");
10. }
11. }

The call of the method is also not different. Just create an object of the class and invoke it.

1. **class** Program
2. {
3. **static** **void** Main(**string**[] args)
4. {
5. TestClass obj = **new** TestClass();
6. obj.TestMethod(); //Way to call implicitely implemented method
7. }
9. }
10. **Output**
11. 

**Explicit interface implementation**

This is another way to implement members of an interface. Here we need to specify the interface name of the members. The following example explains that.

1. **class** TestClass: ITest
2. {
3. **void** ITest.TestMethod()
4. {
5. Console.WriteLine("Explicit Interface Implementation");
6. }
7. }

The constraint with explicit implementation is that an explicitly implemented member cannot be accessed using a class instance, but only through an instance of the interface. Please have a look at the example below.

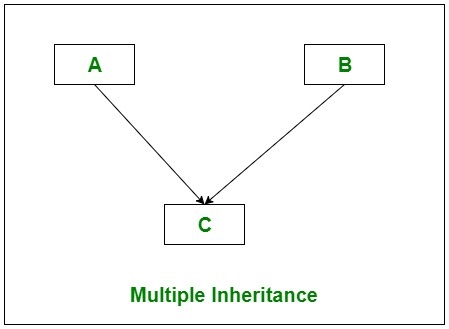
1. **class** Program
2. {
3. **static** **void** Main(**string**[] args)
4. {
5. ITest obj2 = **new** TestClass();
6. obj2.TestMethod();
7. }
8. }

**Output**

****

# 7) Multiple inheritance using c#

In Multiple inheritance, one class can have more than one superclass and inherit features from all its parent classes. As shown in the below diagram, class C inherits the features of class A and B.



But C# does not support multiple class inheritance. To overcome this problem we use interfaces to achieve multiple class inheritance. With the help of the [interface](https://www.geeksforgeeks.org/c-sharp-interface/), class C( as shown in the above diagram) can get the features of class A and B.

But we can indirectly inherit the features of Geeks1 and Geek2 class into GeeksforGeeks class using interfaces. As shown in the below diagram.

**Example 2:** Both *GFG1*and *GFG2*interfaces are implemented by Geeks1 and Geeks2 class. Now Geeks1 and Geeks2 class define languages() and courses() method. When a GeeksforGeeks class inherits GFG1 and GFG2 interfaces you need not to redefine *languages()* and *courses()* method just simply create the objects of *Geeks1*and *Geeks2*class and access the *languages()* and *courses()* method using these objects in GeeksforGeeks class.

|  |
| --- |
| // C# program to illustrate how to  // implement multiple class inheritance  // using interfaces  using System;  using System.Collections;    // Interface 1  interface GFG1 {      void languages();  }    // Parent class 1  class Geeks1 : GFG1 {        // Providing the implementation      // of languages() method      public void languages()      {            // Creating ArrayList          ArrayList My\_list = new ArrayList();            // Adding elements in the          // My\_list ArrayList          My\_list.Add("C");          My\_list.Add("C++");          My\_list.Add("C#");          My\_list.Add("Java");            Console.WriteLine("Languages provided by GeeksforGeeks:");          foreach(var elements in My\_list)          {              Console.WriteLine(elements);          }      }  }    // Interface 2  interface GFG2 {      void courses();  }    // Parent class 2  class Geeks2 : GFG2 {        // Providing the implementation      // of courses() method      public void courses()      {            // Creating ArrayList          ArrayList My\_list = new ArrayList();            // Adding elements in the          // My\_list ArrayList          My\_list.Add("System Design");          My\_list.Add("Fork Python");          My\_list.Add("Geeks Classes DSA");          My\_list.Add("Fork Java");            Console.WriteLine("\nCourses provided by GeeksforGeeks:");          foreach(var elements in My\_list)          {              Console.WriteLine(elements);          }      }  }  // Child class  class GeeksforGeeks : GFG1, GFG2 {        // Creating objects of Geeks1 and Geeks2 class      Geeks1 obj1 = new Geeks1();      Geeks2 obj2 = new Geeks2();        public void languages()      {          obj1.languages();      }        public void courses()      {          obj2.courses();      }  }    // Driver Class  public class GFG {        // Main method      static public void Main()      {            // Creating object of GeeksforGeeks class          GeeksforGeeks obj = new GeeksforGeeks();          obj.languages();          obj.courses();      }  } |

**Output:**

Languages provided by GeeksforGeeks:

C

C++

C#

Java

Courses provided by GeeksforGeeks:

System Design

Fork Python

Geeks Classes DSA

Fork Java

# 7)File Handling IO in c#

# 

**7)Exception Handling in c#**

What is an exception?

Errors occurs during the execution of the program.

Two types of errors are

* 1. Compile time error (Errors occurs due to syntax form)
  2. Run Time error (Errors occurs during the Execution of the program)
     + Run time errors are due to wrong implementation of logic
     + Wrong input supplied
     + Missing required resources

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

* **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
* **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
* **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

**7)File Handling in c#**

A file is a collection of data stored on a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a stream.

In C#, I/O classes are defined in the System.IO namespace. The basic file I/O class is FileStream, File I/O in C# is simpler as compared to other programming languages like C++.