C# Solution: Dec-2017

PART - A

1. What is .NET Framework? List and explain its features?

- **Full Interoperability with Existing Win32code**
  - Existing COM binaries can inter-operate with newer .NET binaries & vice versa.
  - Also, PInvoke(Platform invocation) allows to invoke raw C-based functions from managed-code. **ii. Complete & Total Language Integration**
- .NET supports
  - cross language inheritance
  - cross language exception-handling &
  - cross language debugging
- **A Common Runtime Engine Shared by all .NET-aware languages**
  - Main feature of this engine is "a well-defined set of types that each .NET-aware language understands".
- **A Base Class Library that**
  - protects the programmer from the complexities of raw API calls
  - offers a consistent object-model used by all .NET-aware languages
- **A Truly Simplified Deployment Model**
  - Under .NET, there is no need to register a binary-unit into the system-registry.
  - .NET runtime allows multiple versions of same *.dll to exist in harmony on a single machine.
- **No more COM plumbing** IClasFactory, IUnknown, IDispatch, IDL code and the evil VARIANT compliant data-types have no place in a native .NET binary.

C# language offers the following features

- No pointer required. C# programs typically have no need for direct pointer manipulation.
- Automatic memory management through garbage-collection. Given this, C# does not support a delete keyword.
- Formal syntactic-constructs for enumerations, structures and class properties.
- C++ like ability to overload operators for a custom type without the complexity.
- Full support for interface-based programming techniques.
- Full support for aspect-based programming techniques via attributes. This brand of development allows you to assign characteristics to types and their members to further qualify the behavior.

2. What is metadata? Describe its role in .NET framework?

Metadata is information which holds the information of other data types. A .NET assembly contains full, complete, and accurate metadata, which describes each and every type living within the binary:

- Class
- Structure
- Enumeration, defined in the binary,

As well as the members of each¬ type like:

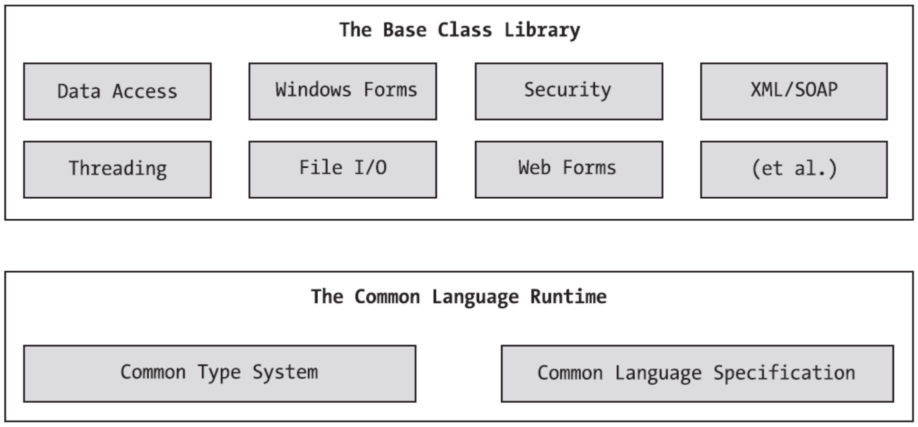
- Properties
- Methods
- Events

For example, if you have a class named Sports car, the type metadata describes details such as Sports Car's base class, which interfaces are implemented by Sports Car (if any), as well as a full description of each member supported by the Sports Car type. It is always the job of the compiler (not the programmer) to emit the latest and greatest type metadata. Finally, in addition to CIL and type metadata, assemblies themselves are also described using metadata, which is officially termed a manifest. The manifest contains information about the current version of the assembly, culture information (used for localizing string and image resources),

3. Describe the building blocks of .NET Framework?

- **Common language runtime or CLR**
  - The primary role of the CLR is to locate, load, and manage .NET types on your behalf. The CLR also takes care of a number of low-level details such as memory management and performing security checks.
- **Common Type System or CTS**
  - The CTS specification fully describes all possible data types and programming constructs supported by the runtime, specifies how these entities can interact with each other, and details how they are represented in the .NET metadata format.
- **Common Language Specification or CLS**
  - The CLS is a related specification that defines a subset of common types and programming constructs that all .NET programming languages can agree on.
  - Thus, if you build .NET types that only expose CLS-compliant features, you can rest assured that all .NET-aware languages can consume them.
- **The Role of the Base Class Libraries**

- The .NET platform provides a base class library that is available to all .NET programming languages.
- Base class library encapsulate various primitives such as threads, file input/output (I/O), graphical rendering, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications.
- For example, the base class libraries define types that facilitate database access, XML manipulation, programmatic security, and the construction of web-enabled front ends.



The CLR, CTS, CLS, and base class library relationship

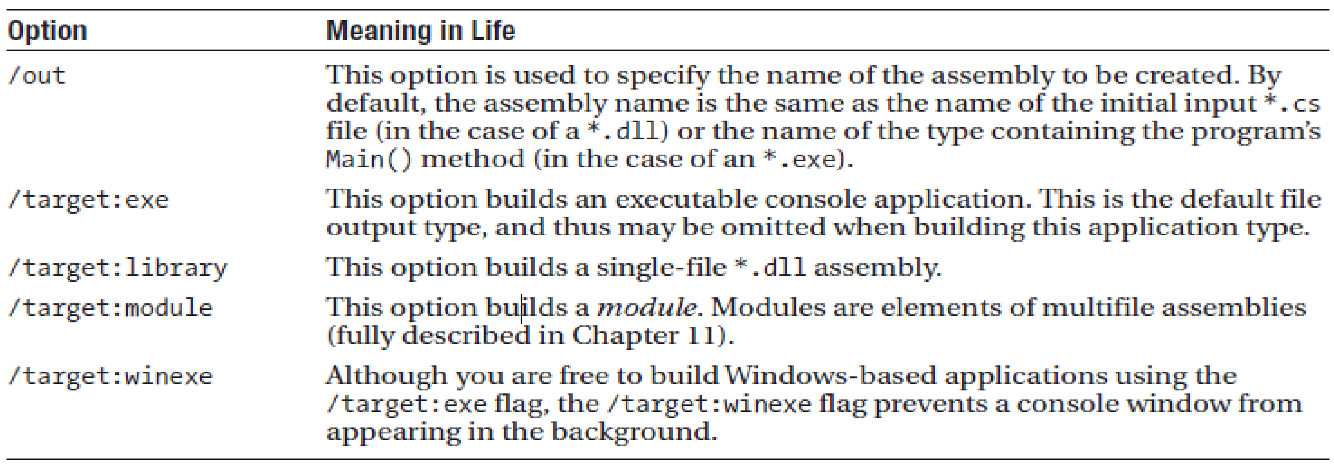
4. Explain how to build C# applications using CSE.EXE?

To build a simple single file assembly named TestApp.exe using the C# command-line compiler and Notepad. First, you need some source code. Open Notepad and enter the following:

```
// A simple C# application.
using System;
class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
}
}
```

Once we have finished, save the file in a convenient location (e.g., C:\CscExample) as TestApp.cs. Each possibility is represented by a specific flag passed into csc.exe as a command-line parameter see below table which are the core options of the C# compiler.

Output-centric Options of the C# Compiler



Option	Meaning in Life
`/out`	This option is used to specify the name of the assembly to be created. By default, the assembly name is the same as the name of the initial input `*.cs` file (in the case of a `*.dll`) or the name of the type containing the program's `Main()` method (in the case of an `*.exe`).
`/target:exe`	This option builds an executable console application. This is the default file output type, and thus may be omitted when building this application type.
`/target:library`	This option builds a single-file `*.dll` assembly.
`/target:module`	This option builds a *module*. Modules are elements of multifile assemblies (fully described in Chapter 11).
`/target:winexe`	Although you are free to build Windows-based applications using the `/target:exe` flag, the `/target:winexe` flag prevents a console window from appearing in the background.

To compile TestApp.cs into a console application named TestApp.exe enter

csc /**target:exe** TestApp.cs

C# compiler flags support an abbreviated version, such as **/t** rather than **/target**

csc /**t:exe** TestApp.cs

default output used by the C# compiler, so compile TestApp.cs simply by typing

csc TestApp.cs

TestApp.exe can now be run from the command line a shows o/p as;

C:\TestApp

Testing! 1, 2, 3

Referencing External Assemblies

- To compile an application that makes use of types defined in a separate .NET assembly. Reference to the System.Console type mscorlib.dll is *automatically referenced* during the compilation process.
- To illustrate the process of referencing external assemblies the TestApp application to display windows Forms message box.
- At the command line, you must inform csc.exe which assembly contains the "used" namespaces.
- Given that you have made use of the MessageBox class, you must specify the **System.Windows.Forms.dll** assembly using the /reference flag (which can be abbreviated to /r):

  csc /**r:System.Windows.Forms.dll** testapp.cs

Compiling Multiple Source Files with csc.exe

Most projects are composed of multiple *.cs files to keep code base a bit more flexible. Assume you have class contained in a new file named HelloMsg.cs:

```
// The HelloMessage class
using System;
using System.Windows.Forms;
```

```
class HelloMessage
{
public void Speak(){
MessageBox.Show("Hello...");
}
 }
```

Now, create TestApp.cs file & write below code

```
using System;
class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
HelloMessage h = new HelloMessage();
h.Speak();
}
}
```

You can compile your C# files by listing each input file explicitly:

csc /r:System.Windows.Forms.dll **testapp.cs hellomsg.cs**

As an alternative, csc /r:System.Windows.Forms.dll ***.cs**

5. Write a program to display message box dialogue containing “Hello World” message using class?

The HelloMessage class

```
using System;
using System.Windows.Forms;
class HelloMessage
{
public void Speak(){
MessageBox.Show("Hello World");
}
 }
```

Now, create TestApp.cs file & write below code

```
using System;
class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
HelloMessage h = new HelloMessage();
h.Speak();
}
}
```

6. **Write a program to count the number of object instances create inside or outside of an assembler?**

```
using System;
class object1
{
        static int ob = 0;
        public object1()
        {
                ob = ob + 1;
        }
        public static void Main(String[] args)
        {
                object1 ob1 = new object1();
                object1 ob2 = new object1();
                object1 ob3 = new object1();
                System. Console. WriteLine("Num of objected created are = {0}",ob);

        }

}
```

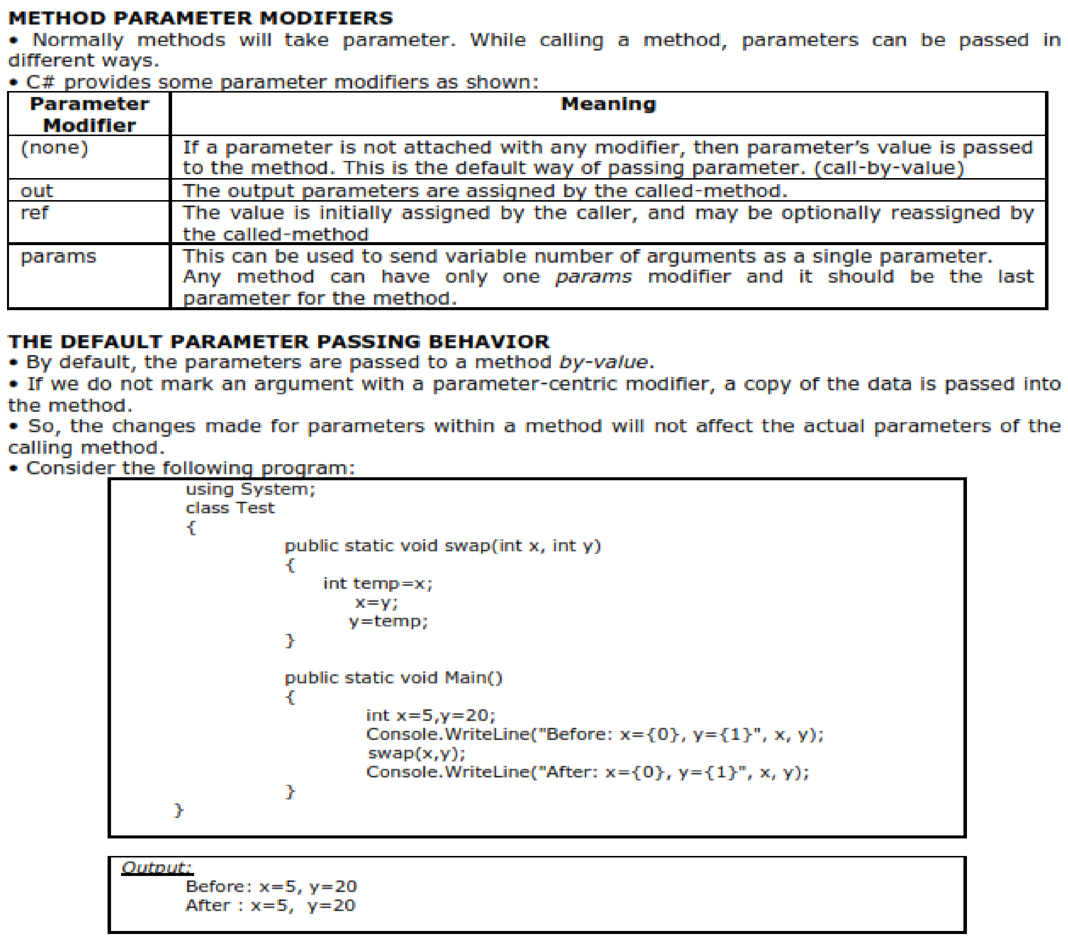
7. **How do you process command line arguments in C# programs? Give an example?**

```
using System;
class Program
{
   Public static void Main(string[] args)
   {
       Console.WriteLine("--- Primes between 0 and 100 ---");
       for (int i = 0; i < args.length; i++)
       {
          bool prime = PrimeTool.IsPrime(i);
          if (prime)
          {
                  Console.Write("Prime: ");
                  Console.WriteLine(i);
          }
       }
     }
}
```

Class that contains IsPrime: PrimeTool.cs, C#

```
using System;
public static class PrimeTool
{
   public static bool IsPrime(int candidate)
   {
     // Test whether the parameter is a prime number.
     if ((candidate & 1) == 0)
     {
        if (candidate == 2)
        {
            return true;
        }
        else
        {
            return false;
        }
     }
     for (int i = 3; (i * i) <= candidate; i += 2)
     {
        if ((candidate % i) == 0)
        {
            return false;
        }
     }
     return candidate != 1;
   }
}
```

8. What are method parameter modifiers? Name and explain all the available method parameter modifiers in C#? Illustrate the use of method parameter modifiers with an example?



METHOD PARAMETER MODIFIERS

- Normally methods will take parameter. While calling a method, parameters can be passed in different ways.
- C# provides some parameter modifiers as shown:

Parameter Modifier	Meaning
(none)	If a parameter is not attached with any modifier, then parameter's value is passed to the method. This is the default way of passing parameter. (call-by-value)
out	The output parameters are assigned by the called-method.
ref	The value is initially assigned by the caller, and may be optionally reassigned by the called-method
params	This can be used to send variable number of arguments as a single parameter. Any method can have only one *params* modifier and it should be the last parameter for the method.

THE DEFAULT PARAMETER PASSING BEHAVIOR

- By default, the parameters are passed to a method *by-value*.
- If we do not mark an argument with a parameter-centric modifier, a copy of the data is passed into the method.
- So, the changes made for parameters within a method will not affect the actual parameters of the calling method.
- Consider the following program:

```
    using System;
    class Test
    {
            public static void swap(int x, int y)
            {
                int temp=x;
                   x=y;
                  y=temp;
            }

            public static void Main()
            {
                    int x=5,y=20;
                    Console.WriteLine("Before: x={0}, y={1}", x, y);
                    swap(x,y);
                    Console.WriteLine("After: x={0}, y={1}", x, y);
            }
   }
```

Output:

```
    Before: x=5, y=20
    After : x=5,  y=20
```

out KEYWORD

• Output parameters are assigned by the called-method.

• In some of the methods, we need to return a value to a calling-method. Instead of using *return* statement, C# provides a modifier for a parameter as *out*.

• Consider the following program:

```
using System;
class Test
{
        public static void add(int x, int y, out int z)
        {
                z=x+y;
        }

        public static void Main()
        {
                int x=5,y=20, z;
                add(x, y, out z);
                Console.WriteLine("z={0}", z);
        }
}
```

Output:

```
z=25
```

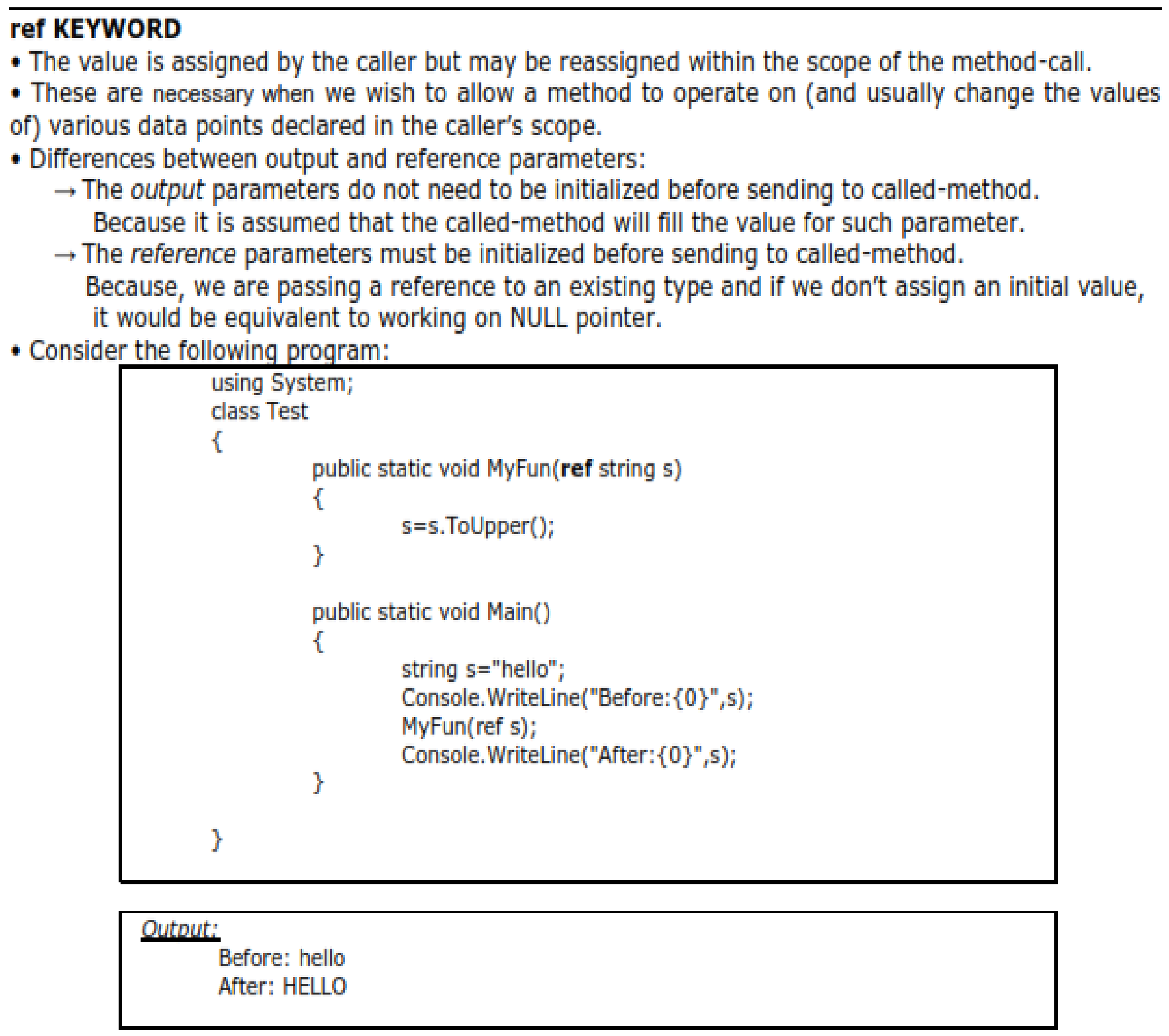
• Useful purpose of out: It allows the caller to obtain multiple return values from a single method-invocation.

• Consider the following program:

```
using System;
class Test
{
        public static void MyFun(out int x, out string y, out bool z)
        {
                x=5;
                y="Hello, how are you?";
                z=true;
        }

        public static void Main()
        {
                int a;
                string str;
                bool b;

                MyFun(out a, out str, out b);
                Console.WriteLine("integer={0} ", a);
                Console.WriteLine("string={0}", str);
                Console.WriteLine("boolean={0} ", b);

        }
}
```

Output:

```
integer=5,
string=Hello, how are you?
boolean=true
```



ref KEYWORD

- The value is assigned by the caller but may be reassigned within the scope of the method-call.
- These are necessary when we wish to allow a method to operate on (and usually change the values of) various data points declared in the caller's scope.
- Differences between output and reference parameters:
  - → The *output* parameters do not need to be initialized before sending to called-method. Because it is assumed that the called-method will fill the value for such parameter.
  - → The *reference* parameters must be initialized before sending to called-method. Because, we are passing a reference to an existing type and if we don't assign an initial value, it would be equivalent to working on NULL pointer.
- Consider the following program:

```
using System;
class Test
{
        public static void MyFun(ref string s)
        {
                s=s.ToUpper();
        }

        public static void Main()
        {
                string s="hello";
                Console.WriteLine("Before:{0}",s);
                MyFun(ref s);
                Console.WriteLine("After:{0}",s);
        }

}
```

Output:

```
Before: hello
After: HELLO
```

9. Briefly describe the support C# language for the pillars of object oriented programming?

Object Oriented Programming

A programming model which is mainly organized around the objects is called Object Oriented Programming or the Programming where the main focus of programmer while modeling any problem is objects. (The name OOP indicates the same).

Encapsulation

Encapsulation is a process of binding data members (variables, properties) and member functions (methods) together. In object oriented programming language we achieve encapsulation through Class.

Real Example

The real life example of encapsulation will be the Capsule. Capsule binds all chemical contents required for curing specific disease together just like the class which binds data members and member functions.

What Is Abstraction?

Abstraction is the process of showing only essential/necessary features of an entity/object to the outside world and hides the other irrelevant information. In programming language we achieve the abstraction through public and private access modifiers and a class. So in a class make things (feature) which we want to show as public and thing which are irrelevant make them as private so they won't be available.

Real Example

Real life example of Abstraction could be the gears of bike. Do we know what happens inside the engine when we change the gear ? Answer is No (what happens inside the engine when we change the gear is irrelevant information from user perspective so we can hide that information). What important from users perspective is whether gear has been changed or not. (This is essential/necessary feature that must be shown to the user).

Inheritance

The abstraction says only expose that details which really matters from users perspective and hide the other details. The process of creating the new class by extending the existing class is called inheritance or the process of inheriting the features of base class is called as inheritance. The existing class is called the base class and new class which is created from it is called the derived class.

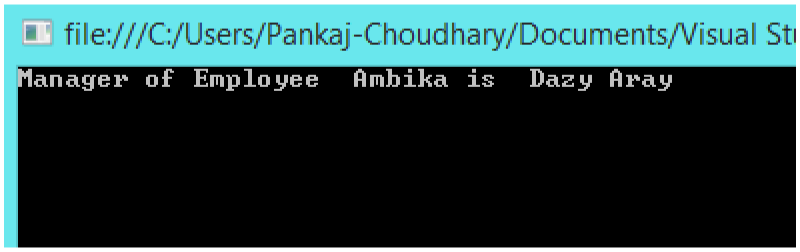
We inherit some of our features (may be body color, nose shape, height of body etc) from Mom and Dad. The most important advantage of inheritance is code re-usability. In Inheritance the derived class possess all attributes/properties and functions of base class, this is where code re-usability comes into the picture.

10. What is has-a relationship? How it is implemented in C#?

Association is a “has-a” type relationship. Association establishes the relationship b/w two classes using through their objects. Association relationship can be one to one, One to many, many to one and many to many. For example suppose we have two classes then these two classes are said to be “has-a” relationship if both of these entities share each other’s object for some work and at the same time they can exists without each other’s dependency or both have their own life time.

```
class Employee
    {
        public Employee() { }
        public string Emp_Name { get; set; }

        public void Manager_Name(Manager Obj)
        {
            Obj.manager_Info(this);
        }
    }

    class Manager
    {

        public Manager() { }
        public string Manager_Name { get; set; }
        public void manager_Info(Employee Obj)
        {
            WriteLine($"Manager of Employee  {Obj.Emp_Name} is  {this.Manager_Name}");

        }


    }
    class Program
    {


        static void Main(string[] args)
        {
            Manager Man_Obj = new Manager();
            Man_Obj.Manager_Name = "Dazy Aray";
            Employee Emp_Obj = new Employee();
            Emp_Obj.Emp_Name = "Ambika";
            Emp_Obj.Manager_Name(Man_Obj);
            ReadLine();
        }
    }

```

Output



```
Manager of Employee  Ambika is  Dazy Aray
```

11. What are the advantages of using virtual and override keywords in C# programs? Give one example?

```
using System;
using System.Collections.Generic;
using System.Text;
        public class Customer
         {
                public virtual void CustomerType()
                {

                Console.WriteLine("I am customer");
                }
        }
         public class CorporateCustomer : Customer
```

```
{
        public override void CustomerType()
        {
        Console.WriteLine("I am Corporate Customer");
        }
}
public class PersonalCustomer : Customer
{
        public override void CustomerType()
        {
        Console.WriteLine("I am personal customer");
        }
}

Class VirtualPgm
{
        Static void Main(string[] args)
        {
                Customer[] c = new Customer[3];
                c[0] = new CorporateCustomer();
                c[1] = new CorporateCustomer();
                c[2] = new Customer();
                foreach (Customer customerobject in c)
                {
                customerobject.CustomerType ();
                }
                Console.ReadLine ();
        }
}
```

OUTPUT:

```
I am Corporate Customer
I am Corporate Customer
I am customer
```

PART – B

1. **What are bugs, errors and exceptions? List and explain core members of System. Exception class?**

System.Exception Property	**Meaning in Life**
Data	This property retrieves a collection of key/value pairs (represented by an object implementing IDictionary) that provides additional, programmer-defined information about the exception. By default, this collection is empty (e.g., null).
HelpLink	This property returns a URL to a help file or website describing the error in full detail.
InnerException	This read-only property can be used to obtain information about the previous exception(s) that caused the current exception to occur. The previous exception(s) are recorded by passing them into the constructor of the most current exception.
Message	This read-only property returns the textual description of a given error. The error message itself is set as a constructor parameter.
Source	This property returns the name of the assembly that threw the Exception.
StackTrace	This read-only property contains a string that identifies the sequence of calls that triggered the exception. As you might guess, this property is very useful during debugging if you wish to dump the error to an external error log.
TargetSite	This read-only property returns a MethodBase type, which describes numerous details about the method that threw the exception (invoking ToString() will identify the method by name).
InnerException	This read-only property can be used to obtain information about the previous exception(s) that caused the current exception to occur.

2. **What is object life time? Explain the garbage collection optimizations process in C#?**

When the CLR is attempting to locate unreachable objects, is does *not* literally examine each and every object placed on the managed heap. Obviously, doing so would involve considerable time, especially in larger (i.e., real-world) applications. To help optimize the process, each object on the heap is assigned to a specific "generation."The idea behind generations is simple: The longer an object has existed on the heap, the more likely it is to stay there. For example, the object implementing Main() will be in memory until the program terminates. Conversely, objects that have been recently placed on the heap are likely to be unreachable rather quickly (such as an object created within a method scope). Given these assumptions, each object on the heap belongs to one of the following generations:

- ***Generation 0***: Identifies a newly allocated object that has never been marked for collection

- ***Generation 1***: Identifies an object that has survived a garbage collection (i.e., it was marked for collection, but was not removed due to the fact that the sufficient heap space was acquired)
- ***Generation 2***: Identifies an object that has survived more than one sweep of the garbage collector The garbage collector will investigate all generation 0 objects first. If marking and sweeping these objects results in the required amount of free memory, any surviving objects are promoted to generation 1. To illustrate how an object's generation affects the collection process.

3. Write a C# application to illustrate handling multiple exceptions?

```
    static void Main(string[] args)

    {
            Console.WriteLine("***** Handling Multiple Exceptions *****\n");
            Car myCar = new Car("Rusty", 90);
            myCar.CrankTunes(true);
    try
    {
            // Speed up car logic.
    }
    catch(CarIsDeadException e)
    {
            // Process CarIsDeadException.
    }
    catch(ArgumentOutOfRangeException e)
    {
            // Process ArgumentOutOfRangeException.
    }
    catch(Exception e)
    {
            // Process any other Exception.
    }
    finally
    {
            // This will always occur. Exception or not.
            myCar.CrankTunes(false);
    }
    Console.WriteLine();
}
```

4. What is an interface? Explain with an example implementations of interfaces in C#?

Interface implementation can also be very helpful whenever you are implementing a number of interfaces that happen to contain identical members. For example, assume you wish to create a class that implements all the following new interface types:

```
public interface
{
        void Draw();
}
```

```
public interface IDrawToPrinter
{
        void Draw();
}
```

If you wish to build a class named SuperImage that supports basic rendering (IDraw), 3D rendering (IDraw3D), as well as printing services (IDrawToPrinter), the only way to provide unique implementa implementation:

```
// Not deriving from Shape, but still injecting a name clash.
public class SuperImage : IDraw, IDrawToPrinter, IDraw3D
{
        void IDraw.Draw()
        { /* Basic drawing logic. */ }
void IDrawToPrinter.Draw()
{ /* Printer logic. */ }
void IDraw3D.Draw()
{ /* 3D rendering logic. */ }
}
```

5. **Explain in detail the IComparable interface along with its different supporting methods?**

System. IComparable interface specifies a behavior that allows an object to be sorted based on some specified key. Here is the formal definition:

```
// This interface allows an object to specify its
// relationship between other like objects.
public interface IComparable{
int CompareTo(object o);
}
```

Let's assume you have updated the Car class to maintain an internal ID number (represented by a simple integer named carID) that can be set via a constructor parameter and manipulated using a new property named ID. Here are the relevant updates to the Car type:

```
public class Car
{
...
private int carID;
public int ID
{
get { return carID; }
set { carID = value; }
}
public Car(string name, int currSp, int id)
{
currSpeed = currSp;
petName = name;
carID = id;
}
...
}
```

Object users might create an array of Car types as follows:

```
static void Main(string[] args)
{
// Make an array of Car types.
Car[] myAutos = new Car[5];
myAutos[0] = new Car("Rusty", 80, 1);
myAutos[1] = new Car("Mary", 40, 234);
myAutos[2] = new Car("Viper", 40, 34);
myAutos[3] = new Car("Mel", 40, 4);
myAutos[4] = new Car("Chucky", 40, 5);
}
```

As you recall, the System.Array class defines a static method named Sort(). When you invoke this method on an array of intrinsic types (int, short, string, etc.), you are able to sort the items in the array in numerical/alphabetic order as these intrinsic data types implement IComparable. However, what if you were to send an array of Car types into the Sort()method as follows?

```
// Sort my cars?
Array.Sort(myAutos);
```

If you run this test, you would find that an Argument Exception is thrown by the runtime, with the following message: “At least one object must implement IComparable.” When you build custom types you can implement IComparableto allow arrays of your types to be sorted. When you flesh out the details of CompareTo(), it will be up to you to decide what the baseline of the ordering operation will be. For the Cartype, the internal carIDseems to be the most logical candidate:

```
// The iteration of the Car can be ordered
// based on the CarID.
public class Car : IComparable
{
...
// IComparable implementation.
int IComparable.CompareTo(object obj){
Car temp = (Car)obj;
if(this.carID > temp.carID)
return 1;
if(this.carID < temp.carID)
return -1;
else
return 0;
}
}
```

6. **How do you build ICloneable interface types? Give one example?**

System.Objectdefines a member named MemberwiseClone (). This method Object users do not call this method directly (as it is protected); however, a given object may call this method itself during the *cloning* process. To illustrate, assume you have a class named Point:

```
// A class named Point.
public class Point
{
// Public for easy access.
public int x, y;
public Point(int x, int y) { this.x = x; this.y = y;}
public Point(){}
// Override Object.ToString().

public override string ToString()
{ return string.Format("X = {0}; Y = {1}", x, y ); }
}
```

Given what you already know about reference types and value types, you are aware that if you assign one reference variable to another, you have two references pointing to the same object in memory. Thus, the following assignment operation Point object on the heap; modifications using either reference affect the same object on the heap:

```
static void Main(string[] args)
{
// Two references
Point p1 = new Point(50, 50);
Point p2 = p1;
p2.x = 0;
Console.WriteLine(p1);
Console.WriteLine(p2);
}
```

When you wish to equip your custom types to support the ability to return an identical copy of itself to the caller, you may implement the standard ICloneable interface. This type defines a single method named Clone():

```
public interface ICloneable
{
Object Clone();
}
```

7. **What are delegates?**

Historically speaking, the Windows API makes frequent use of C-style function pointers to create entities termed *callback functions* or simply *callbacks*. Using callbacks, programmers were able to configure one function to report back to (call back) another function in the application. The problem with standard C-style callback functions is that they represent little more than a raw address in memory. Ideally, callbacks could be configured to include additional type-safe information such as the number of (and types of) parameters and the return value (if any) of the method pointed to. Sadly, this is not the case in traditional callback functions, and, as you may suspect, can therefore be a frequent source of bugs, hard crashes, and other runtime disasters. Nevertheless, callbacks are useful entities. In the .NET Framework, callbacks are still possible, and their functionality is accomplished in a much safer and more object oriented manner using *delegates*. In essence, a delegate is a type-safe object that points to another method (or possibly multiple methods) in the application, which can be invoked at a later time. Specifically speaking, a delegate type maintains three important pieces of information:

- The *name* of the method on which it makes calls
- The *arguments* (if any) of this method
- The *return value* (if any) of this method

Defining a Delegate in C#

When you want to create a delegate in C#, you make use of the delegate keyword. The name of your define the delegate to match the signature of the method it will point to. For example, assume you wish to build a delegate named BinaryOp returns an integer and takes two integers as input parameters: **//
This delegate can point to any method,**
// taking two integers and returning an
// integer.
public delegate int BinaryOp(int x, int y);

When the C# compiler processes delegate types, it automatically generates a sealed ing from System.MulticastDelegate. This class (in conjunction with its base class, System.Delegate) provides the necessary infrastructure for the delegate to hold onto the list of methods to be invoked BinaryOp delegate using ildasm.exe

8. **With an example discuss the advance keywords in C#?**

- **Checked**

  The checked and unchecked operators are used to control the overflow checking context for integral-type arithmetic operations and conversions. It checks, if there is an overflow (this is default).

By default, an expression that contains only constant values causes a compiler error if the expression produces a value that is outside the range of the destination type. If the expression contains one or more non-constant values, the compiler does not detect the overflow. Evaluating the expression assigned to i2 in the following example does not cause a compiler error. The following example causes compiler error CS0220 because 2147483647 is the maximum value for integers.

```
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause a compiler
error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does not cause a run-time
exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

- **Unchecked**

The unchecked keyword prevents overflow-checking when doing integer arithmetic's. It may be used as an *operator* on a single expression or as a statement on a whole block of code.

```
int x, y, z;
x = 1222111000;
y = 1222111000;

// used as an operator
z = unchecked(x*y);

// used as a statement
unchecked {
  z = x*y;
  x = z*z;
}
```

- **Sealed**

Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a **sealed class,** the class cannot be inherited. In C#, the sealed modifier is used to define a class as **sealed**. In Visual Basic .NET the**NotInheritable** keyword serves the purpose of sealed. If a class is derived from a sealed class then the compiler throws an error. If you have ever noticed, structs are sealed. You cannot derive a class from a struct.

```
// Sealed class
```

```
sealed class SealedClass
{

}
```

In the following code, will get error.

```
using System;
class Class1
{
   static void Main(string[] args)
   {
      SealedClass sealedCls = new SealedClass();
      int total = sealedCls.Add(4, 5);
      Console.WriteLine("Total = " + total.ToString());
   }
}
// Sealed class
sealed class SealedClass
{
   public int Add(int x, int y)
   {
      return x + y;
   }
}
```

- **Stackalloc**

  The keyword stackalloc is used in an unsafe code context to allocate a block of memory on the stack.

  - int* fib = stackalloc int[100];
  - In the example above, a block of memory of sufficient size to contain 100 elements of type int is allocated on the stack, not the heap; the address of the block is stored in the pointer fib. This memory is not subject to garbage collection and therefore does not have to be pinned (via fixed). The lifetime of the memory block is limited to the lifetime of the method in which it is defined (there is no way to free the memory before the method returns).
  - stackalloc is only valid in local variable initializers.
  - Because Pointer types are involved, stackalloc requires unsafe context. See Unsafe Code and Pointers.
  - stackalloc is similar to _alloca in the C run-time library

- **Volatile**

  The volatile keyword is used to declare a variable that may change its value over time due to modification by an outside process, the system hardware, or another concurrently running thread. You should use this modifier in your member variable declaration to ensure that whenver the value is read, you are always getting the most recent (up-to-date) value of the variable.

```
class MyClass
{
  public volatile long systemclock;
}
```

9. Write short notes on following:

a. .NET framework assembly format

Part	Description
PE Header	PE32 Header for 32-bit PE32+ Header for 64-bit This is a standard Windows PE header which indicates the type of the file, i.e. whether it is an EXE or a DLL. It also contains the timestamp of the file creation date and time. It also contains some other fields which might be needed for an unmanaged PE (Portable Executable), but not important for a managed one. For managed PE, the next header i.e. CLR header is more important
CLR Header	Contains the version of the CLR required, some flags, token of the entry point method (Main), size and location of the metadata, resources, strong name, etc.
Metadata	There can be many metadata tables. They can be categorized into 2 major categories. 1. Tables that describe the types and members defined in your code 2. Tables that describe the types and members referenced by your code Apart from that, there is another category of tables called Manifest tables, which can be found in the keeper PE file (An assemble could consist of more than one PE files)
IL Code	MSIL representation of the C# code. At runtime, the CLR converts it into native instructions

b. Classic COM binaries versus .NET assemblies

When we compile our code written in any .net language the associated compiler (like C#,VB Compiler) generates binaries called assembly which contains IL code .These Instructions are low level human readable language which can be converted into machine language by the run time compiler during its first execution. It is done just during execution so that the compiler will have before hand knowledge of which environment it’s going to run so that it can emit the optimized machine language code targeting that platform. The .net Framework’s such compiler is called Just-in-Time (JIT) compiler or *Jitter*. A .net assembly consist of following elements:

1. A Win32 file Header - The header data identifies the kind of application (console, Windows,

code lib) to be hosted by Windows operating system.

2. A CLR file Header- the CLR header is a block of data that provides information which allow it to be hosted by the CLR. CLR header contain information about the run time version used for building the assembly, public key etc.

3. CIL Code - CIL code are actual implementation of code in terms of instructions. CIL code is described in length below.

4. Type Metadata - Describes the types contained within the assembly and the format of types referenced by that assembly.

5. An Assembly Manifest-Describes the modules within the assembly, the version of the assembly, all the external assemblies' reference by that assembly.

c. Common Intermediate Language

CIL is a low level language based on the Common language Infrastructure specification document. (http://www.ecma-international.org/publications/standards/Ecma-335.htm). CIL was earlier referred as *MSIL*(Microsoft Intermediate Language) but later changed to CIL to standardize the name of the language which is based on the Common Language Infrastructure specification.

CIL is CPU and platform-independent instructions that can be executed in any environment supporting the Common Language Infrastructure, such as the .NET run time or the cross-platform Mono run time. These instructions are later processed by run time compiler and are then converted into native language. CIL is an object-oriented assembly language, and is entirely stack-based.

Before we go into the details of CIL, its stack nature etc, let's have a close look at the CIL code. The below code display the classic Hello World console application and the corresponding IL code generated by ILDASM (IL Disassemble can be found under Windows SDK Tools).