Scheme and Solution for C#.NET and Programming
I Internal Sept 2016

1. What are building blocks of .NET platform? Give the relationship between .NET runtime layer and class library? -10M

***Common language runtime* or *CLR*: 2M**

- 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*: 2M**

- 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.
-

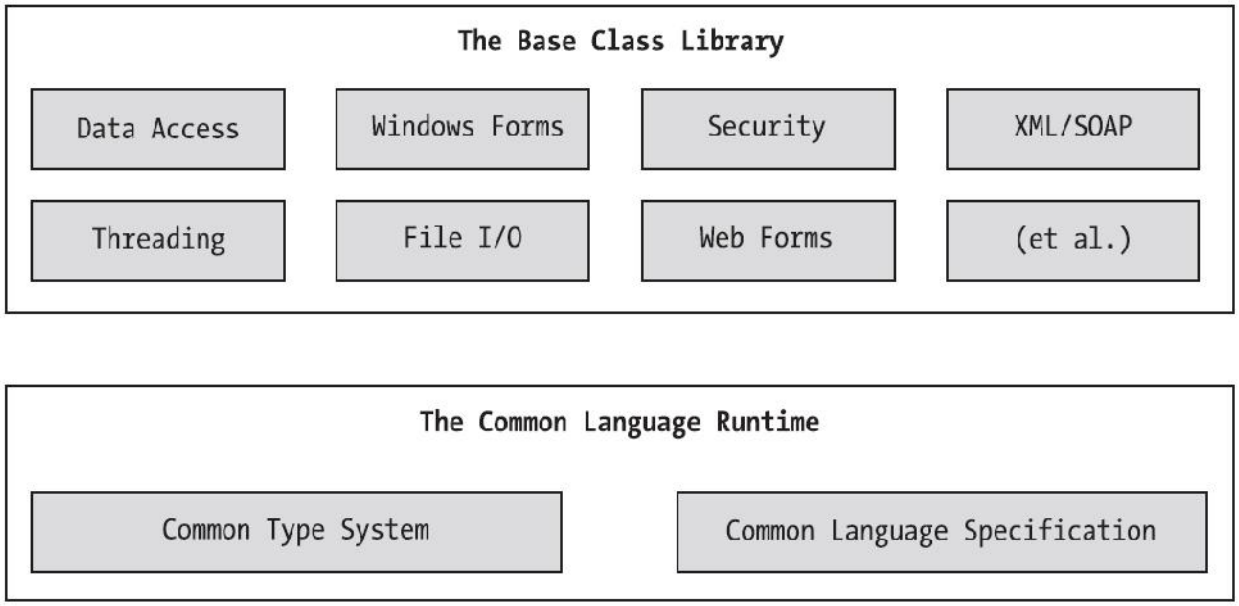
***Common Language Specification or CLS*: 1M**

- 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 -2M

- 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 relationship between the CLR, CTS, CLS, and the base class library, as shown in Figure.- **3M**



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

2. List the functions associated with System.Object class and override any two methods -10M

THE MASTER NODE: SYSTEM.OBJECT

- In .NET, every data type is derived from a common base class: *System.Object*.
- The Object class defines a common set of members supported by every type in the .NET framework.
- When we create a class, it is implicitly derived from System.Object.
- For example, the following declaration is common way to use.

```
class Test                                  class Test : System.Object
{                                           {
 ...            But, internally, it means that   ...
}                                           }
```

- System.Object defines a set of instance-level(non-static) and class-level(static) members.
- Some of the instance-level members are declared using the *virtual* keyword and can therefore be *overridden* by a derived-class:

```
// The structure of  System.Object class
namespace System
{
        public class Object
        {
                public Object();
                public virtual Boolean Equals(Object obj);
                public virtual Int32 GetHashCode();
                public Type GetType();
                public virtual String ToString();
                protected virtual void Finalize();
                protected Object MemberwiseClone();
                public static bool Equals(object objA, object objB);
                public static bool ReferenceEquals(object objA, object objB);
        }
}
```

-4M

Any two methods **3*2=6M**

OVERRIDING SOME DEFAULT BEHAVIORS OF SYSTEM.OBJECT

- In many of the programs, we may want to override some of the behaviors of System.Object.
- *Overriding* is the process of redefining the behavior of an inherited *virtual* member in a derived class.
- We have seen that System.Object class has some virtual methods like ToString(), Equals() etc. These can be overridden by the programmer.

OVERRIDING TOSTRING()

- Consider the following code:

```
using System;
using System.Text;
class Person
{
        public string Name, SSN;
        public byte age;

        public Person(string n, string s, byte a)
        {
                Name = n;
                SSN = s;
                age = a;
        }

        public Person(){ }

        // Overriding System.Object.ToString()
        public override string ToString()
        {
                StringBuilder sb = new StringBuilder();
                sb.AppendFormat("[Name={0}", this. Name);
                sb.AppendFormat(" SSN={0}", this.SSN);
                sb.AppendFormat(" Age={0}]", this.age);
                return sb.ToString();
        }

        public static void Main()
        {
                Person p1 = new Person("Ram", "11-12", 25);
                Console.WriteLine("p1 is {0}", p1.ToString());
        }
}
```

Output:

```
p1 is [Name=Ram SSN=11-12 Age=25]
```

- In the above example, we have overridden ToString() method to display the contents of the object in the form of tuple.
- The System.Text.StringBuilder is class which allows access to the buffer of character data and it is a more efficient alternative to C# string concatenation.

OVERRIDING EQUALS()

• By default, System.Object.Equals() returns true only if the two references being compared are referencing same object in memory.

• But in many situations, we are more interested if the two objects have the same content. Consider an example:

```
using System;
class Person
{
        public string Name, SSN;
        public byte age;

        public Person(string n, string s, byte a)
        {
                Name = n;
                SSN = s;
                age = a;
        }

        public Person(){ }

        public override bool Equals(object ob)
        {
           if (ob != null && ob is Person)
           {
              Person p = (Person)ob;

              if (p.Name == this.Name && p.SSN == this.SSN && p.age == this.age)
                  return true;
           }
          return false;
        }

        public static void Main()
        {
                Person p1 = new Person("Ram", "11-12", 25);
                Person p2 = new Person("John", "11-10", 20);
                Person p3 = new Person("Ram", "11-12", 25);
                Person p4 = p2;
                if(p1.Equals(p2))
                     Console.WriteLine("p1 and p2 are same");
                else
                     Console.WriteLine("p1 and p2 are not same");

                if(p1.Equals(p3))
                    Console.WriteLine("p1 and p3 are same");
                else
                    Console.WriteLine("p1 and p3 are not same");

                if(p2.Equals(p4))                    //compares based on content, not on reference
                    Console.WriteLine("p4 and p2 are same");
                else
                    Console.WriteLine("p4 and p2 are not same");
        }
}
```

3. a. Write a C# program to accept an inter as command line parameter from the user and check whether it is prime number or not-**5M**

```
/* C# Program to Check Whether the Given Number is a Prime number if so then
Display its Largest Factor */
using System;
namespace example
{
   class prime
   {
      public static void Main()
      {
         Console.Write("Enter a Number : ");-                    -1M
         int num;
         num = Convert.ToInt32(Console.ReadLine());   -1M
         int k;
         k = 0;
         for (int i = 1; i <= num; i++)                              -2M
         {
            if (num % i == 0)
            {
               k++;
            }
         }
         if (k == 2)                                                    -1M
         {
                        Console.WriteLine("Entered Number is a Prime Number and the
Largest Factor is {0}",num);
         }
         else
         {
            Console.WriteLine("Not a Prime Number");
         }
         Console.ReadLine();
      }
   }
}
```

3. b. Write a short note on enumeration and explain their usage with example-5M

Definition and explanation -**2M**

The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.

Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience. However, an enum can also be nested within a class or struct.

By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example, in the following enumeration, Sat is 0, Sun is 1, Mon is 2, and so forth.

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

Example -**3M**

```
public class EnumTest
{
    enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

    static void Main()
    {
        int x = (int)Days.Sun;
        int y = (int)Days.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
/* Output:
   Sun = 0
   Fri = 5
*
```

5. Explain four method parameter modifiers with example-10M

'in' parameter 2.5M

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’ parameter -2.5M

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’ parameter – 2.5M

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
```

‘params’ parameter -2.5M

- From the above example, we can observe that for *params* parameter, we can pass an array or individual elements.
- We can use *params* even when the parameters to be passed are of different types.
- Consider the following program:

```
using System;
class Test
{
        public static void MyFun(params object[] arr)
        {
                for(int i=0; i<arr.Length; i++)
                {
                        if(arr[i] is Int32)
                            Console.WriteLine("{0} is an integer", arr[i]);
                        else if(arr[i] is string)
                                Console.WriteLine("{0} is a string", arr[i]);
                             else if(arr[i] is bool)
                                        Console.WriteLine("{0} is a boolean",arr[i]);
                }
        }

        public static void Main()
        {
                int x=5;
                string s="hello";
                bool b=true;

                MyFun(b, x, s);
        }
}
```

Output:

```
True is a Boolean
5 is an integer
hello is a string
```

6. How to build C# application using cse.exe?-10M

-4M

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

-3M

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

-3M

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**

7. List six differences between value type and reference type
Write a program to illustrate value type containing reference type-10M

Each difference 0.5*6= 3M

VALUE TYPES	REFERENCE TYPES
Allocated on the stack	Allocated on the managed heap
Variables die when they fall out of the defining scope	Variables die when the managed heap is garbage collected
Variables are local copies	Variables are pointing to the memory occupied by the allocated instance
Variable are passed by value	Variables are passed by reference
Variables must directly derive from System.ValueType	Variables can derive from any other type as long as that type is not "sealed"
Value types are always sealed and cannot be extended	Reference type is not sealed, so it may function as a base to other types.
Value types are never placed onto the heap and therefore do not need to be finalized	Reference types finalized before garbage collection occurs

• Consider the following code:

```
class TheRefType
{
        public string x;
        public TheRefType(string s)
        {x=s;}
}

struct InnerRef
{
        public TheRefType refType;
        public int structData;
        public InnerRef(string s)
        {
                refType=new TheRefType(s);
                structData=9;
        }
}

class ValRef
{
        public static int Main(string[] args)
        {
        Console.WriteLine("making InnerRef type and setting structData to 666");
        InnerRef valWithRef=new InnerRef("initial value");
        valWithRef.structData=666;

        Console.WriteLine("assigning valWithRef2 to valWithRef");
        InnerRef valWithRef2;
        valWithRef2=valWithRef;

        Console.WriteLine("changing all values of valWithRef2");
        valWithRef2.refType.x="I AM NEW";
        valWithRef2.structData=777;

        Console.WriteLine("values after change");
        Console.WriteLine("valWithRef.refType.x is {0}", valWithRef.refType.x);
        Console.WriteLine("valWithRef2.refType.x is {0}", valWthRef2.refType.x);
        Console.WriteLine("valWithRef.structData is {0}", valWithRef.structData);
        Console.WriteLine("valWithRef2.structData is {0}", valWithRef2.structData);
        }
}
```

Output:

```
making InnerRef type and setting structData to 666
assigning valWithRef2 to valWithRef
changing all values of valWithRef2

values after change
valWithRef.refType.x is I AM NEW
valWithRef2.refType.x is I AM NEW
valWithRef.structData is 666
valWithRef2.structData is 777
```

• When a value-type contains other reference-types, assignment results in a copy of the references. In this way, we have 2 independent structures, each of which contains a reference pointing to the same object in memory i.e. **shallow copy**.

• When we want to perform a **deep copy** (where the state of internal references is fully copied into a new object), we need to implement the ICloneable interface.

Example 4M+Output 2M+ Explanation1M=-7M