CMR
INSTITUTE OF
TECHNOLOGY

**Internal Assessment Test 2 – November 2017(Answer Key)**

| Sub: | Programming using C#.NET | Code: | 13MCA53 |
|---|---|---|---|
| Sem:V | | Branch: | MCA |

Marks

1. a. **Describe the significance of 'is' and 'as' operator.** [5]

**is and as operator**

The `is` operator in C# is used to check the object type and it returns `bool` value: **true** if the object is the same type and **false** if not.

For `null` objects, it returns `false`.

```
namespace IsAndAsOperators
 {
   // Sample Student Class
        class Student
   {
 public int stuNo { get; set; }
public string Name { get; set; }
public int Age { get; set; }

  }
// Sample Employee Class
class Employee
{
public int EmpNo { get; set; }
public string Name { get; set; }
public int Age { get; set; }
public double Salary { get; set; }

}
class Program
{

 static void Main(string[] args)
{
        Student stuObj = new Student();
        stuObj.stuNo = 1;
        stuObj.Name = "Siva";
        stuObj.Age = 15;

        Employee EMPobj=new Employee();
        EMPobj.EmpNo=20;
        EMPobj.Name="Rajesh";
        EMPobj.Salary=100000;
        EMPobj.Age=25;
```

```
                // Is operator

        // Check Employee EMPobj is Student Type

        bool isStudent = (EMPobj is Student);
                System.Console.WriteLine("Empobj is a Student ?: {0}",
isStudent.ToString());

        // Check Student stiObj is Student Typoe
                isStudent = (stuObj is Student);
                System.Console.WriteLine("Stuobj is a Student ?: {0}",
isStudent.ToString());

        stuObj = null;
                // Check  null object Type
        isStudent = (stuObj is Student);
                System.Console.WriteLine("Stuobj(null) is a Student ?: {0}",
isStudent.ToString());
                System.Console.ReadLine();
    }
  }
```

The as operator does the same job of is operator but the difference is instead of bool, it returns the **object** if they are compatible to that type, else it returns null.

```
namespace IsAndAsOperators
{
  // Sample Student Class
  class Student
  {
    public int stuNo { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

  }
  // Sample Employee Class
  class Employee
  {
    public int EmpNo { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public double Salary { get; set; }

  }
  class Program
  {
    static void Main(string[] args)
    {
      Student stuObj = new Student();
      stuObj.stuNo = 1;
      stuObj.Name = "Praveen";
      stuObj.Age = 15;

      Employee EMPobj=new Employee();
      EMPobj.EmpNo=20;
      EMPobj.Name="Rajesh";
      EMPobj.Salary=100000;
```

```
            EMPobj.Age=25;

            System.Console.WriteLine("Empobj is a Student ?: {0}",
CheckAndConvertobject(EMPobj));

            System.Console.WriteLine("StuObj is a Student ?: {0}",
CheckAndConvertobject(stuObj));
            System.Console.ReadLine();

        }

        public static string CheckAndConvertobject(dynamic obj)
        {
          // If obj is Type student it asign value to Stuobj else it asign null
            Student stuobj = obj as Student;
            if (stuobj != null)
                return "This is a Student and his name is " + stuobj.Name;

                return "Not a Student";


        }
    }
}
```

b. **Demonstrate null coalescing operator using Console Application** [5]

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication23
{
    class Program
    {
        static void Main(string[] args)
        {
            int? i = null;
            int? j= 30;
            int k;
            k = i ?? 25;
            Console.WriteLine(" I is null" + k);
            k = j ?? 25;
            Console.WriteLine(" J is not null" + k);
            Console.ReadKey();

        }
    }
}
```

2. a. **Explain with an example the following Class.** [5]

**a) Partial Class**

It is possible to split the definition of a class or a struct, an interface or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

namespace ConsoleApplication25
{
    class Program
    {
        static void Main(string[] args)
        {
            demo d=new demo();
            d.get();
            d.compute();
            Console.ReadKey();
         }
    }
    partial class demo
    {
        double p, n, r, si;
        public void get()
        {
            Console.WriteLine("Enter Principal:");
            p = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter number of Years:");
            n = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter ROI:");
            r = Convert.ToDouble(Console.ReadLine());
       }
        partial void show();

    }

        partial class demo
        {
            partial void show()
            {
                Console.WriteLine("Principal is:"+p);
                Console.WriteLine("Total is:"+(si+p));
            }
            public void compute()
            {
             si = (p * n * r) / 100;
            Console.WriteLine("Simple Interest:" + si);
            show();
            }

        }
    }
```

**b) Sealed Class**

Sealed class is used to define the inheritance level of a class.

The sealed modifier is used to prevent derivation from a class. An error occurs if a sealed class is specified as the base class of another class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication32
{
    sealed class sample
    {
```

```
        public void display()
        {
            Console.WriteLine("This is sealed class, we cannot inherit the
            properies of sealed class");
        }

    }
    //class sample1 : sample
    //{
    //    public void get()
    //    {
    //        Console.Write("Trying to inherit seaqled class");
    //    }

    //}
    class Program
    {
        static void Main(string[] args)
        {
            sample s=new sample();
            s.display();
            Console.ReadKey();
        }
    }
}
```

b. **Mention the difference between abstract class and Interface with syntax.** [5]

| **Abstract class** | **Interface** |
|---|---|
| A class can extend only one abstract class | A class can implement multiple interfaces. |
| A derived class that uses abstract methods of an abstract class must override all abstract methods. | A class that implements an interface must override all the methods defined by the interface. |
| Any class can extend an abstract class | Only an interface can extend another interface. |
| Syntax:<br>abstract class baseclass<br>{ }<br>Class derivedclass: baseclass<br>{ }<br>Class Program<br>{<br>void Main(String[] args)<br>{<br>dervivedclass d;<br>}<br>} | Syntax:<br>interface <interface name><br>{<br>// abstract method declaration in interface body<br><br>} |

3. a. **Demonstrate how you would enforce encapsulation using accessor and mutators.** [5]

Encapsulation provides a way to protect data from accidental corruption. Rather than defining the data in the form of public, we can declare those fields as private. The Private data are manipulated indirectly by two ways. Let us see some example programs in C# to demonstrate Encapsulation by those two methods. The first

method is using a pair of conventional accessor and mutator methods. Another one method is using a named property. Whatever be the method our aim is to use the data with out any damage or change.

```
// Program to demonstrate encapsulation using accessor and mutator

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication30
{
    class gomesencap
    {
        private string name;
        //accessor
        public string get()
        {
            return name;
        }
        //mutator
        public void set(string n)
        {
            name = n;

        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            gomesencap g = new gomesencap();
            g.set("Hello");
            Console.WriteLine("String passed is:" + g.get());
            Console.ReadKey();
        }
     }
}
```

b. **Difference between Static and Non-Static members.** [5]

| **Parameter for Difference** | **Static Members** | **Non-Static Members** |
|---|---|---|
| Belongs | Belongs to a class. This means a static members can be called by a class and by the object of the class | Belongs to object of a class. This means that non-static members can be called only by the objects of the class. |
| Accessibility | Accesses only other static members of a class. | Access both static and non-static members of a class. |
| Instantiation | Creates only one copy of variable and its value remains the same even when the class is instantiated. | Creates a copy of the variables for their objects, every time the class is instantiated. |

4. a. **How delegates are used in C#? Discuss single cast and multicast delegates.** [10]

**Delegate**: A delegate is a special type of object that contains the details of a method rather than data.
In C# delegate is a class type object, which is used to invoke the method that has

been encapsulated into it at the time of its creation. A delegate can be used to hold the reference to a method of any class.
Delegate contains 3 important piece of information

1. The name of the method on which it makes calls
2. Argument of this method
3. Return value of this method

Creating and using delegate:

1. Declaring a delegate
2. Defining delegate methods
3. Creating delegate objects
4. Invoking delegate objects

**Declaring a delegate**
Access-modifier delegate return-type delegate-name (parameter-list);
Public delegate void compute(int x, int y);
**Defining Delegate Methods**
Public static void Add(int a, int b)
{
Console.WriteLine("Sum={0}",a +b);
}

**Creating Delegate Objects:**
Delegate-name object-name=new delegate-name(expression);

**Invoking Delegate object**
Delegate-object(argument-list)
Cmp1(30,20);

```
using System;
delegate void CustomDel(string s);
class TestClass
{
    static void Hello(string s)
    {
        System.Console.WriteLine("  Hello, {0}!", s);
    }
    static void Goodbye(string s)
    {
        System.Console.WriteLine("  Goodbye, {0}!", s);
    }
    static void Main()
    {
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;
        hiDel = Hello;
        byeDel = Goodbye;
        multiDel = hiDel + byeDel;
        multiMinusHiDel = multiDel - hiDel;
        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
        Console.ReadLine();
    }
}
```

**Multicasting with delegates:**
A delegate object can hold reference of and invoke multiple methods.

```
using System;
```

```
delegate void CustomDel(string s);
class TestClass
{
    static void Hello(string s)
    {
        System.Console.WriteLine("  Hello, {0}!", s);
    }
    static void Goodbye(string s)
    {
        System.Console.WriteLine("  Goodbye, {0}!", s);
    }
    static void Main()
    {
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;
        hiDel = Hello;
        byeDel = Goodbye;
        multiDel = hiDel + byeDel;
        multiMinusHiDel = multiDel - hiDel;
        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
        Console.ReadLine();
    }
}
```

5. a. **Explain the following terms: Events, Event Source, Event Handler and demonstrate concept of Events using Console Application.** [10]

Events : An event is a message sent by an object to signal the occurrence of an action. The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic.

Event Source: The event source is the name of the software that logs the event. It is often the name of the application or the name of a subcomponent of the application if the application is large.

EventHandler: We perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button. To receive notifications when the event occurs, our event handler method must subscribe to the event.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Delevents
{
    public delegate int demo_del(int a,int b);
    class Program
    {
        event demo_del devent;
        public Program()
        {
            this.devent = new demo_del(this.msg);
        }
        public int msg(int x, int y)
        {
            int z = x + y;
            return(z);
        }
        static void Main(string[] args)
```

```
        {
            Program p = new Program();
            int c= p.devent(10, 20);
            Console.WriteLine("Output is:" + c);
            Console.ReadKey();
        }
    }
}
```

6 a. **Define properties in C#? properties in C#? Explain different types of properties and show the implementation with type-cast get () and set () method.** [10]

In C#, a property is a member that provides flexible mechanism to read, write and compute value of a private field.

Read-Only property: A read only property has a get accessor but not a set accessor.
Static Property: It can be used to access only static members.
Anonymous type: Encapsulate read-only property of an object into a single object without having to first explicitly define a type.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace properties
{
   public class Student
   {
      private int usn;
      private string percentile;
      private string fullName;

      public int ID
      {
         get { return usn; }
         set
         {
            usn = value;
         }
      }
            public string Name
      {
         get { return fullName; }
         set { fullName = value; }
      }
      public string marks
      {
         get { return percentile; }
         set { percentile = value; }
      }}
     class Properties
   {
      static void Main(string[] args)
      {
         Student e = new Student();
         e.ID = 81;
         e.Name = "SCIENTIST THE RESEARCHIST";
         e.marks = "58%";
         Console.WriteLine("ID= {0},NAME={1},marks percentile={2}", e.ID,
e.Name, e.marks);
         Console.ReadLine();
       }
   }
}
```

7 a. **List the advantages of polymorphism. Explain Compile Time and Run Time Polymorphism** [10]

Polymorphism: is used to exhibit different forms of any particular procedure.

Advantages:

Allows us to invoke methods of a derived class through base class reference during runtime.

Provides different implementation of methods in a class that are called through the same name

Polymorphism can be static(Compile) or dynamic(run time). In static polymorphism, the response to a function is determined at the compile time. In dynamic polymorphism, it is decided at run-time.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

- Function overloading
- Operator overloading

We discuss operator overloading in next chapter.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function print() to print different data types −

```
using System;

namespace PolymorphismApplication {

  class Printdata {

    void print(int i) {
      Console.WriteLine("Printing int: {0}", i );
    }

    void print(double f) {
      Console.WriteLine("Printing float: {0}" , f);
    }

    void print(string s) {
      Console.WriteLine("Printing string: {0}", s);
    }
```

```
    static void Main(string[] args) {
      Printdata p = new Printdata();

      // Call print to print integer
      p.print(5);

      // Call print to print float
      p.print(500.263);

      // Call print to print string
      p.print("Hello C++");
      Console.ReadKey();
    }
  }
}
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. Abstract classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes −

You cannot create an instance of an abstract class

You cannot declare an abstract method outside an abstract class

When a class is declared sealed, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class −

```
using System;

namespace PolymorphismApplication {

  abstract class Shape {
    public abstract int area();
  }

  class Rectangle:  Shape {
    private int length;
    private int width;

    public Rectangle( int a = 0, int b = 0) {
      length = a;
      width = b;
    }

    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * length);
    }
  }

  class RectangleTester {
        static void Main(string[] args) {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
```

```
    }
  }
}
```

8 a. **Explain Classes and Objects in C#** [4]

Classes allows us to control all functions that can be applied to a given set of data as well as how to access the data.

The basic syntax of class:

```
<access specifier> class  class_name {
   // member variables
   <access specifier> <data type> variable1;
   <access specifier> <data type> variable2;
   ...
   <access specifier> <data type> variableN;
   // member methods
   <access specifier> <return type> method1(parameter_list) {
      // method body
   }
```

C# objects help us to access the members of a class by using the dot operator.

The basic syntax:

<classname> <object name> = new <classname>();

b. **Demonstrate the concept of Creating an Array of Objects using C# program.** [6]

```
// Array of Objects

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication28
{
    class Program
    {
        static void Main(string[] args)
        {
            Employee[] manager = new Employee[10];
            for (int k =1; k <=3; k++)
            {
                manager[k] = new Employee();
             }
            for (int i = 1; i <= 3; i++)
            {
               Console.WriteLine("Enter " + i + " manager data");
                manager[i].getdata();

            }

            for (int j = 1; j <= 3; j++)
            {
                Console.WriteLine("Data of manager"+ j +"is");
                manager[j].putdata();
            }
            Console.ReadKey();
        }
    }
```

```
    class Employee
    {
        string name;
        int age;
        public void getdata()
        {
            Console.WriteLine("Enter name");
            name = Console.ReadLine();
            Console.WriteLine("Enter age");
            age = Convert.ToInt32(Console.ReadLine());
        }
        public void putdata()
        {
            Console.WriteLine("Name"+name);

            Console.WriteLine("age"+age.ToString());

        }
    }
}
```