Section 4

***Question 1***

***What is shared memory and when will you use it?***  
Answer:  
Shared memory is a memory shared between two or more processes. Each process has its own address space; if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter-process communication) techniques.  
shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.

Use of Shared memory:

We Use IPC (inter-related process communication) through Shared Memory. inter-related process communication is performed using Pipes.

But the problem with pipes, FIFO, and message queue is that the information exchange between two processes goes through the kernel, and it works as follows.

* The server reads from the input file.
* The server writes this data in a message using pipe, FIFO, or message queue.
* The client reads the data from the IPC channel, again requiring the data to be copied from the kernel's IPC buffer to the client's buffer.
* Finally, the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory, the data is only copied twice, from the input file into shared memory and from shared memory to the output file.

***Question 2***

***What are the restrictions, if any, in C# OOP?***

Restrictions on Properties: We have to follow a few rules or restrictions while writing properties which are as follows:

A property cannot be passed via ref or out parameter to a method: Properties cannot be passed via out or ref, as properties, are actually methods. Both ref and out are not considered to be a part of method signature at the time of compilation. Due to this, it will give a compile-time error. There is no solution to this restriction except to remove ref or out from the program entirely.

You cannot overload a property: A property cannot be overloaded. It means that one can only put a single get and set accessor and mutator in a class respectively. The program given below shows what happens when we give more than one get accessor in the same class.

***Question 3***

***Give an example, in your own words, of polymorphism and when it should be used.***

Polymorphism meaning "one name many forms". In other words, one object has many forms or has one name with multiple functionalities. "Poly" means many and "morph" means forms. Polymorphism provides the ability to a class to have multiple implementations with the same name.

Method overloading is an example of Static Polymorphism. In overloading, the method / function has a same name but different signatures. It is also known as Compile Time Polymorphism because the decision of which method is to be called is made at compile time. Overloading is the concept in which method names are the same with a different set of parameters.

Here C# compiler checks the number of parameters passed and the type of parameter and make the decision of which method to call and it throw an error if no matching method is found.

In the following example, a class has two methods with the same name "Add" but with different input parameters (the first method has three parameters and the second method has two parameters).

public class TestData

{

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

{

return a + b + c;

}

public int Add(int a, int b)

{

return a + b;

} }

class Program

{

static void Main(string[] args)

{

TestData dataClass = new TestData();

int add2 = dataClass.Add(45, 34, 67);

int add1 = dataClass.Add(23, 34);

}

}

In c#, Run Time Polymorphism means overriding a base class method in the derived class by creating a similar function. This can be achieved by using override & virtual keywords and the inheritance principle.

Using run-time polymorphism, we can override a base class method in the derived class by creating a method with the same name and parameters to perform a different task.

In c#, the run time polymorphism can be achieved by using method overriding, and it is also called late binding or dynamic binding.

Following is the code snippet of implementing a method overriding to achieve run time polymorphism in c#.

// Base Class

public class Users

{

public virtual void GetInfo()

{

Console.WriteLine("Base Class");

}

}

// Derived Class

public class Details: Users

{

public override void GetInfo()

{

Console.WriteLine("Derived Class");

}

}

***Question 4***

***Please provide a practical example of interface implementation in C#, describing exactly what benefits***

***you would achieve through your design.When to use Interface?***

Arithmetic operations using the interface.

Code:

using System;

namespace arth\_interface {

public interface SampleInterface {

void sam\_add(int a, int b);

void sam\_sub(int a, int b);

void display();

}

class interface\_class : SampleInterface {

int x, y;

public void sam\_add(int a, int b) {

int m, n;

m = a;

n = b;

x = m + n;

}

public void sam\_sub(int a, int b) {

int m, n;

m = a;

n = b;

y = a - b;

}

public void display() {

Console.WriteLine("Added Value is:" + x);

Console.WriteLine("Subtracted value is:" + y);

}

}

class arth\_interface {

static void Main(string[] args) {

interface\_class obj\_interface\_class = new interface\_class();

int fnumber, snumber;

Console.WriteLine("Please Enter 1st Number to perform Addition and Subtraction:");

fnumber = Convert.ToInt16(Console.ReadLine());

Console.WriteLine("Now 2nd Number to perform Addition and Subtraction:");

snumber = Convert.ToInt16(Console.ReadLine());

obj\_interface\_class.sam\_add(fnumber, snumber);

obj\_interface\_class.sam\_sub(fnumber, snumber);

obj\_interface\_class.display();

Console.ReadKey();

}

}

}

Below are some of the advantages of Interface in C#:

One of the major advantages of Interface in C# is a better alternative to implement multiple inheritances.

* The interface enables the plug-and-play method.
* Complete Abstraction can be achieved by the implementation of Interface.
* Along with making our code easy to maintain, concept loose coupling can be achieved.

Security: When we have to simply hide some features and have to use those later. It is essential to hide a few details while only showing the details important to the user.

Multiple Inheritance: In c#, one class can inherit from a simple parent class, inheriting all its features. Multiple Inheritance is not supported in C# for the simple reason to not make C# complex. But with the use of an interface, multiple interfaces can be implemented into a single class.

***Question 5***

***In your experience what aspects of object orientated development has yielded the most***

***maintainable, versatile, flexible solution. Provide examples where possible.***

Here’s a look at some of OOP’s top benefits:

1. Modularity for easier troubleshooting

When working with object-oriented programming languages, you know exactly where to look when something goes wrong. “Oh, the car object broke down? The problem must be in the Car class!” You don’t have to go line-by-line through all your code.

That’s the beauty of encapsulation. Objects are self-contained, and each bit of functionality does its own thing while leaving the other bits alone. Also, this modularity allows an IT team to work on multiple objects simultaneously while minimizing the chance that one person might duplicate someone else’s functionality.

2. Reuse of code through inheritance

Suppose that in addition to your Car object, one colleague needs a RaceCar object, and another needs a Limousine object. Everyone builds their objects separately but discover commonalities between them. In fact, each object is just a different kind of Car. This is where the inheritance technique saves time: Create one generic class (Car), and then define the subclasses (RaceCar and Limousine) that are to inherit the generic class’s traits.

Of course, Limousine and RaceCar still have their unique attributes and functions. If the RaceCar object needs a method to “fireAfterBurners” and the Limousine object requires a Chauffeur, each class could implement separate functions just for itself. However, because both classes inherit key aspects from the Car class, for example the “drive” or “fillUpGas” methods, your inheriting classes can simply reuse existing code instead of writing these functions all over again.

What if you want to make a change to all Car objects, regardless of type? This is another advantage of the OOP approach. Make a change to your Car class, and all car objects will simply inherit the new code.

3. Flexibility through polymorphism

Riffing on this example, you now need just a few drivers, or functions, like “driveCar,” driveRaceCar” and “DriveLimousine.” RaceCarDrivers share some traits with LimousineDrivers, but other things, like RaceHelmets and BeverageSponsorships, are unique.

This is where object-oriented programming’s polymorphism comes into play. Because a single function can shape-shift to adapt to whichever class it’s in, you could create one function in the parent Car class called “drive” — not “driveCar” or “driveRaceCar,” but just “drive.” This one function would work with the RaceCarDriver, LimousineDriver and so on. In fact, you could even have “raceCar.drive(myRaceCarDriver)” or “limo.drive(myChauffeur).”

4. Effective problem solving

Many people avoid learning OOP because the learning curve seems steeper than that for top-down programming. But take the time to master OOP and you’ll find it’s the easier, more intuitive approach for developing big projects.

Object-oriented programming is ultimately about taking a huge problem and breaking it down to solvable chunks. For each mini-problem, you write a class that does what you require. And then — best of all — you can reuse those classes, which makes it even quicker to solve the next problem.

This isn’t to say that OOP is the only way to write software. But there’s a reason that languages like C++, C# and Java are the go-to options for serious software development.