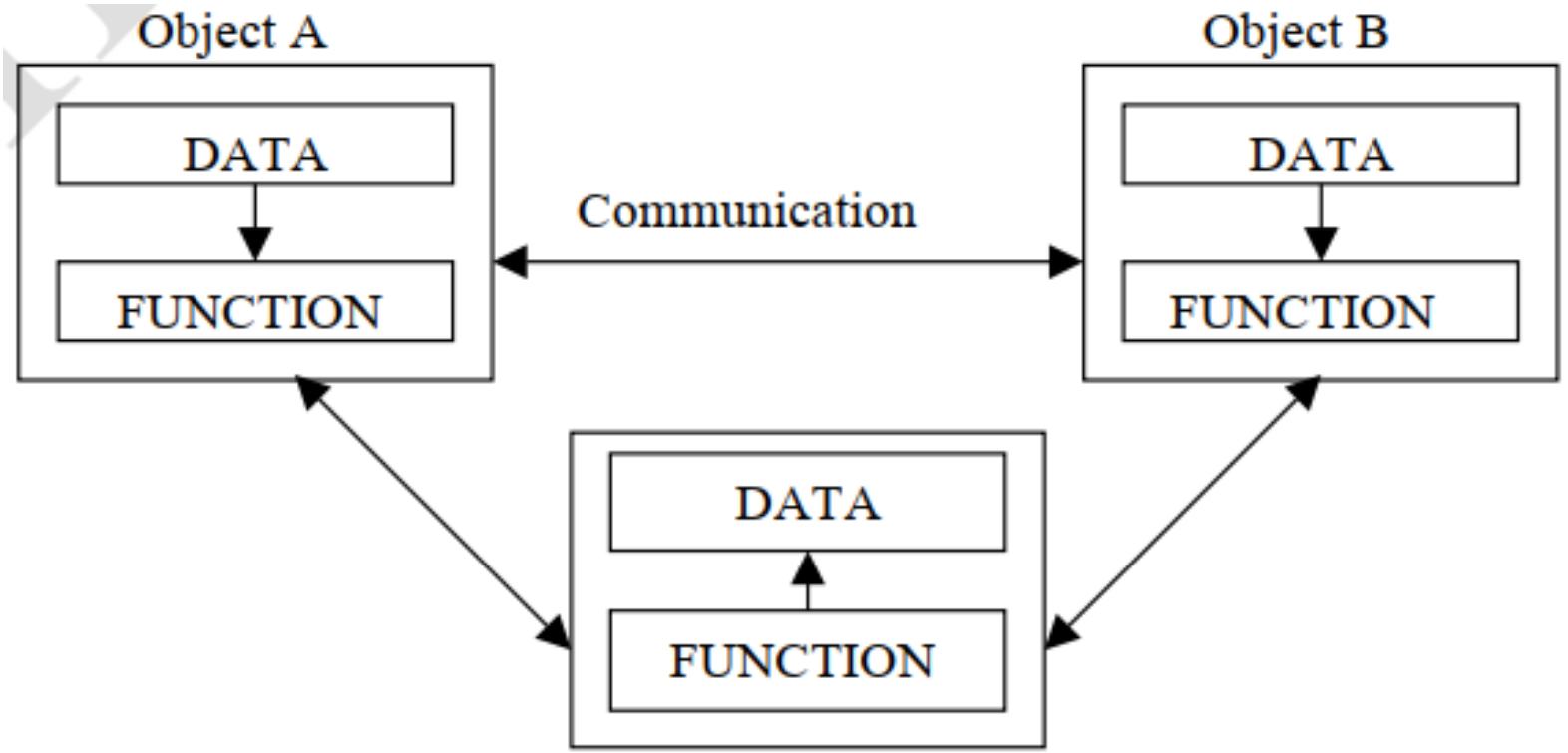
Object Oriented Programming

Abstraction
Abstract Classes and Methods



Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces**

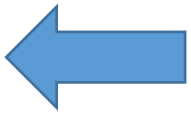
The abstract keyword is used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class student
{
    public abstract void studentinfo();
    public void studentavarege()
    {
        av = (d1 + d2 + d3) / 3;
    }
}
```



```
student myObj = new student();
```

it is not possible to create an object of the **student** class:

```
Will generate an error (Cannot create an instance of the abstract class or interface 'Student')
```

To access the abstract class, it must be inherited from another class.

we use the `override` keyword to override the base class method.

```
using System;
class Program9
{
    abstract class student
    {
        private string name;
        private int d1, d2, d3;
        private double av;

        public abstract void studentinfo();  // Abstract method (does not have a body)

        public double studentavarege()         // Regular method
        {
            d1 = 100; d2 = 80; d3 = 90;
            av = (d1 + d2 + d3) / 3;
            return av;
        }
    }
    class stu : student
    {
        public override void studentinfo()
        {
            // The body of studentinfo is provided here
            Console.WriteLine("NAME = AHMAD");
        }
    }
    static void Main(string[] args)
    {
        stu myobject = new stu();           // Create a myobject object
        double a;
        myobject.studentinfo();             // Call the abstract method
        a=myobject.studentavarege();        // Call the regular method
        Console.WriteLine("AVAREGE =" + a);

        Console.ReadKey();
    }
}
```

Why To Use Abstract Classes and Methods?

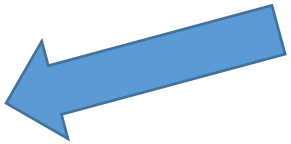
- To achieve security
- hide certain details
- show the important details of an object.

Interface

Another way to achieve abstraction

An `interface` is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies): not fields(attributes).

Example



```
// interface
interface Istudent
{
  void studentinfo();     // interface method (does not have a body)
  void studentavarege(); // interface method (does not have a body)
}
```

It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.
By default, members of an interface are abstract and public.
Note: Interfaces can contain properties and methods, but not fields(attributes).

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class.

To implement an interface, use the `:` symbol (just like with inheritance).

Note that you do not have to use the `override` keyword when implementing an interface:

```
Using system;

 class Program

{

interface Istudent

{

 void studentinfo(); // interface (does not have a body)

}

class Stu : Istudent  // Stu "implements" the Istudent interface

{

   public void studentinfo()

  {

    // The body of studentinfo() is provided here

    Console.WriteLine("NAME= AHMAD");

  } }

  static void Main(string[] args)

    {

      // Create a Stu object

      Stu myobject = new Stu();

      myobject.studentinfo();

    }

  }
```

Notes on Interfaces:

- Like **abstract classes**, interfaces cannot be used to create objects
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface members are by default abstract and public
- An interface cannot contain a constructor (as it cannot be used to create objects)

Note: To implement multiple interfaces, separate them with a comma

You cannot apply access modifiers to interface members.
All the members are public by default.
If you use an access modifier in an interface, then the C# compiler will give a compile-time error "The modifier 'public/private/protected' is not valid for this item"

```
interface Ifile
 {
   protected void  ReadFile();                 //compile-time error
   private    void  WriteFile(string text);    //compile-time error
 }
```

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma.

Multiple Interfaces

Using system;

program programinterfacemulti;

{

```
interface IFirstInterface
{
  void myMethod(); // interface method
}
 interface ISecondInterface
{
  void myOtherMethod(); // interface
method
}
```

```
// Implement multiple interfaces
class DemoClass : IFirstInterface, ISecondInterface
{   public void myMethod()
    { Console.WriteLine("METHOD 1..."); }
    public void myOtherMethod()
    { Console.WriteLine("METHOD 2..."); }
}
static void Main(string[] args)
  {  DemoClass myObj = new DemoClass();
     myObj.myMethod();
     myObj.myOtherMethod();
  }
}
}
```

Example: Explicit Implementation

```
interface Ifile
 {
     void ReadFile();
     void WriteFile(string text);
 }
```

```
class FileInfo : Ifile
 {
   void IFile.ReadFile()
   {
     Console.WriteLine("Reading File");
   }
   void IFile.WriteFile(string text)
   {
     Console.WriteLine("Writing to file");
   }
}
```

When you implement an interface explicitly, you can access interface members only through the instance of an interface type.

```
Using system;
public class Program
{
nterface Ifile
 {
  void ReadFile();
  void WriteFile(string text);
 }
class FileInfo : Ifile
 {
   void IFile.ReadFile()
 {
   Console.WriteLine("Reading File");
 }
void IFile.WriteFile(string text)
 {
 Console.WriteLine("Writing to file");
 }
public void Search(string text)
 {
  Console.WriteLine("Searching in file");
 }
}
```

```
public static void Main()
 {
  IFile file1 = new FileInfo();
  FileInfo file2 = new FileInfo();

  file1.ReadFile();
  file1.WriteFile("content");

  //file1.Search("text  to  be  searched")//compile-
time error

  file2.Search("text to be searched");
  //file2.ReadFile(); //compile-time error

//file2.WriteFile("content"); //compile-time error
}
}
```

In the above example, file1 object can only access members of IFile, and file2 can only access members of FileInfo class. This is the limitation of explicit implementation.

Implementing Multiple Interfaces

A class or struct can implement multiple interfaces. It must provide the implementation of all the members of all interfaces. Example: Implement Multiple Interfaces

```
interface Ifile
{
void ReadFile();
}
interface IBinaryFile
{
void OpenBinaryFile();
void ReadFile();
}
class FileInfo : IFile, IBinaryFile
{
void IFile.ReadFile()
{
Console.WriteLine("Reading Text File");
}
void IBinaryFile.OpenBinaryFile()
{
Console.WriteLine("Opening Binary File");
 }
void IBinaryFile.ReadFile()
{
Console.WriteLine("Reading Binary File");
}
public void Search(string text)
{
Console.WriteLine("Searching in File");
}
}
```

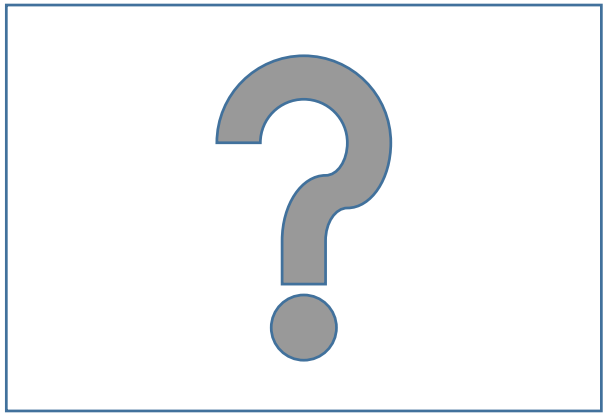
```
public class Program
{
public static void Main()
{
IFile file1 = new FileInfo();
IBinaryFile file2 = new FileInfo();
FileInfo file3 = new FileInfo();
file1.ReadFile();        //file1.OpenBinaryFile(); //compile-time error        //file1.SearchFile("text to be searched"); //compile-time error
file2.OpenBinaryFile();
file2.ReadFile();              //file2.SearchFile("text to be searched"); //compile-time error
file3.Search("text to be searched");
//file3.ReadFile();              //compile-time error
//file3.OpenBinaryFile(); //compile-time error
}
}
```

the FileInfo implements two interfaces IFile and IBinaryFile explicitly. It is recommended to implement interfaces explicitly when implementing multiple interfaces to avoid confusion and more readability.

Points to Remember :

•Interface can contain declarations of method, properties, indexers, and events.
•Interface cannot include private, protected, or internal members. All the members are public by default.
•Interface cannot contain fields, and auto-implemented properties.
•A class or a struct can implement one or more interfaces implicitly or explicitly. Use public modifier when implementing interface implicitly, whereas don't use it in case of explicit implementation.
•Implement interface explicitly using InterfaceName.MemberName.
•An interface can inherit one or more interfaces.

QUESTION



Google Classroom :