CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 1 Answer Key – Nov. 2021

Sub:	Programming Using C# .Net						
Date:	11/11//2021	Duration:	90 min's	Max Marks:	50	Sem:	V

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

	PART I	MARKS	OBE CO	OBE RBT
1	**Explain the different components of .NET Framework 4.0** **Components of .NET Framework 4.0:** The .NET Framework provides all the necessary components to develop and run an application. The components of .NET Framework 4.0 architecture are as follows: • Common Language Runtime (CLR) • Common Type System (CTS) • Metadata and Assemblies • .NET Framework class library • Windows Forms • ASP.NET and ASP.NET AJAX • ADO.NET • Windows Workflow Foundation • Windows Presentation Foundatio • Windows Communication Found • Windows CardSpace • LINQ Let's now discuss about each of them in detail. **CLR[Common Language Runtime]:** **"CLR** *is an* ***Execution Engine for .NET Framework applications*****".** CLR is a heart of the.NET Framework. It provides a run-time environment to run the code and various services to develop the application easily. The services provided by CLR are – ▪ Memory Management ▪ Exception Handling ▪ Debugging ▪ Security • Thread execution • Code execution • Language Integration • Code s • Verific • Compi	[10]	CO1	L1

The following figure shows the **process** of compilation and execution of the code by the JIT Compiler:

i. After verifying, a **JIT** [*Just-In-Time*] compiler extracts the metadata from the file to translate that verified IL code into ***CPU-specific code*** or ***native code***. These type of IL Code is called as **managed code**.

ii. The source code which is directly compiles to the machine code and runs on the machine where it has been compiled such a code called as **unmanaged code**. It does not have any services of CLR.

iii. Automatic garbage collection, exception handling, and memory management are also the responsibility of the CLR.

Managed Code: Managed code is the code that is executed directly by the CLR. The application that are created using managed code automatically have CLR services, such as type checking, security, and automatic garbage collection.

The process of executing a piece of managed code is as follows:

- Selecting a language compiler
- Compiling the code to IL[This intermediate language is called managed code]
- Compiling IL to native code Executing the code

Unmanaged Code: Unmanaged Code directly compiles to the machine code and runs on the machine where it has been compiled. It does not have services, such as security or memory management, which are provided by the runtime. If your code is not security-prone, it can be directly interpreted by any user, which can prove harmful.

Automatic Memory Management: CLR calls various predefined functions of .NET framework to allocate and de-allocate memory of .NET objects. So that, developers need not to write code to explicitly allocate and de-allocate memory.

CTS [Common Type Specifications]:

The CTS defines the rules for declaring, using, and managing types at runtime. It is an integral part of the runtime for supporting cross-language communication.

The common type system performs the following functions:

- Enables cross-language integration, type safety, and high-performance code execution.
- Provides an object-oriented model for implementation of many programming languages.
- Defines rules that every language must follow which runs under .NET framework like C#, VB.NET, F# etc. can interact with each other.

	The CTS can be classified into two data types, are iv. Value Types v. Reference Type			
2	**What is an Assembly? Explain each component in the assembly** An **assembly** is a file that is automatically generated by the compiler upon successful compilation of every . NET application. It can be either a Dynamic Link Library or an executable file. It is generated only once for an application and upon each subsequent compilation the **assembly** gets updated Assemblies can stored in two types: **Static assemblies:** Static assemblies include interfaces, classes and resources. These assemblies are stored in PE (Portable executable) files on a disk. **Dynamic assemblies:** Dynamic assemblies run directly from the memory without being saved to disk before execution. However, after execution you can save the dynamic assemblies on the disk. **Global Assembly Cache:** The Global Assembly Cache (GAC) is ***a folder in Windows directory*** to store the .NET assemblies that are specifically designated to be shared by all applications executed on a system. ➢ The assemblies must be sharable by registering them in the GAC, only when needed; otherwise, they must be kept private. ➢ Each assembly is accessed globally without any conflict by identifying its name, version, architecture, culture and public key. You can deploy an assembly in GAC by using any one of the following: • An installer that is designed to work with the GAC • The GAC tool known as **Gacutil.exe** • The Windows Explorer to drag assemblies into the cache. **Strong Name Assembly:** A Strong Name contains the assembly's identity, that is, the information about the assembly's name, version number, architecture, culture and public key. • Using Microsoft Visual Studio .NET and other tools, you can provide a strong name to an assembly. • By providing strong names to the assembly, you can ensure that assembly is globally unique. **Private and Shared Assembly:** A single application uses an assembly, then it is called as **a private assembly.** Example: If you have created a DLL assembly containing information about your business logic, then the DLL can be used by your client application only. Therefore, to run the application, the DLL must be included in the same folder in which the	[10]	CO1	L1

client application has been installed. This makes the assembly private to your application.

Assemblies that are placed in the Global Assembly cache so that they can be used by multiple applications, then it is called as a **shared assembly**.

Example: Suppose the DLL needs to be reused in different applications. In this scenario, instead of downloading a copy of the DLL to each and every client application, the DLL can be placed in the global assembly cache by using the **Gacutil.exe** tool, from where the application can be accessed by any client application.

Side-by-Side Execution Assembly:
The process of executing **multiple versions** of an *application* or an *assembly* is known as **side-by-side execution**. Support for side-by-side storage and execution of different versions of the same assembly is an integral part of creating a strong name for an assembly.

- Strong naming of .NET assembly is used to provide unique assembly identity by using the sn.exe command utility.
- The strong-named assembly's version number is a part of its identity, the runtime can store multiple versions of the same assembly in the GAC.
- Load these assemblies at runtime.

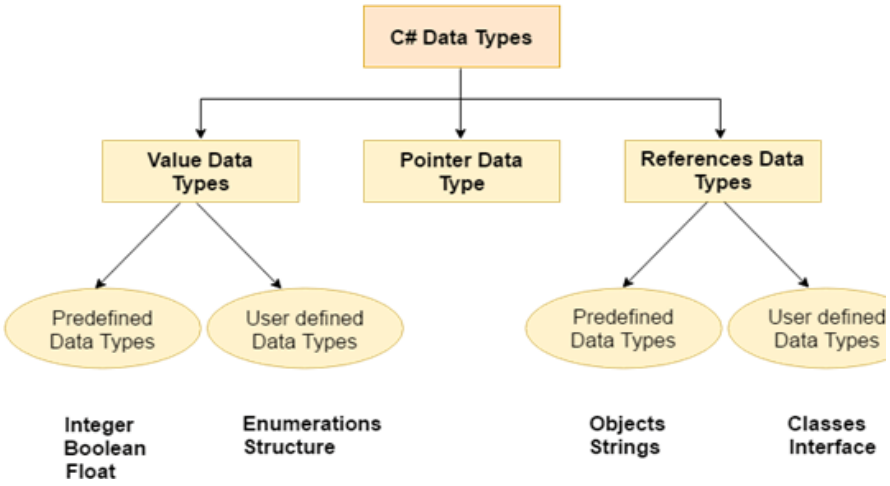
3	With example explain in detail about the following categories of C# data types. 1.Value Type 2.Reference Type 3.Pointer Type	[10]	CO1	L1

C# DataTypes

Types	Data Types
Value Data Type	short, int, char, float, doub
Reference Data Type	String, Class, Object and Int
Pointer Data Type	Pointers

Value Data Type

The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.



There are 2 types of value data type in C# language.

1) **Predefined Data Types** - such as Integer, Boolean, Float, etc.

2) **User defined Data Types** - such as Structure, Enumerations, etc.

The memory size of data types may change according to 32 or 64 bit operating system.

Let's see the value data types. It size is given according to 32 bit OS.

Data Types	Memory Size	Range
char	2 byte	-128 to 127

signed char	2 byte	-128 to 127
unsigned char	2 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	4 byte	-2,147,483,648 to -2,147,483,64
signed int	4 byte	-2,147,483,648 to -2,147,483,64
unsigned int	4 byte	0 to 4,294,967,295
long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long	8 byte	0 - 18,446,744,073,709,551,615
float	4 byte	1.5 * 10-45 - 3.4 * 1038, 7-dig
double	8 byte	5.0 * 10-324 - 1.7 * 10308, 15-precision
decimal	16 byte	at least -7.9 * 10?28 - 7.9 * 1 least 28-digit precision

Reference Data Type

The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables. In other words they refer to the memory location.

If the data is changed by one of the variables, the other variable automatically reflects this change in value.

There are 2 types of reference data type in C# language.

1) Predefined Types - such as Objects, String.

2) User defined Types - such as Classes, Interface.

Object Type: The Object Type is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
Object ob1;
ob1 = 100; // This is boxing
```

Dynamic Type : You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

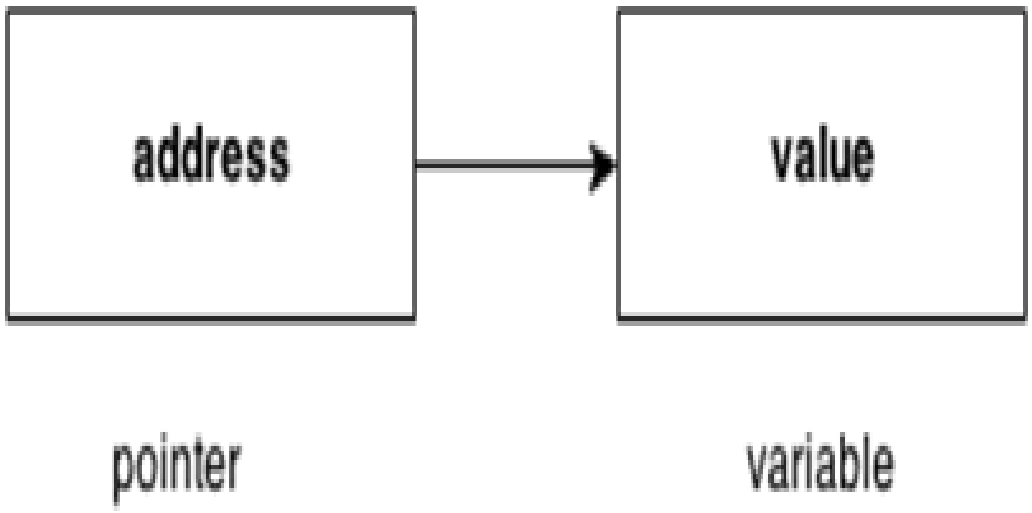
dynamic variablename = value; dynamic d

= 10;

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

Pointer Data Type

The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.



Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the a variable.
* (asterisk	Indirection	Access the val

	sign)	operator	address.			

Example:

int *a; char

*b;

4	**Define Type Casting and its forms.** Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms − • **Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes. • **Explicit type conversion** − These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.	[10]	CO1	L2

Sr.No.	Methods & Description
1	**ToBoolean** Converts a type to a Boolean value, where possible.
2	**ToByte** Converts a type to a byte.
3	**ToChar** Converts a type to a single Unicode character, where possible.
4	**ToDateTime** Converts a type (integer or string type) to date-time structures.
5	**ToDecimal** Converts a floating point or integer type to a decimal type.
6	**ToDouble**

		Converts a type to a double type.			
	7	**ToInt16** Converts a type to a 16-bit integer.			
	8	**ToInt32** Converts a type to a 32-bit integer.			
	9	**ToInt64** Converts a type to a 64-bit integer.			
	10	**ToSbyte** Converts a type to a signed byte type.			
	11	**ToSingle** Converts a type to a small floating point number.			
	12	**ToString** Converts a type to a string.			
	13	**ToType** Converts a type to a specified type.			
	14	**ToUInt16** Converts a type to an unsigned int type.			
	15	**ToUInt32** Converts a type to an unsigned long type.			
	16	**ToUInt64** Converts a type to an unsigned big integer.			
5	**Discuss the following with proper example:** **1. <u>Creating a class</u>** A class declaration in C# is composed of attributes, modifiers, the class name, base class and interfaces, and a body. Attributes, modifiers, and bases are all optional. The body of		[10]	CO1	L2

the class contains class members that can include constants, fields (or variables), methods, properties, indexers, events, operators, and nested types. Nested types are defined by class, interface, delegate, struct, or enum declarations within the class body.

Syntax:

```
class <Class_name> {
<access_modifier> [static] variable_type fields, constants
<access_modifier> [static] return_type methodName(args..){ - - - - }
... constructors, destructors ...
... properties ...// for component-based programming
... events ...
... indexers ... // for convenience
... overloaded operators ...
... nested types (classes, interfaces, structs, enums, delegates)
}
```

2. Creating an Object

In C#, objects help you to access the members of a class – fields, methods and properties by using the dot (□)
operator.
□ An object is a given instance of a particular class in memory.
In C#, the new keyword is the way to create an object.
Syntax:

```
<ClassName> <ObjectName> = new <ClassName>();
```

3. Using this keyword

The “this” keyword refers to the current instance of a class. With the “this” keyword, you can access an instance
of a class and use it with instance methods, instance constructors, and instance properties.
Syntax: Note:
□ To access the instance members of a class, use dot(□) operator
this.members;
□
Cannot be used with static members because static members are
accessed by a class and not by the instance of the class.

Example:

```
class Student{
string name, sid;
int marks;
public Student(string name, string sid, int marks){
this.name = name;
this.sid = sid;
this.marks = marks;
}
//If the user calls this ctor, forward to the 3-arg version.
//public Student() : this("Kalpana","MCA01",95) { }
public Student() {
name = "Kalpana";
```

```
sid = "MCA01";
marks = 95;
}
public void displayData(){
Console.WriteLine("Name: {0}\nID: {1}\nMarks:{2}", name,
sid, marks);
}
}
class thisdemo {
static void Main(string[] args){
Student st1 = new Student();
Console.WriteLine("Student Details:");
st1.displayData();
Student st2 = new Student("Tanuja", "MCA02", 85);
st2.displayData();
Console.ReadLine();
} } }
```

6	**Discuss the following with proper example:**
	
	1. <u>Creating an Array of objects</u>
	Creating an Array of Objects:
	To create an array of objects in C#, follow the below steps:
	□ Create an array
	□ Then create the individual elements of the array.
	Syntax:
	<ClassName>[] ObjectName = new <ClassName>[size];
	Example: Employee[] manager = new Employee[size];
	Note: you cannot create an array and its individual elements simultaneously.
	
	2. <u>Nested classes</u>
	A Nested class is a class that is defined inside another class. This class acts as the member of the parent class
	in which it is defined.
	Advantage: Accessing all the members of its outer class.
	
	Example:
	using System;
	namespace Class_Demos {
	class OuterClass {
	int i;
	public OuterClass() { i = 10; }
	public void OutDisplay() {
	Console.WriteLine("This is Outer Class");
	}
	public **class InnerClass** {
	public void InDisplay(OuterClass o){
	Console.WriteLine("This is Inner Class");
	Console.WriteLine("i value is:" +o.i);
	}
	}
		[10]	CO1	L2

```
}
class NestedClEx{
public static void Main(){
//Creating instance to outer class
OuterClass ob1 = new OuterClass();
ob1.OutDisplay();
//Creating instance to inner class
OuterClass.InnerClass iob = new OuterClass.InnerClass();
iob.InDisplay(ob1);
Console.Read();
} } }
```

3. Partial classes and methods

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.

Syntax:

```
public partial class student{
public void avgmarks() { }
}
public partial class student{
public void avgmarks() { }
}
```

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 partial method except for **static**
Partial methods are **private** by default.

Example:

```
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();
	}}}			

7	**Write a c# program to print factorial of a number**
	
	using System;
	public class FactorialExample
	{ **public static void** Main(**string**[] args)
	{
	int i,fact=1,number;
	Console.Write("Enter any Number:
	"); number=
	int.Parse(Console.ReadLine());
	for(i=1;i<=number;i++){
	fact=fact*i;
	}
	Console.Write("Factorial of " +number+" is: "+fact);
	}
	}
		[10]	CO5	L4

8.	**Explain about arrays and its types & Write a c# program for jagged array .**
	
	A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array of arrays."
	
	The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:
	
	int[][] jaggedArray = new int[3][];
	Before using jaggedArray, its elements must be initialized.
	jaggedArray[0] = new int[5];
	jaggedArray[1] = new int[4];
	jaggedArray[2] = new int[2];
	Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.
	
	Example:
	
	using System;
	using System.Collections.Generic;
	using System.Linq;
	using System.Text;
	
	namespace Program4
	{
	class Program
	{
	static void Main(string[] args)
	{
	int[][] arr = new int[3][];
	int arr0_length, arr1_length, arr2_length;
	int i,j, sum;
	
	Console.WriteLine ("Enter the length of the first array : ");
	arr0_length =Convert.ToInt32(Console.ReadLine());
	arr[0] = new int[arr0_length];
	Console.WriteLine ("Enter the Element of First array : ");
	for (i = 0; i < arr0_length; i++)
	arr[0][i] = Convert.ToInt32(Console.ReadLine());
	
	Console.WriteLine ("Enter the length of the second array : ");
	arr1_length =Convert.ToInt32(Console.ReadLine());
	arr[1] = new int[arr1_length];
	Console.WriteLine ("Enter the Element of Second array : ");
	for(i=0;i<arr1_length;i++)
	arr[1][i]=Convert.ToInt32(Console.ReadLine());
	
	Console.WriteLine ("Enter the length of the Third array : ");
	arr2_length =Convert.ToInt32(Console.ReadLine());
	arr[2] = new int[arr2_length];
		[10]	CO5	L3

```
            Console.WriteLine ("Enter the Element of Third array : ");
            for(i=0;i<arr2_length;i++)
               arr[2][i]=Convert.ToInt32(Console.ReadLine());

            Console.WriteLine("Array with its length  :");
            for (i = 0; i < 3; i++)
            {
               sum = 0;
               Console.WriteLine("Elements of Array " + i + "and its Sum ");
               Console.WriteLine();
               for (j = 0; j< arr[i].Length; j++)
               {
                  Console.Write(arr[i][j] + "   ");
                  sum = sum + arr[i][j];
               }
               Console.WriteLine(" Sum : "+sum);
               Console.WriteLine();
            }
            Console.ReadLine();
         }
      }
   }
```

9. **Define operator overloading. Write an example C# program for unary and binary operator overloading** [10] CO5 L4

All unary and binary operators have pre-defined implementations, that are automatically available in any expressions. In addition to this pre-defined implementations, user defined implementations can also be introduced in C#. The mechanism of giving a special meaning to a standard C# operator with respect to a user defined data type such as classes or structures is known as operator overloading.
Example:

```
class Prg3
{
   class Complex
   {
      private int x, y;
      public Complex() { }
      public Complex(int i, int j)
      {
         x = i;
         y = j;
      }
      public void ShowXY()
      {
         Console.WriteLine(Convert.ToString(x) + " " + Convert.ToString(y));
      }
      public static Complex operator -(Complex c)
      {
         Complex temp = new Complex();
```

```
        temp.x = -c.x;
        temp.y = -c.y;
        return temp;
    }
    public static Complex operator +(Complex c1, Complex c2)
    {
        Complex temp = new Complex();
        temp.x = c1.x + c2.x;
        temp.y = c1.y + c2.y;
        return temp;
    }
  }
  static void Main(string[] args)
  {
        Complex c1 = new Complex(10, 20);
        Complex c2 = new Complex(30, 40);
        Complex c3, c4 = new Complex();
        Console.Write("Complex Number1 :");
        c1.ShowXY();
        Console.Write ("Complex Number2 :");
        c2.ShowXY();
        c3 = -c1;
        Console.Write("Changing the sign of Complex Number1 :");
        c3.ShowXY();
        c4 = c1 + c2;
Console.Write("Addition of two complex Numbers 1 and 2:");
        c4.ShowXY();
        Console.ReadLine();
  }
 }
```

10	**Illustrate with an example about function overloading in C#** ***Method Overloading*** is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. C# can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class. › Overloaded methods are differentiated based on the number and type of the parameters passed as arguments to the methods. › You can not define more than one method with the same name, Order and the type of the arguments. It would be compiler error. › The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error. If both methods have the same parameter types, but different return type, then it is not possible. › <u>Example:</u>	[10]	CO5	L4

```
using System;
class GFG {

    // adding two integer values.
    public int Add(int a, int b)
    {
        int sum = a + b;
        return sum;
    }

    // adding three integer values.
    public int Add(int a, int b, int c)
    {
        int sum = a + b + c;
        return sum;
    }

    // Main Method
    public static void Main(String[] args)
    {

        // Creating Object
        GFG ob = new GFG();

        int sum1 = ob.Add(1, 2);
        Console.WriteLine("sum of the two "
                          + "integer value : " + sum1);

        int sum2 = ob.Add(1, 2, 3);
        Console.WriteLine("sum of the three "
                          + "integer value : " + sum2);
    }
}
```