Module 3 Practicals

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using ClassLibrary1;
using CA2=ConsoleApp2;
using System.Collections;
using System.Reflection;

//delegate declaration-Delegate declaration determines the methods
//that can be referenced by the delegate. A delegate can refer to a method,
//which has the same signature as that of the delegate.
delegate int NumberChanger(int n);

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            /* ----------------Part-1-------------------*/
            //usage of classes
            Box Box1 = new Box(5.0,6.0,7.0);
            double volume = 0.0;


            volume = Box1.getVolume();//volume
            Console.WriteLine("Volume of Box1 : {0}", volume);

            Box1.count();
            Box1.count();
            Box1.count();

            Console.WriteLine("Variable num: {0}", Box.getNum());

            //static class usage
            Author.details();

            Console.WriteLine("Author name : {0} ", Author.A_name);
            Console.WriteLine("Book Name : {0} ", Author.B_name);
            Console.WriteLine("ID : {0} ",    Author.id);

            //sealed class usage
            SealedClass slc = new SealedClass();
            int total = slc.Add(6, 4);
```

```
            Console.WriteLine("Total = " + total.ToString());

            //sealed method usage
            Printer p = new Printer();
            p.show();
            p.print();

            Printer ls = new LaserJet();
            ls.show();
            ls.print();

            Printer of = new Officejet();
            of.show();
            of.print();

            //abstract class usage
            Square s = new Square(6);
            Console.WriteLine("Area    = " + s.Area());

            Console.ReadKey();

        /* ----------------Part-2--------------------*/
                        int x = 5;
                        int y = 10;

            //class library[.net library] usage
            Class1 o = new Class1();              //class 1 belongs to class library
    1
                        int res = o.add(x, y);
                        Console.WriteLine(res);

                        //namespace usage
            CA2.Program.display();
        //namespace in another solution/project-added reference,using
    alias=<namspace name>,use the namespace

                        Console.Read();

    /* ----------------Part-3--------------------*/
            //public access modifier-Access is granted to the entire
program(another assembly/method)
            Student S = new Student(1, "Heli");
            Console.WriteLine("Roll number: {0}", S.rollNo);//directly accessing
members
            Console.WriteLine("Name: {0}", S.name);
            Console.WriteLine("Roll number: {0}", S.getRollNo());//via member
method
            Console.WriteLine("Name: {0}", S.getName());
```

```
            //protected-Access is limited to the class that contains the member and
derived types of this class
            X obj1 = new X();
            Y obj2 = new Y();
            Console.WriteLine("Value of x is : {0}", obj2.getX());

            //internal-Access is limited to only the current Assembly
            Complex c = new Complex();//can't be accessed in another namespace
            c.setData(2, 1);
            c.displayData();

            //protected internal-Access is limited to the current assembly
            //(if class accessing member is not derived class then that class
            //must be in same assembly as of containing class) or
            //the derived types of the containing class(if accessing class is derived
            //then it can be within/outside assembly as of containing class)

            //private-Access is only granted to the containing class
            Parent obj = new Parent();
            //no meaning to create obj of child as showValue will give error
            // obj.value = 5;
            // Also gives an error
            // Use public functions to assign
            // and use value of the member 'value'
            obj.setValue(4);
            Console.WriteLine("Value = " + obj.getValue());

            //private protected-Access is granted to the containing class
            //or its derived types present in the current assembly
            //(derived class in another assembly can't access)
            Child obj3 = new Child();
            // obj3.value1 = 5;
            // Also gives an error
            // Use public functions to assign
            // and use value of the member 'value1'
            obj3.setValue1(4);
            Console.WriteLine("Value1 = " + obj3.getValue1());
            obj3.showValue1();

            Console.ReadKey();

        /* ----------------Part-4------------------    */

                    //arraylist-an ordered collection of an object that can be
    indexed individually
                    //unlike array you can add and remove items from a list at a
    specified position
                    //using an index and the array resizes itself automatically.
                    //It also allows dynamic memory allocation, adding, searching
    and sorting items in the list.
```

```
ArrayList al = new ArrayList();

al.Add(45);
al.Add(78);
al.Add(33);
al.Add(56);
al.Add(12);
al.Add(23);
al.Add(9);

Console.WriteLine("Capacity: {0} ", al.Capacity);
Console.WriteLine("Count: {0}", al.Count);

Console.Write("Content: ");
foreach (int i in al)
{
    Console.Write(i + " ");
}
Console.WriteLine();

Console.Write("Sorted Content: ");
al.Sort();
foreach (int i in al)
{
    Console.Write(i + " ");
}
Console.WriteLine();

//hashtable-collection of key-and-value pairs
//that are organized based on the hash code of the key.
//It uses the key to access the elements in the collection.
Hashtable ht = new Hashtable();

ht.Add("001", "Heli");
ht.Add("002", "Deepika");
ht.Add("003", "Ranveer");
ht.Add("004", "Shahrukh");
ht.Add("005", "Salman");
ht.Add("006", "Alia");
ht.Add("007", "Neha");

if (ht.ContainsValue("Durga"))
{
    Console.WriteLine("This student name is already in the
list");
}
else
{
    ht.Add("008", "Durga");
}
```

```
                    ICollection key = ht.Keys;

                    foreach (string k in key)
                    {
                        Console.WriteLine(k + ": " + ht[k]);
                    }

                    //sortedlist-collection of key-and-value pairs that are sorted by
the keys
                    //and are accessible by key and by index.
                    //A sorted list is a combination of an array and a hash table.
                    //It contains a list of items that can be accessed using a key or
an index.
                    //If you access items using an index, it is an ArrayList, and if
you access items using a key,
                    //it is a Hashtable. The collection of items is always sorted by
the key value.
                    SortedList sl = new SortedList();

                    sl.Add("001", "Heli");
                    sl.Add("002", "Deepu");
                    sl.Add("003", "Ranveer");
                    sl.Add("004", "Nehu");
                    sl.Add("005", "Rohanpreet");
                    sl.Add("006", "Alia");
                    sl.Add("007", "Ritesh");

                    if (sl.ContainsValue("Durga"))
                    {
                        Console.WriteLine("This student name is already in the
list");
                    }
                    else
                    {
                        sl.Add("008", "Durga");
                    }

                    ICollection key_sl = sl.Keys;

                    foreach (string k in key_sl)
                    {
                        Console.WriteLine(k + ": " + sl[k]);
                    }

                    //stack-a last-in, first out collection of object.
                    //It is used when you need a last-in, first-out access of items.
When you add an item in the list,
                    //it is called pushing the item and when you remove it, it is
called popping the item.
```

```
                    Stack st = new Stack();

                    st.Push('A');
                    st.Push('M');
                    st.Push('G');
                    st.Push('W');

                    Console.WriteLine("Current stack: ");
                    foreach (char c in st)
                    {
                        Console.Write(c + " ");
                    }
                    Console.WriteLine();

                    st.Push('V');
                    st.Push('H');
                    Console.WriteLine("Current stack: ");
                    foreach (char c in st)
                    {
                        Console.Write(c + " ");
                    }
                    Console.WriteLine();
                    Console.WriteLine("The next poppable value in stack: {0}",
st.Peek());

                    Console.WriteLine("Removing values ");
                    st.Pop();
                    st.Pop();
                    st.Pop();

                    Console.WriteLine("Current stack: ");
                    foreach (char c in st)
                    {
                        Console.Write(c + " ");
                    }
                    Console.WriteLine();

                    //queue-a first-in, first out collection of object. It is used when
                    //you need a first-in, first-out access of items.
                    //When you add an item in the list, it is called enqueue,
                    //and when you remove an item, it is called deque.
                    Queue q = new Queue();

                    q.Enqueue('A');
                    q.Enqueue('M');
                    q.Enqueue('G');
                    q.Enqueue('W');

                    Console.WriteLine("Current queue: ");
                    foreach (char c in q) Console.Write(c + " ");
```

```
            Console.WriteLine();

            q.Enqueue('V');
            q.Enqueue('H');
            Console.WriteLine("Current queue: ");
            foreach (char c in q) Console.Write(c + " ");
            Console.WriteLine();

            Console.WriteLine("Removing some values ");
            char ch = (char)q.Dequeue();
            Console.WriteLine("The removed value: {0}", ch);
            ch = (char)q.Dequeue();
            Console.WriteLine("The removed value: {0}", ch);

            Console.ReadKey();

    /* ----------------Part-5-------------------*/

                //reflection-Reflection objects are used for obtaining type
information at runtime. The classes
                //that give access to the metadata of a running program
are in the System.Reflection namespace.
                //It allows view attribute information at runtime.
                //It allows examining various types in an assembly and
instantiate these types.
                //It allows late binding to methods and properties.
                //It allows creating new types at runtime and then
performs some tasks using those types.

                //load assembly file which we want to dig
                var assembly =
Assembly.LoadFile(@"C:\Users\SMART\Documents\Visual Studio
2019\Projects\ConsoleApp1\bin\Debug\ConsoleApp1.exe");

                //if we want to dig current file:
                //var assembly = Assembly.GetExecutingAssembly();

                Console.WriteLine(assembly.FullName);

                //to get all the things like class,interfaces etc ,if
exist ,from assembly
                var type = assembly.GetTypes();
                foreach (var t in type)
                {
                    Console.WriteLine("Base type:" + t.BaseType);
                    Console.WriteLine("Name:" + t.Name);
                    Console.WriteLine("Full Name:" + t.FullName);
                    Console.WriteLine("Namespace:" + t.Namespace);

                    //particular class t digging
```

```
                    var field = t.GetFields();
                    foreach (var f in field)
                    {
                        //only accessible if fields are public
                        Console.WriteLine("Field Name:" + f.Name);
                        Console.WriteLine("Whether Private:" +
f.IsPrivate);
                    }

                    var method = t.GetMethods();
                    foreach (var m in method)
                    {
                        //only accessible if fields are not private
                        Console.WriteLine("Method Name:" +
m.Name);
                        Console.WriteLine("Whether Private:" +
m.IsPrivate);

                        var parameter = m.GetParameters();
                        foreach (var p in parameter)
                        {
                            Console.WriteLine("parameter name:" +
p.Name);
                            Console.WriteLine("parameter type:" +
p.ParameterType);
                        }
                    }

                    var cons = t.GetConstructors();
                    foreach (var c in cons)
                    {
                        Console.WriteLine("Constructor name:" +
c.Name);
                    }

                    var prop = t.GetProperties();
                    foreach (var p in prop)
                    {
                        Console.WriteLine("Property name:" +
p.Name);
                        Console.WriteLine("Property type:" +
p.PropertyType);
                    }

                    //leaving line between components
                    Console.WriteLine();
                }

                //properties-Properties are named members of classes,
structures, and interfaces.
```

```
            //Member variables or methods in a class or structures are
called Fields. Properties
            //are an extension of fields and are accessed using the
same syntax. They use accessors
            //through which the values of the private fields can be
read, written or manipulated.
            //Properties do not name the storage locations. Instead,
they have accessors that read,
            //write, or compute their values.
            //Abstract properties can also be there where abstract
class contains abstract properties and
            //derived class will have variables for which properties
would be overriden.
            Student s = new Student();

            // Setting code, name and the age of the student
            s.Code = "C1";
            s.Name = "Heli";
            s.Age = 21;
            Console.WriteLine("Student Info:\n{0}", s);

            //let us increase age
            s.Age += 1;
            Console.WriteLine("Student Info:\n{0}", s);

            //indexer-allows an object to be indexed such as an array.
            //When you define an indexer for a class, this class
behaves similar to a virtual array.
            //You can then access the instance of this class using the
array access operator ([ ]).
            //Declaration of behavior of an indexer is to some extent
similar to a property
            //, you use get and set accessors for defining an
indexer.However, properties return or
            //set a specific data member, whereas indexers returns or
sets a particular value from the object instance.
            //In other words,it breaks the instance data into smaller
parts and indexes each part, gets or sets each part.
            //Defining a property involves providing a property name.
Indexers are not defined with names, but with
            //the this keyword, which refers to the object instance.
            A o1 = new A();
            o1[0] = "heli";
            Console.WriteLine(o1[0]);
            o1[1] = 1;
            Console.WriteLine(o1[1]);

            B o = new B();
            o[0] = "hi";
            o[1] = "heli";
```

```
            Console.WriteLine(o[0]);
            Console.WriteLine(o[1]);

            //delegates-C# delegates are similar to pointers to
functions, in C or C++.
            //A delegate is a reference type variable that holds the
reference to a method. The reference
            //can be changed at runtime.Delegates are especially used
for implementing events and the call - back
            //methods.All delegates are implicitly derived from the
System.Delegate class.
            //create delegate instances-Once a delegate type is
declared, a delegate object must be created with
            //the new keyword and be associated with a particular
method. When creating a delegate, the argument
            //passed to the new expression is written similar to a
method call, but without the arguments to the method.
            NumberChanger nc1 = new
NumberChanger(Delegate.AddNum);
            NumberChanger nc2 = new
NumberChanger(Delegate.MultNum);

            //calling the methods using the delegate objects
            nc1(25);
            Console.WriteLine("Value of Num: {0}",
Delegate.getNum());
            nc2(5);
            Console.WriteLine("Value of Num: {0}",
Delegate.getNum());

            //multicasting of delegates-Delegate objects can be
composed using the "+" operator.
            //A composed delegate calls the two delegates it was
composed from. Only delegates of the same
            //type can be composed. The "-" operator can be used to
remove a component delegate from a composed delegate.
            //Using this property of delegates you can create an
invocation list of methods that will be called
            //when a delegate is invoked.This is called multicasting of
a delegate.
            NumberChanger nc;

            nc = nc1;
            nc += nc2;

            //calling multicast
            nc(5);
            Console.WriteLine("Value of Num: {0}",
Delegate.getNum());
```

```
                    //events-Events are user actions such as key press, clicks,
    mouse movements, etc., or some occurrence such as
                    //system generated notifications. Applications need to
    respond to events when they occur. For example,
                    //interrupts. Events are used for inter-process
    communication.The events are declared and raised in a
                    //class and associated with the event handlers using
    delegates within the same class or some other class.
                    //The class containing the event is used to publish the
    event. This is called the publisher class. Some other
                    //class that accepts this event is called the subscriber class.
    Events use the publisher-subscriber model.
                    //A publisher is an object that contains the definition of
    the event and the delegate. The event-delegate
                    //association is also defined in this object. A publisher
    class object invokes the event and it is notified
                    //to other objects.A subscriber is an object that accepts the
    event and provides an event handler.
                    //The delegate in the publisher class invokes the
    method(event handler) of the subscriber class.
                    Event.display();//events can't be accessed outside the loc
    at which it is declared

                    //anonymous methods-provide a technique to pass a code
    block as a delegate parameter.
                    //Anonymous methods are the methods without a name,
    just the body.
                    //You need not specify the return type in an anonymous
    method; it is inferred from the
                    //return statement inside the method body.Anonymous
    methods are declared with the creation
                    //of the delegate instance, with a delegate keyword.The
    delegate could be called both with anonymous
                    //methods as well as named methods in the same way, i.e.,
    by passing the method parameters to the delegate
                    //object.

                    //create delegate instances using anonymous method
                    nc = delegate (int x) {
                          return x;//body of anonymous method
                    };
                    //calling the delegate using the anonymous method
                    Console.WriteLine(nc(10));

                    Console.ReadKey();
      }
}

//class usage
```

```
class Box
{
    //encapsulation
    private double length;      // Length of a box
    private double breadth;     // Breadth of a box
    private double height;      // Height of a box

    public Box(double len,double bre,double hei)//paramaterised
constructor
    {
        length = len;
        breadth = bre;
        height = hei;
    }
    ~Box()//destructor
    {

    }
    public double getVolume()//method declaration
    {
        return length * breadth * height;
    }

    public static int num;      //static variable

    public void count() //accessing static variable in non-static method
    {
        num++;
    }
    public static int getNum() //static method
    {
        return num;
    }
}

//static class-can only have static members
static class Author
{
    // Static data members
    public static string A_name = "Heli";
    public static string B_name = "C#";
    public static int id = 14;

    //static constructor
    static Author()
    {
        Console.WriteLine("Static constructor");
    }

    // Static method
```

```
        public static void details()
        {
            Console.WriteLine("The details of Author is:");
        }
    }

    //class to show sealed class
    sealed class SealedClass
    {

        public int Add(int a, int b)
        {
            return a + b;
        }
    }

    //classes for sealed method
    class Printer
    {
        public virtual void show()//virtual methods
        {
            Console.WriteLine("Display dimension : 6*6");
        }

        public virtual void print()
        {
            Console.WriteLine("Printer printing....\n");
        }
    }

    class LaserJet : Printer
    {

        sealed override public void show()//sealed can only be used with
override for methods
        {
            Console.WriteLine("Display dimension : 12*12");
        }

        override public void print()
        {
            Console.WriteLine("Laserjet printer printing....\n");
        }
    }

    class Officejet : LaserJet
    {

        /*can not override show function or else
        compiler error : 'Officejet.show()' :
```

```
            cannot override inherited member
            'LaserJet.show()' because it is sealed.
            */

                override public void print()
                {
                    Console.WriteLine("Officejet printer printing....");
                }
        }

            // classes for abstraction
            abstract class Shape
            {
                abstract public int Area();//abstract method
            }
            class Square : Shape//inheriting Shape class
            {
                int side = 0;

                public Square(int n)
                {
                    side = n;
                }

                public override int Area()//overriding abstract method
                {
                    return side * side;
                }
            }

    //class for public modifier
        class Student
        {
            public int rollNo;
            public string name;

            public Student(int r, string n)
            {
                rollNo = r;
                name = n;
            }

            public int getRollNo()
            {
                return rollNo;
            }
            public string getName()
            {
                return name;
            }
```

```
}

//classes for protected modifier
class X
{
	protected int x;

	public X()
	{
		x = 10;
	}
}

class Y : X
{

	public int getX()
	{
		return x;
	}
}

//class for internal modifier
internal class Complex
{
	int real;
	int img;

	public void setData(int r, int i)
	{
		real = r;
		img = i;
	}

	public void displayData()
	{
		Console.WriteLine("Real = {0}", real);
		Console.WriteLine("Imaginary = {0}", img);
	}
}

//classes for private and private protected modifier
class Parent
{
	private int value;
	private protected int value1;

	public void setValue(int v)
	{
		value = v;
```

```
        }

        public int getValue()
        {
            return value;
        }
        public void setValue1(int v)
        {
            value1 = v;
        }

        public int getValue1()
        {
            return value1;
        }
    }
    class Child : Parent
    {
        public void showValue()
        {
            // Trying to access value
            // Inside a derived class
            // Console.WriteLine( "Value = " + value );
            // Gives an error
        }

        public void showValue1()
        {
            //Console.WriteLine("Value1 = " + value1);
        }
    }

//class for property
    class Student
    {
        private string code = "N.A";
        private string name = "not known";
        private int age = 0;

        // Declare a Code property of type string:
        public string Code
        {
            get
            {
                return code;
            }
            set
            {
                code = value;
            }
```

```
        }

        // Declare a Name property of type string:
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }

        // Declare a Age property of type int:
        public int Age
        {
            get
            {
                return age;
            }
            set
            {
                age = value;
            }
        }
        public override string ToString()
        {
            return "Code = " + Code + ", Name = " + Name + ", Age = " +
Age;
        }
    }

    //classes for indexer
    class A
    {
        string name;
        int i;
        public object this[int j]
        {
            get
            {
                if (j == 0)
                    return (string)name;
                else
                    return (int)i;
            }
            set
            {
```

```
                if (j == 0)
                    name = (string)value;

                else
                    i = (int)value;
            }
        }
    }
    class B
    {
        string[] name = new string[2];
        int[] i = new int[2];
        public object this[int j]
        {
            get
            {
                if (j == 0 || j == 1)
                {
                    return (string)name[j];
                }
                else if (j == 2)
                {
                    return i[0];
                }
                else
                {
                    return i[1];
                }
            }
            set
            {
                if (j == 0 || j == 1)
                { name[j] = (string)value; }


                else if (j == 2)
                {
                    i[0] = (int)value;
                }
                else
                {
                    i[1] = (int)value;
                }
            }
        }
    }

    //class for delegate
    class Delegate
    {
```

```
        static int num = 10;

        public static int AddNum(int p)
        {
            num += p;
            return num;
        }
        public static int MultNum(int q)
        {
            num *= q;
            return num;
        }
        public static int getNum()
        {
            return num;
        }
    }

    //class for event
    class Event
    {
        public event NumberChanger MyEvent;
        public Event()
        {
            this.MyEvent = new NumberChanger(this.MultiplyFive);
        }
        public int MultiplyFive(int n) //same signature as of delegate
        {
            return n*5;
        }
        public static void display()
        {
            Event obj1 = new Event();
            Console.WriteLine(obj1.MyEvent(5));
        }

    }
}
```