

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Internal Assessment Test 2 – June 2022

Sub:	Programming Using C#							Sub Code:	20MCA42
Date:	4/06/2022	Duration:	90 min's	Max Marks:	50	Sem:	IV	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

PART I		MARKS	OBE	
			CO	RBT
1	Make a short note on Inheritance giving example. OR	[10]	CO1	L1
2	Explain how to defining partial class and partial method with an example.	[10]	CO1	L2
PART II				
3	What are the three pillars of object oriented programming in C#? OR	[10]	CO1	L2
4	What are the two ways of enforcing encapsulation? Give examples for both the methods	[10]	CO1	L2

PART III				
5	Write a short note on: i. Static data ii. Virtual method iii. base keyword OR	[10]	CO1	L2
6	Explain : a) Classical Inheritance b) Containment / Delegation model	[10]	CO1	L2
PART IV				
7	Make a short note on abstraction. OR	[10]	CO1	L2
8	Explain : i. Method Overloading ii. Operator Overloading	[10]	CO1	L2
PART V				
9	Write a program to overload + and – operators for adding and subtracting two square matrices. OR	[10]	CO1	L3
10	What is Interface? Explain how to create interfaces and implemented with an example.	[10]	CO1	L3

--	--	--	--	--	--	--	--	--	--

Internal Assessment Test 2 – June 2022

Sub:	Programming Using C#						
Date:	4/06/2022	Duration:	90 min's	Max Marks:	50	Sem:	IV

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

PART I

1 Make a short note on Inheritance giving example.

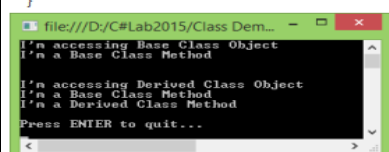
Inheritance provides you to reuse existing code and fast implementation time. The relationship between two or more classes is termed as Inheritance. In essence, inheritance allows to extend the behavior of a base (or parent/super) class by enabling a subclass to inherit core functionality (also called a derived class/child class). All public or protected variables and methods in the base class can be called in the derived classes.

Inheritance and Constructors: As you know, a constructor is a special method of a class, which is used to initialize the members of the same class. A constructor is called by default whenever an object of a class is created. It is important to note that a Base class constructors are not inherited by derived classes. Thus cannot instantiate a base constructor using derived class object, so we need the “base” keyword to access constructor of the base class from within a derived class.

Inheritance is of four types, which are as follows: i. Single Inheritance: Refers to inheritance in which there is only one base class and one derived class. This means that a derived class inherits properties from single base class. ii. Hierarchical Inheritance: Refers to inheritance in which multiple derived classes are inherited from the same base class. iii. Multilevel Inheritance: Refers to inheritance in which a child class is derived from a class, which in turn is derived from another class. iv. Multiple Inheritance: Refers to inheritance in which a child class is derived from multiple base class. C# supports single, hierarchical, and multilevel inheritance because there is only a single base class. It does not support multiple inheritance directly.

InherDemo.cs

```
using System;
namespace Class_Demos{
    class BaseClass{
        public int dm;
        public void BCMethod(){
            Console.WriteLine("I'm a Base Class Method");
        }
    }
    class DerivedClass:BaseClass{
        public void DCMethod(){
            Console.WriteLine("I'm a Derived Class Method");
        }
    }
}
```



```
class InherDemo{
    static void Main(){
        //Create a Base Class Object
        Console.WriteLine("I'm accessing Base Class Object");
        BaseClass bc = new BaseClass();
        bc.dm = 10;
        bc.BCMethod();
        //Create a Derived Class Object
        Console.WriteLine("I'm accessing Derived Class Object");
        DerivedClass dc=new DerivedClass();
        dc.dm = 20;
        dc.BCMethod();
        dc.DCMethod();
        Console.WriteLine("\nPress ENTER to quit...");
        Console.Read();
    }
}
```

2 Explain how to defining partial class and partial method with an example.

The partial class is a class that enables you to specify the definition of a class, structure, or interface in two or more source files. All the source files, each containing a section of class definition, combine when the application is complete. You may need a partial class when developers are working on large projects. A partial class distributes a class over multiple separate files; allowing developers to work on the class simultaneously. You can declare a class as partial by using the “partial” keyword. All the divided sections of the partial class must be available to form the final class when you compile the program. Let’s see that in above figure. All the section must have the same accessibility modifiers, such as public or private.

Partial method are only allowed in partial types, such as classes and structs. A partial method consists of 2 parts that listed below: Deals with defining the partial method\ Deals with implementing the partial method\ declaration Rules: Must have a void return type No access modifier are allowed for declaring a partialmethod except for static Partial methods are private by default

```
using System;
namespace Class_Demos{
    partial class MyTest {
        private int a;
        private int b;
        public void getAnswer(int a1, int b1){
            a = a1;
            b = b1;
        }
        static partial void Message();
    }
    partial class MyTest{
        partial void Message() {
            Console.WriteLine("Successfully accessed. . . . . ");
        }
        public void DisplayAns() {
            Console.WriteLine("Integer values: {0}, {1}", a, b);
            Console.WriteLine("Addition:{0}", a + b);
            Console.WriteLine("Multiply:{0}", a * b);
            Message();
        }
    }
    class PartialEx{
        public static void Main(){
            MyTest ts = new MyTest();
            ts.getAnswer(2, 3);
            ts.DisplayAns();
            Console.Read();
        }
    }
}}
```

PART II

3 What are the three pillars of object oriented programming in C#?

- i. Encapsulation : How does this language hide an object’s internal implementation?
- ii. Inheritance : How does this language promote code reuse?
- iii. Polymorphism : How does this language let you treat related objects in a similar way?

4 What are the two ways of enforcing encapsulation? Give examples for both the methods

Encapsulation using accessors and mutators: Rather than defining the data in the form of public, we can

declare those fields as private so that we achieved encapsulation. The Private data are manipulated using accessor (get: store the data to private members) and mutator (set: to interact with the variable) methods. Syntax:

```
set { }  
get { }
```

. A property defined with both a getter and a setter is called a read-write property. A property defined with only a getter is called a read-only property. A property defined with only a setter is called a write-only property.

Encapsulation using Properties: i. Write-Only Property: Properties can be made write-only. This is accomplished by having only a set mutator in the property implementation.

```
using System;  
namespace Chapter4_Examples{  
    class Student{  
        string name, branch, usn;  
  
        public string Studusn{  
            set{ usn = value; }  
            get{return usn;}  
        }  
    }  
}
```

```
class GetSetDemol {  
    static void Main() {  
        Student st1 = new Student();  
        st1.Studusn = "1RX12MCA01";  
        Console.WriteLine("USN: " + st1.Studusn);  
        Console.ReadKey();  
    }  
}
```



Example:

```
class Student  
{  
string name, branch, usn;  
public string Studusn  
{ set{usn = value;} }  
}
```

In this example, the property “Studusn” is having only set (mutator) but not get (accessor). So Studusn can called as a “write-only property”

. ii. Creating Read-Only Fields: Properties can be made read-only. This is accomplished by having only get accessor in property implementation. Example 2.4: In the below example, the property “Studusn” is having only get (accessor) but not set(mutator). So can call Studusn as a “read-only property”.

```
class Student{ string name, branch, usn; public string Studusn{ get{usn = value;} }
```

```
using System;
namespace Class_Demos{
class Student{
string studusn;
public Student(){
studusn="1RX12MCA01";
}
public string Studusn{
get{
return studusn;
}
}
}
}
```

```
class ReadOnly{
static void Main(){
Student st1 = new Student();
Console.WriteLine("USN: "
+ st1.Studusn);
Console.ReadKey();
}
}
```

file:///C:/Users/Suma/
USN: 1RX12MCA01

“static” property: C# also supports “static” properties. The *static members* are accessed at the class level, not from an instance (object) of that class. Static properties are manipulated in the same manner as static methods, as seen here:

Example 2.5: Assume that the Student type defines a point of static data to represent the name of the institution studying these students. You may define a static property as follows:

```
using System; namespace
Class Demos{ class
Student
string name, branch, usn;
static string instName
public static string Institution

set instName value }
get return instName;
```

```
class StaticProperty{
static void Main(){
Student.Institution = "RNSIT";
Console.WriteLine("InstitutionName:"
+Student.Institution);
Console.ReadKey();
}
}
```

file:///C:/Users/Suma/Documents/
Institution Name: RNSIT

5

Write a short note on:

- i. Static data
- ii. Virtual method
- iii. base keyword

Static data members:

When we declare a static field inside a class, it can be initialized with a value or all un-initialized static fields automatically get initialized to their default values when the class is loaded at first time. Characteristics: It is visible only within the class, but its lifetime is the entire program. Only one copy of static data member will exist for the entire class and is shared by all the objects of that class. No matter how many objects of a class are created. All static variables are initialized to zero when the first object of its class is created.

```

using System;
namespace Examples {
    class Stddata {
        static int record=0;
        Stddata() {
            record++;
        }
        void printrecord() {
            Console.WriteLine ("No of stud record: {0}", record);
        }
        static void Main() {
            Stddata std1=new Stddata();
            Stddata std2=new Stddata();
            std1.printrecord();
            std2.printrecord ();
            Console.ReadLine();
        }
    }
}

```

ii. Virtual method

C# virtual method is a method that can be redefined in derived classes. In C#, a virtual method has an implementation in a base class as well as derived the class. It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class. A virtual method is created in the base class that can be overridden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overridden in the derived class using the override keyword. When a method is declared as a virtual method in a base class then that method can be defined in a base class and it is optional for the derived class to override that method. The overriding method also provides more than one form for a method. Hence it is also an example for polymorphism. When a method is declared as a virtual method in a base class and that method has the same definition in a derived class then there is no need to override it in the derived class. But when a virtual method has a different definition in the base class and the derived class then there is a need to override it in the derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

iii. base keyword

C#, base keyword is used to access fields, constructors and methods of base class.

C# base keyword: accessing base class field

```

using System;
public class Animal{
    public string color = "white";
}
public class Dog: Animal
{

```

```

string color = "black";
public void showColor()
{
    Console.WriteLine(base.color);
    Console.WriteLine(color);
}
}
public class TestBase
{
    public static void Main()
    {
        Dog d = new Dog();
        d.showColor();
    }
}

```

C# base keyword example: calling base class method

By the help of base keyword, we can call the base class method also. It is useful if base and derived classes defines same method. In other words, if method is overridden. If derived class doesn't define same method, there is no need to use base keyword. Base class method can be directly called by the derived class method.

```

using System;
public class Animal{
    public virtual void eat(){
        Console.WriteLine("eating...");
    }
}
public class Dog: Animal
{
    public override void eat()
    {
        base.eat();
        Console.WriteLine("eating bread...");
    }
}
public class TestBase
{
    public static void Main()
    {

```

```

    Dog d = new Dog();
    d.eat();
}
}

```

C# inheritance: calling base class constructor internally

Whenever you inherit the base class, base class constructor is internally invoked. **using** System;

```

public class Animal{
    public Animal(){
        Console.WriteLine("animal...");
    }
}

```

```

public class Dog: Animal
{
    public Dog()
    {
        Console.WriteLine("dog...");
    }
}

```

```

public class TestOverriding
{
    public static void Main()
    {
        Dog d = new Dog();
    }
}

```

6

Explain :

- a) Classical Inheritance
- b) Containment / Delegation model

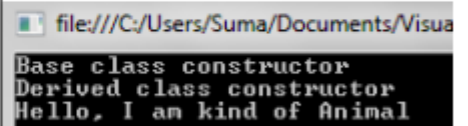
Classical inheritance (“is-a” relationship): When “is-a” relationship have established between classes, we are building a dependency between types. The basic idea behind classical inheritance is that new classes may extend the functionality of other classes.

Assume that we wish to define two additional classes to model Animal and Dog. The hierarchy looks like as shown below and we notice that Animal “is-a” Mammal, Dog IS-A Animal; Hence dog IS-A mammal as well. In “is-a” model, base classes are used to define general characteristics that are common to all

subclasses and classes are extended by using “:” operator. The derived classes inherit the base class's properties and methods and clients of the derived class have no knowledge of the base class.

```
using System;
namespace Chapter4_Examples{
    class Animal {
        public Animal() {
            Console.WriteLine ("Base class constructor" );
        }
        public void Greet() {
            Console.WriteLine ("Hello,I am kind of Animal" );
        }
    }
    class Dog : Animal{
        public Dog() {
            Console.WriteLine ("Derived class constructor" );
        }
    }
}
```

```
class isademo{
    static void Main() {
        Dog d = new Dog();
        d.Greet();
        Console.ReadKey();
    }
}
```



Containment / Delegation model (“Has-A”): The “HAS-A” relationship specifies how one class is made up of other classes

Consider we have two different classes Engine and a Car when both of these entities share each other’s object for some work and at the same time they can exist without each other’s dependency (having their own life time) and there should be no single owner both have to be independent from each other than type of relationship is known as "has-a" relationship i.e. Association.

```
using System;
namespace Chapter4_Examples{
    class Engine{
        public int horsepower;
        public void start() {
            Console.WriteLine ("Engine Started!" );
        }
    }
    class Car{
        public string make;
        public Engine eng; //Car has an Engine
        public void start() {
            eng.start();
        }
    }
}
```

```
class hasademo{
    static void Main() {
        Console.WriteLine("Manufacturing a Car");
        Car mycar = new Car();
        mycar.make = "Toyoto";
        Console.WriteLine("Manufacturing a Engine to start car");
        mycar.eng = new Engine();
        mycar.eng.horsepower = 220;
        Console.WriteLine("\n***Car Details***");
        Console.WriteLine("Brand:" + mycar.make);
        Console.WriteLine("Power: " + mycar.eng.horsepower);
        mycar.start();
        Console.Read();
    }
}
```

7 Make a short note on abstraction.

Abstraction is the process of hiding the details of a particular concept or object from a user and exposing only the essential features. The characteristics of abstraction are as follows: Decomposing complex systems into smaller components. Let’s learn about the abstract classes and methods in detail. → Managing the complexity of the code →

Abstract Classes: Classes can be declared as abstract by putting the keyword “abstract” before the class definitions. The main purpose of the Abstract classes is to make classes that only represent base classes, and don’t want anyone to create objects of these class types. An abstract class cannot be instantiated because cannot create an object of the class.

Characteristics: Restricts instantiation, implying that you cannot create an object of an abstract class.

Allows you to define abstract as well as non-abstract members in it
Requires at least one abstract method in it
Restrict the use of sealed keyword
Possesses public access specifier; therefore, it can be used anywhere in a program

Abstract methods:

Abstract methods have no implementation, so the method definitions is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods.

Characteristics:

Restricts its implementation in an abstract derived class
Allows implementation in a non-abstract derived class
Requires declaration in an abstract class only
Restrict declaration with static and virtual keywords
Allows you to override a virtual method.⌘

8 Explain :

- i. Method Overloading
- ii. Operator Overloading

i. Method overloading: In method overloading, can be define many methods with the same name but different signatures. A method signature is the combination of the method's name along with the number, type, and order of the parameters.

In this application, the Area() method of Shape class is overloaded for calculating the area of square, rectangle, circle and triangle shapes. In the Main() method, the Area()method is called multiple times by passing different arguments.

```
using System;
namespace Class_Demos{
    class Shape{
        public void Area(int side){
            Console.WriteLine("The area of Square is: " + side * side);
        }
        public void Area(int length, int breadth){
            Console.WriteLine("The area of Rectangle is: "
                + length * breadth);
        }
    }
}
```

```

public void Area(double radius){
    Console.WriteLine("The area of Circle is: "
        + 3.14 * radius * radius);
}
public void Area(double base1, double height){
    Console.WriteLine("The area of Squate is: "
        + (base1 * height)/2);
}
}
class MOverload{
    static void Main(){
        Shape shape = new Shape();
        shape.Area(15);
        shape.Area(10, 20);
        shape.Area(10.5);
        shape.Area(15.5, 20.4);
        Console.Read();
    }
}
}

```

9 Write a program to overload + and – operators for adding and subtracting two square matrices.

```

// Source Code starts
using System;

class Matrix
{
    public const int DimSize = 3;
    private double[,] m_matrix = new double[DimSize, DimSize];

    // allow callers to initialize
    public double this[int x, int y]
    {
        get { return m_matrix[x, y]; }
        set { m_matrix[x, y] = value; }
    }

    // let user add matrices
    public static Matrix operator +(Matrix mat1, Matrix mat2)
    {
        Matrix newMatrix = new Matrix();

        for (int x=0; x < DimSize; x++)
            for (int y=0; y < DimSize; y++)
                newMatrix[x, y] = mat1[x, y] + mat2[x, y];

        return newMatrix;
    }

    public static Matrix operator -(Matrix mat1, Matrix mat2)
    {
        Matrix newMatrix = new Matrix();

        for (int x=0; x < DimSize; x++)
            for (int y=0; y < DimSize; y++)
                newMatrix[x, y] = mat1[x, y] - mat2[x, y];

        return newMatrix;
    }
}

```

```

}

class MatrixTest
{
    // used in the InitMatrix method.
    public static Random m_rand = new Random();

    // test Matrix
    static void Main()
    {
        Matrix mat1 = new Matrix();
        Matrix mat2 = new Matrix();

        // init matrices with random values
        InitMatrix(mat1);
        InitMatrix(mat2);

        // print out matrices
        Console.WriteLine("Matrix 1: ");
        PrintMatrix(mat1);

        Console.WriteLine("Matrix 2: ");
        PrintMatrix(mat2);

        // perform operation and print out
        results
        Matrix mat3 = mat1 + mat2;

        Matrix mat4 = mat1 - mat2;
        Console.WriteLine();
        Console.WriteLine("Matrix 1 + Matrix
2 = ");
        PrintMatrix(mat3);
        PrintMatrix(mat4);
        Console.ReadLine();
    }

    // initialize matrix with random values
    public static void InitMatrix(Matrix mat)
    {
        for (int x=0; x < Matrix.DimSize; x++)
            for (int y=0; y < Matrix.DimSize; y++)
                mat[x, y] = m_rand.NextDouble();
    }

    // print matrix to console
    public static void PrintMatrix(Matrix mat)
    {
        Console.WriteLine();
        for (int x=0; x < Matrix.DimSize; x++)
        {
            Console.Write("[ ");
            for (int y=0; y < Matrix.DimSize; y++)
            {
                // format the output
                Console.Write("{0,8:#.000000}", mat[x, y]);

                if ((y+1 % 2) < 3)
                    Console.Write(", ");
            }
            Console.WriteLine(" ]");
        }
        Console.WriteLine();
    }
}

```

```
}
```

10 What is Interface? Explain how to create interfaces and implemented with an example.

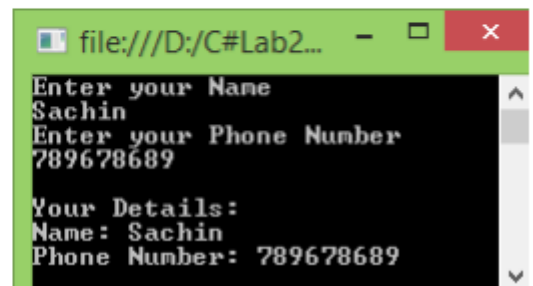
When an interface is implemented by a base class, then the derived class of the base class automatically inherits method of the interface. You can initialize an object of the interface by type casting the object of the derived class with the interface itself.

```
using System;
namespace Class_Demos{
    interface BaseInterface{
        void GetPersonalDetail();
        void GetContactDetail();
    }
    interface DerivedInterface:BaseInterface{
        void ShowDetail();
    }
}
```

```
class InterfaceImplementer : DerivedInterface{
    string name;
    long phonenum;
    public void GetPersonalDetail(){
        Console.WriteLine("Enter your Name");
        name = Console.ReadLine();
    }
    public void GetContactDetail(){
        Console.WriteLine("Enter your Phone Number");
        phonenum = int.Parse(Console.ReadLine());
    }
    public void ShowDetail(){
        Console.WriteLine("\nYour Details:");
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Phone Number: " + phonenum);
    }
}

class InterfaceDemol{
    static void Main(){
        InterfaceImplementer Myobj = new InterfaceImplementer();
        Myobj.GetPersonalDetail();
        Myobj.GetContactDetail();
        Myobj.ShowDetail();

        Console.ReadLine();
    }
}
```



```
file:///D:/C#Lab2...
Enter your Name
Sachin
Enter your Phone Number
789678689

Your Details:
Name: Sachin
Phone Number: 789678689
```