Module-3

14. Understanding Classes.

A class is like a blueprint of a specific object. In object-oriented programming, a class defines some properties, fields, events, methods, etc. In C#, a class can be defined by using the class keyword.

Types of classes:

Static class

In C#, a static class is a class that cannot be instantiated. Static classes are created using the static keyword in C#. A static class can contain static members only. You can't create an object for the static class. Static classes are sealed, means you cannot inherit a static class from another class.

- **Static Data Members:** As static class always contains static data members, so static data members are declared using static keyword and they are directly accessed by using the class name.
- **Static Methods:** As static class always contains static methods, so static methods are declared using static keyword. These methods only access static data members , they cannot access non-static data members.
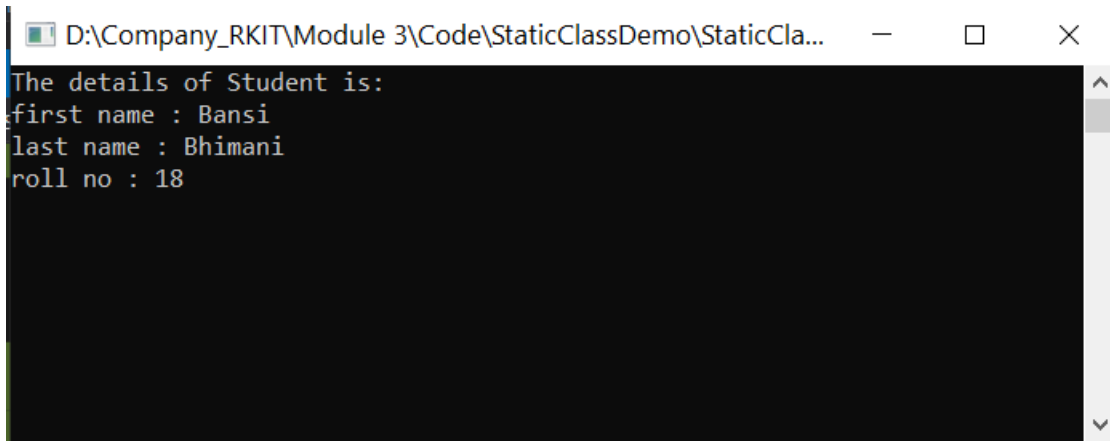
Code:

```csharp
using System;

namespace StaticClassDemo
{
    //Creating static class Using static keyword
    static class Student
    {
        // Static data members of Student
        public static string f_name = "Bansi";
        public static string l_name = "Bhimani";
        public static int roll_no = 18;
        // Static method of Student
        public static void details()
        {
            Console.WriteLine("The details of Student is:");
        }
    }
    class Program
    {
    // main method
        static void Main(string[] args)
        {
            // Calling static method of student
            Student.details();
            //Accessing the static data members of student
            Console.WriteLine("first name : {0} ", Student.f_name);
            Console.WriteLine("last name : {0} ", Student.l_name);
            Console.WriteLine("roll no : {0} ", Student.roll_no);
```

```
        Console.Read();

      }
   }
}
```

Output:

Abstract class

A class with abstract modifier indicate that class is abstract class. An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

Abstract Method: A method which is declared abstract, has no "body" and declared inside the abstract class only. An abstract method must be implemented in all non-abstract classes using the override keyword. After overriding the abstract method is in the non-Abstract class. We can derive this class in another class, and again we can override the same abstract method with it.

Code:

```csharp
using System;

namespace AbstractClassDemo
{
    //declare class 'AreaClass' as abstract
    abstract class AreaClass
    {
        // declare method 'Area' as abstract
        abstract public int Area();
    }
    // class 'AreaClass' inherit in child class 'Square'
    class Square : AreaClass
    {
        int side = 0;
        // constructor
        public Square(int n)
        {
            side = n;
        }
        // the abstract method 'Area' is overridden here
        public override int Area()
        {
            return side * side;
```
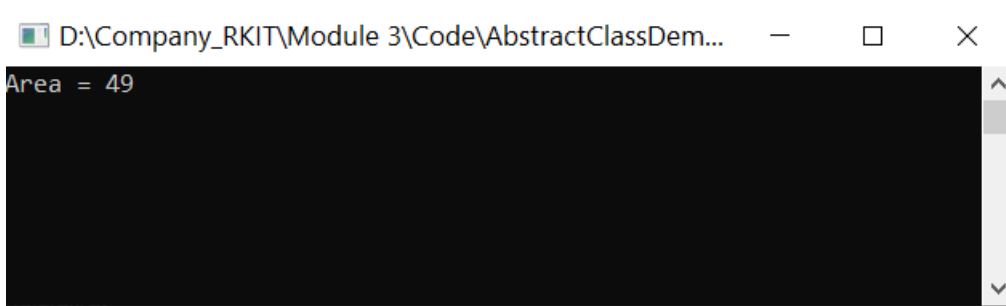
```
        }
    }

    class Program
    {
        // Main method
        static void Main(string[] args)
        {
            Square objs = new Square(7);
            Console.WriteLine("Area = " + objs.Area());
            Console.Read();

        }
    }
}
```

Output:


```
D:\Company_RKIT\Module 3\Code\AbstractClassDem...   —   □   ×
Area = 49
```

Sealed class

A class with sealed keyword indicates that class is sealed to prevent inheritance. Sealed class cannot inheritance.

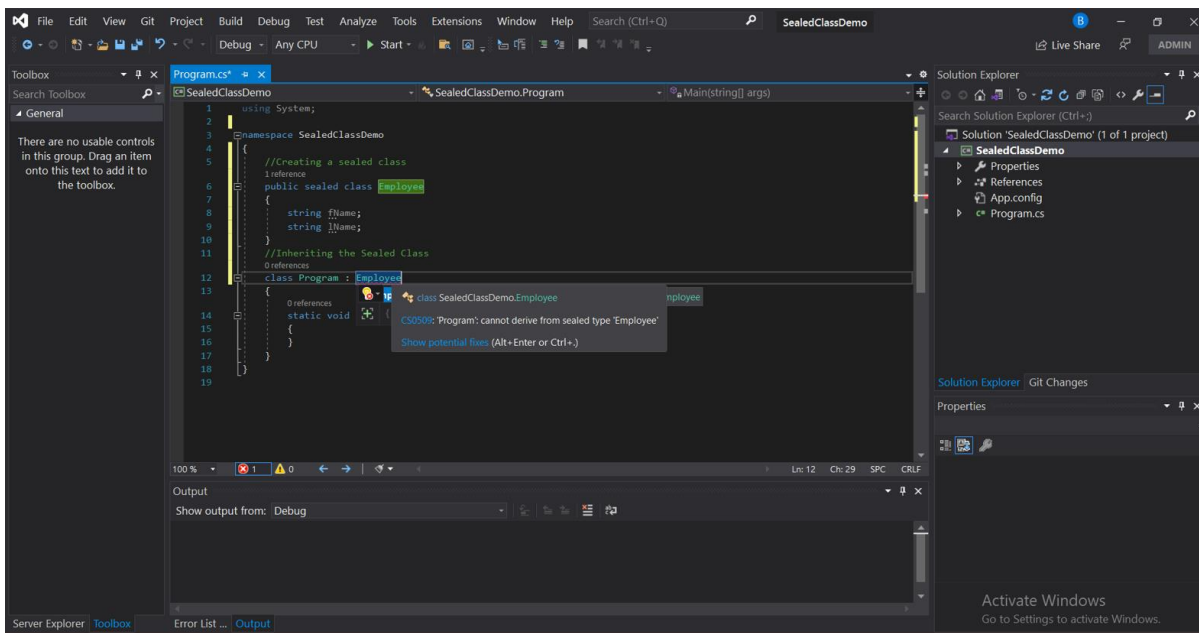**Code:**

```
using System;

namespace SealedClassDemo
{
    //Creating a sealed class
    public sealed class Employee
    {
        string fName;
        string lName;
    }
    //Inheriting the Sealed Class
    class Program : Employee
    {
        static void Main(string[] args)
        {
        }
    }
}
```
Error message is as shown as:

15. Depth in classes.

Objects: In C#, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc. In other words, object is an entity that has state and behaviour. Here, state means data and behaviour means functionality. Object is a runtime entity, it is created at runtime. Object is an instance of a class. All the members of the class can be accessed through object.

Properties: Properties are the special type of class members that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called **accessors**. It uses pre-defined methods which are "get" and "set" methods which help to access and modify the properties.

• **Read and Write Properties:** When property contains both get and set methods.

• **Read-Only Properties:** When property contains only get method.

• **Write Only Properties:** When property contains only set method.

• **Auto Implemented Properties:** When there is no additional logic in the property accessors and it introduce in C# 3.0.

Get Accessor: It specifies that the value of a field can access publicly. It returns a single value and it specifies the read-only property.

Set Accessor: It will specify the assignment of a value to a private field in a property. It returns a single value and it specifies the write-only property.

Method: A method is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also

known as functions. Why use methods? To reuse code: define the code once, and use it many times.

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.

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. The classes that receive (or handle) the event are called subscribers. 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. 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.

To declare an event inside a class, first of all, you must declare a delegate type for the event as:

public delegate string BoilerLogHandler(string str);

then, declare the event using the event keyword –

event BoilerLogHandler BoilerEventLog;

Example of events by delegated:

```csharp
using System;

namespace EventDemo
{
    //declare a 'MyDel' delegate
    public delegate string MyDel(string str);
    class EventProgram
    {
        //declare the 'MyEvent' event of delagate 'MyDel'
        event MyDel MyEvent;
        public EventProgram()
        {
            this.MyEvent += new MyDel(this.WelcomeUser);
        }
    public string WelcomeUser(string username)
        {
            return "Welcome " + username;
```
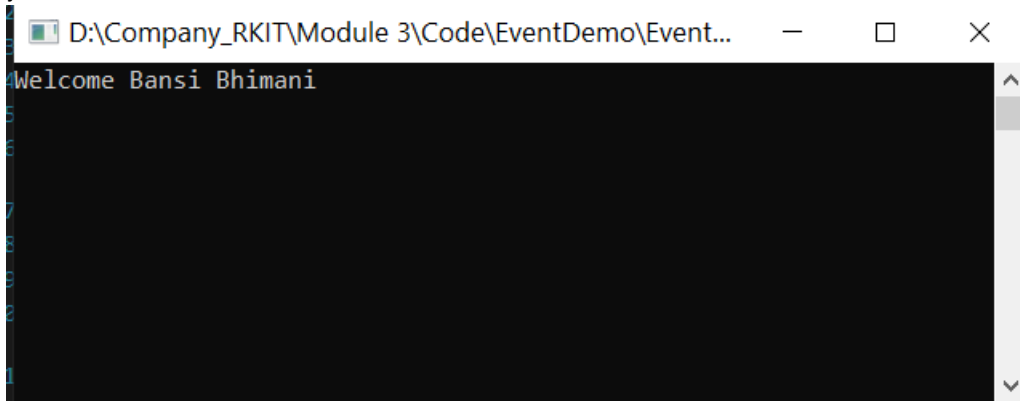
```
        }
        static void Main(string[] args)
        {
            //creating a object of EventProgram
            EventProgram obj1 = new EventProgram();
            string result = obj1.MyEvent("Bansi Bhimani");
            Console.WriteLine(result);
            Console.Read();


        }
    }
}
```


D:\Company_RKIT\Module 3\Code\EventDemo\Event...   —   □   ×
Welcome Bansi Bhimani

Example of properties:

```
using System;

namespace PropertiesDemo
{

    public class Student
    {
        // Declare name field
        private string name = "Bansi R. Bhimani";
        // Declare name property
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
    }

    class Program
    {
        public static void Main(string[] args)
        {
            //Creating a object of Student
            Student objs = new Student();
            // calls set accessor of the property Name ,and pass "Bansi Bhimani" as value of the standard field 'value'.
            objs.Name = "Bansi Bhimani";
            // displays Bansi Bhimani, Calls the get accessor of the property Name.
            Console.WriteLine("Name: " + objs.Name);
```
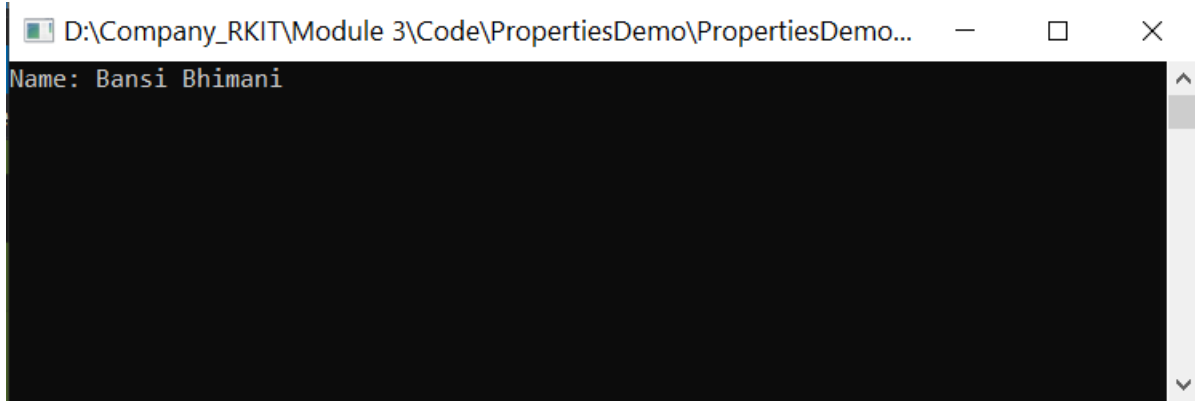
```
        Console.Read();

    }
  }
}
```


D:\Company_RKIT\Module 3\Code\PropertiesDemo\PropertiesDemo...
Name: Bansi Bhimani

16. Scope & Accessibility Modifiers.

Access Modifiers are keywords that define the accessibility of a member, class or datatype in a program. These are mainly used to restrict unwanted data manipulation by external programs or classes.

• public

• protected

• private

Public:

Access is granted to the entire program. This means that another method or another assembly which contains the class reference can access these members or types. This access modifier has the most permissive access level in comparison to all other access modifiers.

Example: Here, we declare a class Student which consists of two class members rollNo and name which are public. These members can access from anywhere throughout the code in the current and another assembly in the program. The method getName are also declared as public.

Code:

```csharp
using System;

namespace PublicAM
{
    // Declaring members rollNo and name as public
    class Student
    {
        public int rollNo;
        public string name;
        //constructor
        public Student(int r, string n)
```

```csharp
        {
            rollNo = r;
            name = n;
        }
        // method getName also declared as public
        public string getName()
        {
            return name;
        }
    }

    class Program
    {
        //Main method
        static void Main(string[] args)
        {
            // Creating object of the class Student
            Student objs = new Student(152, "Bansi");
            // Displaying details directly using the class members accessible through another method
            Console.WriteLine("Roll number: {0}", objs.rollNo);
            Console.WriteLine("Name: {0}", objs.name);
            Console.WriteLine();
            // Displaying details using member method also public
            Console.WriteLine("Name: {0}", objs.getName());
            Console.Read();


        }
    }
}
```

Output:


```
D:\Company_RKIT\Module 3\Code\AccessModifiers\PublicAM\PublicA...      —    □    ×
Roll number: 152
Name: Bansi

Name: Bansi
```

Protected:

Access is limited to the class that contains the member and derived types of this class. It means a class which is the subclass of the containing class anywhere in the program can access the protected members.

Example: In the code given below, the class Y inherits from X, therefore, any protected members of X can be accessed from Y but the values cannot be modified.

Code:

```csharp
using System;

namespace ProtectedAM
```

```
{
    class X
    {
        // Member x declared as protected
        protected int x;
        public X()
        {
            x = 100;
        }
    }
    // class Y inherits the class X
    class Y : X
    {
        // Members of Y can access 'x'
        public int getX()
        {
            return x;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            X obj1 = new X();
            Y obj2 = new Y();
            // Displaying the value of x
            Console.WriteLine("Value of x is : {0}", obj2.getX());
            Console.Read();
        }
    }
}
```
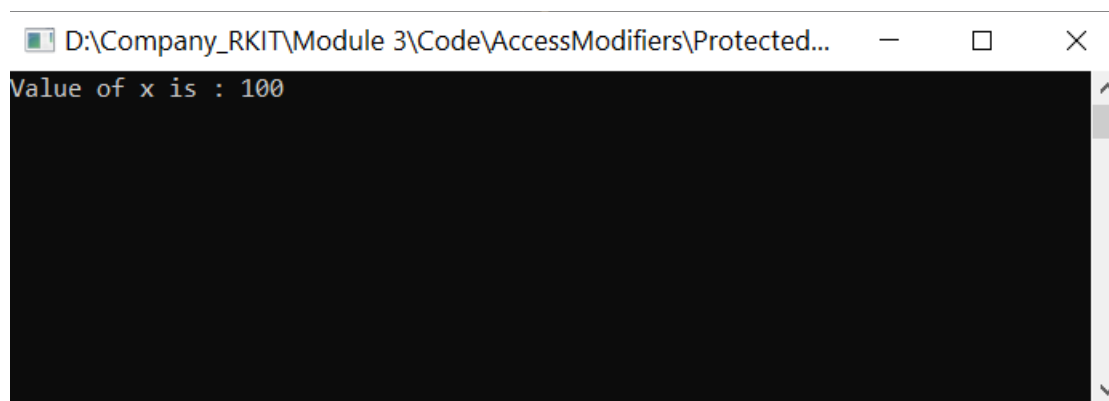
Output:

Private:

Access is only granted to the containing class. Any other class inside the current or another assembly is not granted access to these members.

Example: In this code we declare the member value of class Parent as private therefore its access is restricted to only the containing class. We try to access value inside of a derived class named Child but the compiler throws an error {error CS0122: 'PrivateAccessModifier.Parent.value' is inaccessible due to its protection level}. Similarly, inside

main {which is a method in another class}. objp.value will throw the above error. So we can use public member methods that can set or get values of private members.
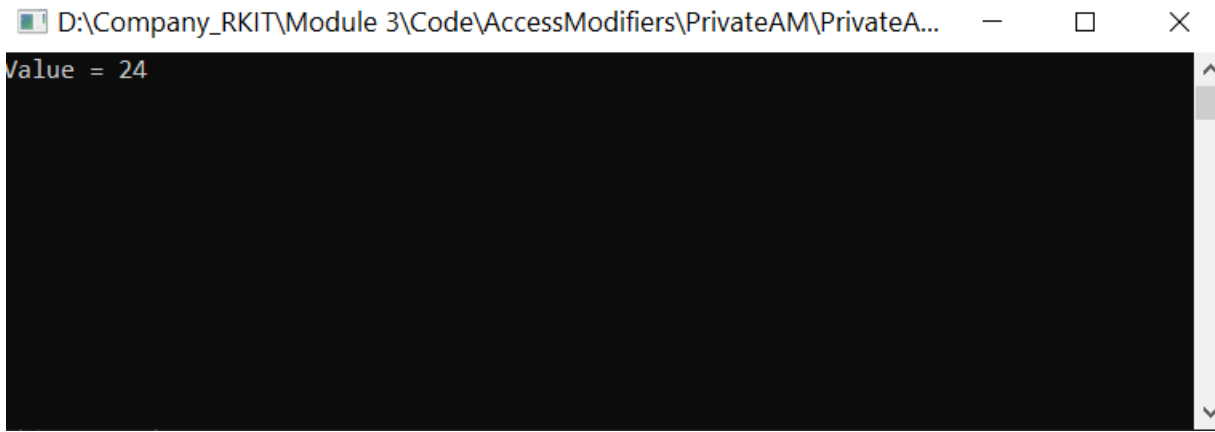
Code:

```csharp
using System;

namespace PrivateAM
{
    class Parent
    {
        // Member is declared as private
        private int value;
        // value is Accessible only inside the class
        public void setValue(int v)
        {
            value = v;
        }
        public int getValue()
        {
            return value;
        }
    }
    class Child : Parent
    {
        public void showValue()
        {
            // Trying to access value Inside a derived class
            // Console.WriteLine( "Value = " + value );
            // Gives an error
        }
    }

    class Program
    {
        //Main method
        static void Main(string[] args)
        {
            Parent objp = new Parent();
            // objp.value = 15;
            // Also gives an error
            // Use public functions to assign and use value of the member 'value'
            objp.setValue(24);
            Console.WriteLine("Value = " + objp.getValue());
            Console.Read();

        }
    }
}
```

Output:

```
Value = 24
```

17. Namespace & .Net Library.

Namespace: Namespaces are used to organize the classes. It helps to control the scope of methods and classes in larger .Net programming projects. It provides a way to keep one set of names(like class names) different from other sets of names. The biggest advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace. It is also referred as named group of classes having common features. The members of a namespace can be namespaces, interfaces, structures, and delegates.

To define syntax is:

namespace namespace_name {

// code declarations

}

To access member:

Syntax: [namespace_name].[member_name]

C# provides a keyword "using" which help the user to avoid writing fully qualified names again and again.

Syntax: using [namespace_name][.][sub-namespace_name];

.net library:

The Class Library contains program code, data, and resources that can be can used by other programs and are easily implemented into other Visual Studio projects. In visual studio we can implement code in project type as class library and then it can be referred by other programs or even in same program.

Simple program that demonstrates use of class library is shown below. I have prepared one class library project and another console application which uses method specified in the class library.

Class library code:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestClassLibrary
{
    public  class Algebra
    {
        public int add(int a, int b)
        {
            return a + b;
        }
        public int sub(int a, int b)
        {
            return a - b;
        }
    }
}
```
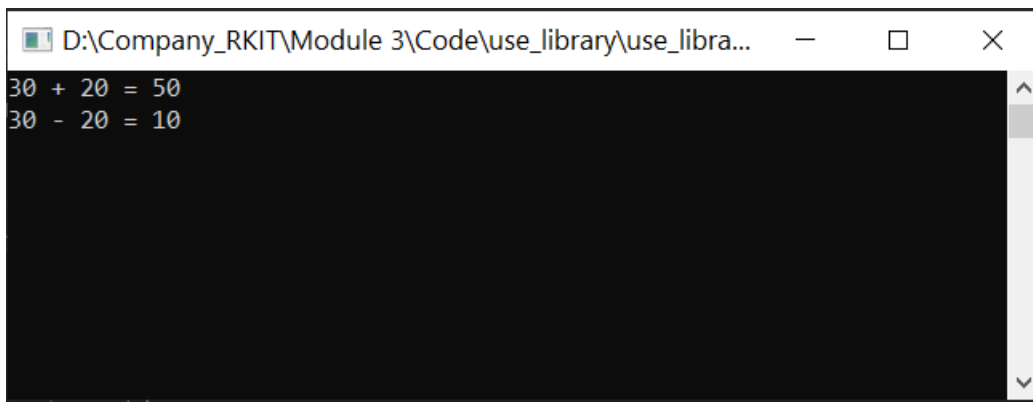
Console application which uses the class library method:

```csharp
using System;
using TestClassLibrary;

namespace use_library
{
    class Program
    {
        static void Main(string[] args)
        {
            Algebra obja = new Algebra();
            int n1 = 30;
            int n2 = 20;
            int result = obja.add(n1, n2);
            Console.WriteLine("{0} + {1} = {2}", n1, n2, result);
            result = obja.sub(n1, n2);
            Console.WriteLine("{0} - {1} = {2}", n1, n2, result);
            Console.Read();
        }
    }
}
```

Output:

```
D:\Company_RKIT\Module 3\Code\use_library\use_libra...   —   □   ✕
30 + 20 = 50
30 - 20 = 10
```

18. Creating and adding ref. to assemblies.

Assemblies form the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.

In .NET Core and .NET Framework, you can build an assembly from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This allows larger projects to be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly.

An assembly is a file that contains the compiled code. There are also library assemblies, which have the DLL file extension. DLL stand for dynamic link library. A class library is a name for dynamic link library (DLL). An assembly is a distributable file in .NET and it can be a DLL or an exe. Following example illustrates to create a dll file and add reference to a project in the same solution.

dll file code:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestClassLibrary
{
    public class Algebra
    {
        public int add(int a, int b)
        {
            return a + b;
        }
        public int sub(int a, int b)
        {
            return a - b;
        }
    }
}
```
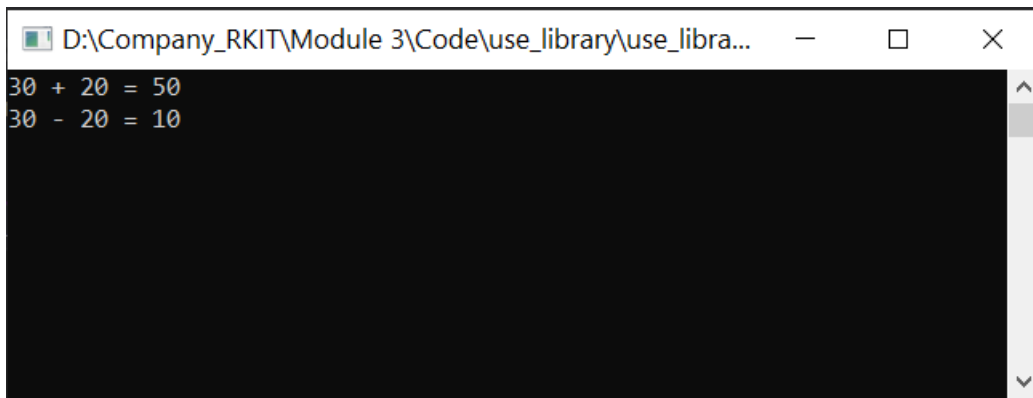
```
    }
}
```

Console application which uses the class library method:

```csharp
using System;
using TestClassLibrary;

namespace use_library
{
    class Program
    {
        static void Main(string[] args)
        {
            Algebra obja = new Algebra();
            int n1 = 30;
            int n2 = 20;
            int result = obja.add(n1, n2);
            Console.WriteLine("{0} + {1} = {2}", n1, n2, result);
            result = obja.sub(n1, n2);
            Console.WriteLine("{0} - {1} = {2}", n1, n2, result);
            Console.Read();
        }
    }
}
```
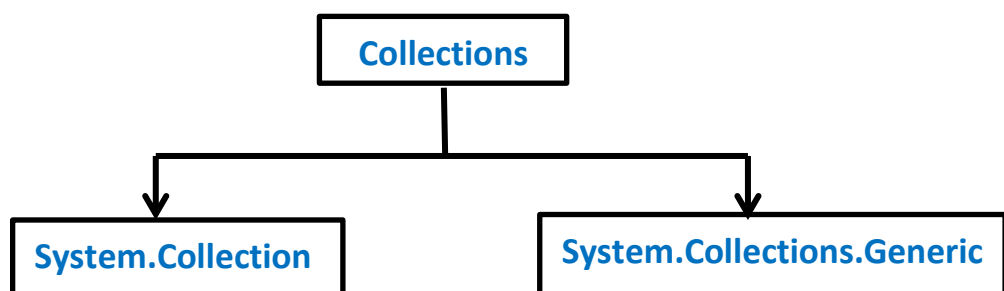
Output:


```
D:\Company_RKIT\Module 3\Code\use_library\use_libra...    —    □    ×
30 + 20 = 50
30 - 20 = 10
```

19. Working with collections.

Collections standardize the way of which the objects are handled by your program. In other words, it contains a set of classes to contain elements in a generalized manner. With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.

C# divide collection in several classes, some of the common classes are shown below:

Generic collection in C# is defined in System.Collection.Generic namespace. It provides a generic implementation of standard data structure like linked lists, stacks, queues, and dictionaries.

Class name	Description
Dictionary<TKey,TValue>	It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.
List<T>	It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class.
Queue<T>	A first-in, first-out list and provides functionality similar to that found in the non-generic Queue class.
SortedList<TKey,TValue>	It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class.
Stack<T>	It is a first-in, last-out list and provides functionality similar to that found in the non-generic Stack class.
HashSet<T>	It is an unordered collection of the unique elements. It prevent duplicates from being inserted in the collection.
LinkedList<T>	It allows fast inserting and removing of elements. It implements a classic linked list.

Non-Generic collection in C# is defined in System.Collections namespace. It is a general-purpose data structure that works on object references, so it can handle any type of object, but not in a safe-type manner.

Class name	Description
ArrayList	It is a dynamic array means the size of the array is not fixed, it can increase and decrease at runtime.
Hashtable	It represents a collection of key-and-value pairs that are organized based on the hash code of the key.
Queue	It represents a first-in, first out collection of objects. It is used when you need a first-in, first-out access of items.
Stack	It is a linear data structure. It follows LIFO(Last In, First Out) pattern for Input/output.

Implementation of list:
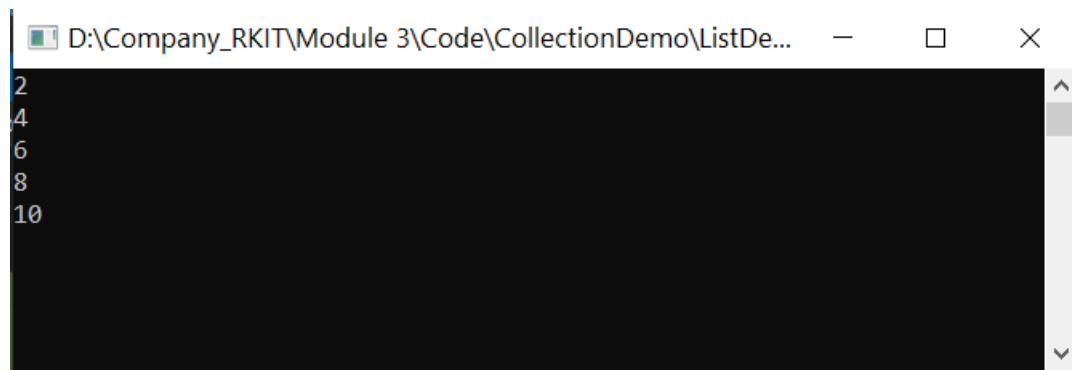
```
using System;
using System.Collections.Generic;

namespace ListDemo
{
    class Program
```

```csharp
    {
        static void Main(string[] args)
        {
            // Creating a List of integers
            List<int> mylist = new List<int>();
            // adding items in mylist
            for (int j = 1; j < 6; j++)
            {
                mylist.Add((j * 2));
            }
            // Displaying items of mylist by using foreach loop
            foreach (int items in mylist)
            {
                Console.WriteLine(items);
            }
            Console.Read();

        }
    }
}
```

Output:

```
D:\Company_RKIT\Module 3\Code\CollectionDemo\ListDe...    —    ☐    ✕
2
4
6
8
10
```

Implementation of non-generic queue:

```csharp
using System;
using System.Collections;

namespace QueueDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Creating a Queue
            Queue myQueue = new Queue();
            // Inserting the elements into the Queue
            myQueue.Enqueue("ab");
            myQueue.Enqueue("cd");
            myQueue.Enqueue("ef");
            myQueue.Enqueue("gh");
            myQueue.Enqueue("ij");
            // Displaying the count of elements
            // contained in the Queue
            Console.Write("Total number of elements present in the Queue are: ");
            Console.WriteLine(myQueue.Count);
            // Displaying the beginning element of Queue
            Console.WriteLine("Beginning Item is: " + myQueue.Peek());
            Console.Read();
```
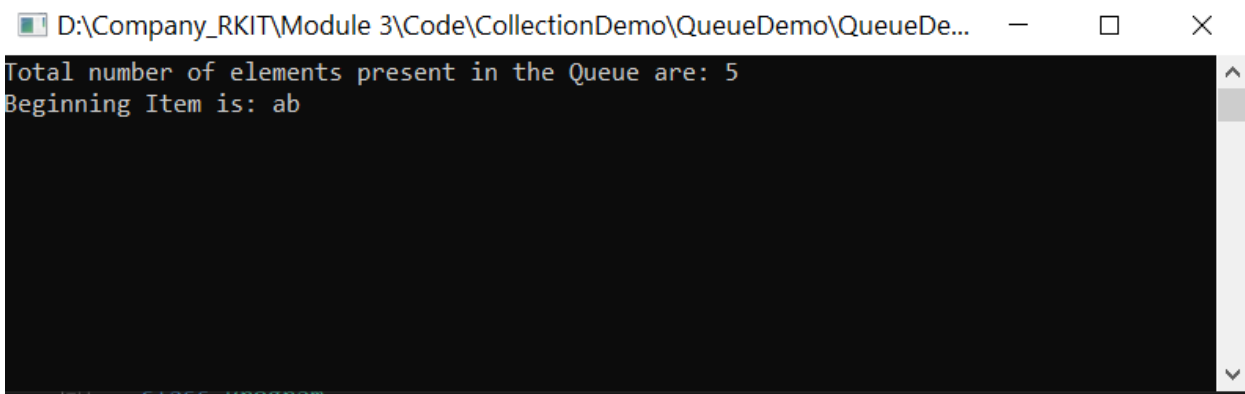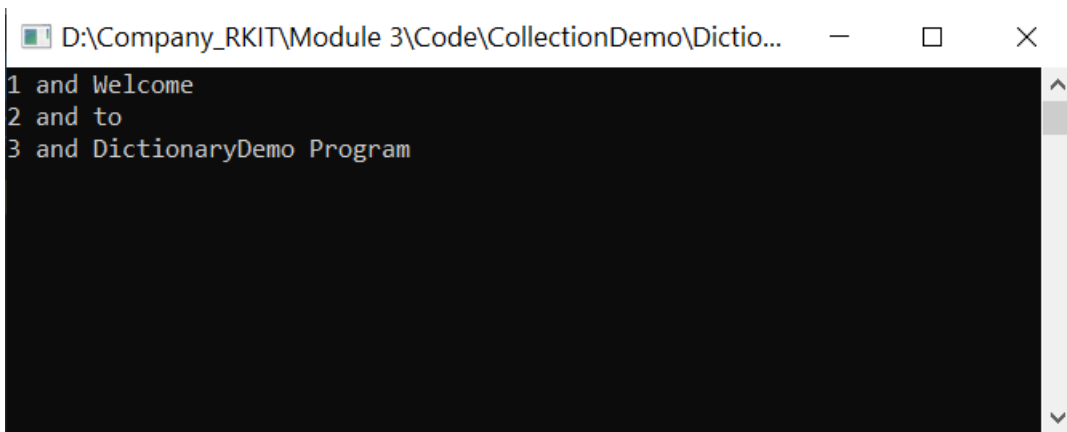
```
        }
    }
}
```

Output:

Implementation of dictionary:

```csharp
using System;
using System.Collections.Generic;
namespace DictionaryDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<int, string> dict = new Dictionary<int, string>();
            // Adding key/value pairs
            dict.Add(1, "Welcome");
            dict.Add(2, "to");
            dict.Add(3, "DictionaryDemo Program");
            // Displaying pair of dict by using foreach loop
            foreach (KeyValuePair<int, string> pair in dict)
            {
                Console.WriteLine("{0} and {1}",
                pair.Key, pair.Value);
            }
            Console.WriteLine();
            Console.Read();
        }
    }
}
```

Output: