**14. Understanding classes**

**14.1 Classes types and usage**

* Class is a template or blueprint of the program which contain relative variables and methods.
* A class defines the kinds of data and the functionalities their project will have.
* Classes are the reference types.
* Syntax: <access modifier> Class class\_name {}

**14.2 Types of classes**

1. **Static Class**

* A class with a static keyword which contains only static members, is defined as static class.
* Static class can only contain static data members, static methods and a static constructer.
* A static class cannot initialize using a new keyword, It is not allowed to create objects of the static class.
* You cannot inherit a static class from another class.
* Syntax:

static class class\_name

{

//static data members

// static method

}

1. **Sealed Class**

* Sealed classes are used to restrict the users from inheriting the class.
* A class can be sealed using the **sealed** keyword.
* The keyword tells compiler that the class is sealed, therefore can not be extended.
* No class can be derived from a sealed class.
* Syntax:

sealed class class\_name

{

// data members and methods

}

1. **Abstract Class**

* Abstraction is a process to hide the internal details and showing only the functionality.
* Abstraction class is a way to achieve the abstraction.
* A class can be abstract using the keyword **abstract**.
* It can only be used as a BASE class.
* Generally, we use the abstract class at the time of inheritance
* It can contain abstract and non-abstract methods.
* Abstract method has no body.
* To use abstract class we have to inherit it from another class and provide implementations to the abstract method in it.
* Syntax:

abstract class name { }

**15. Depth in a class**

1. **Object**

* In c# object is a instance of a class that can be used to access the data members and member functions of a class.
* A program may create many objects of the same class.
* In c# we can create objects by using a new keyword followed by the name of the class.
* Syntax:

Class Name object Name = new Class Name ();

1. **Methods**

* A method is a group of statements that together perform a task.
* Every c# program has atleast one class with a method named main.
* To use a method we need to

1. Define the method
2. Call the method

* Defining method Syntax:

<access modifier> <Return type> <method name> (Param list)

{

//Method body

}

* There is a several types of methods.

1. **Abstract Method**

* A method which contains Abstract keyword is known as abstract method.
* Abstract method have no implementation, It means this methods have no body.
* Syntax:

<Access-modifier> abstract <Return type> <Methodname>();

1. **Virtual method**

* Virtual methods makes some default functionality, inother words virtual methods are being implemented in the base class and can be overridden in derived class.
* Syntax:

<access modifier> virtual <Return type> <method name> (Param list)

{

//Method body

}

1. **Static method**

* We can define static methods sing static keyword.
* It does not belong to instance of the type, we can access static methods by classname.
* Syntax:

<access modifier> staticl <Return type> <method name> (Param list)

{

//Method body

}

1. **Instance method**

* An instance method operateson a given instance o a class, and that instance can be accessed as this.
* Syntax:

<access modifier> <Return type> <method name> (Param list)

{

//Method body

}

1. **Properties**

* A property is a member 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 actually they are special methods called accessors.
* Property enables a class to expose a public way of getting and setting values.
* A get property accessor is used to return the property value, and set property accessor is used to assign a new value.
* Properties can be

1. Read-write (having both a get and a set accessor)
2. Read-only (having a get accessor but no set accessor)
3. Write-only (having a set accessor but no get accessor)

* There is also auto implement properties this property make property-declaration more concise when no additional logic is required in the property accessor.
* Syntax:

<access modifier><Return type <Property name>

{

get { // body }

set { // body }

}

1. **Events**

* Event enable a class or object to notify other classes or objects when something of interest occurs.
* The class that sends(Raises) the event is called publisher and the classes that receives (Handle) the event are called subscriber.
* The publisher determines when an event is raised, The subscriber determines what action is taken in response to the event.
* An event can have a multiple subscribers. Subscriber can handle multiple events from multiple publishers.
* In C# events are encapsulated delegates so first, we need to declare delegate before we declare an event inside of a class by using event keyword.
* Event that have no subscriber are never raised.
* An event can be declared in two steps.

1. Declare a delegate
2. Declare the event using event keyword

* In C# to raise an event we need to invoke the evnt delegate and subscribe to the event using +=operator. In case , if you want to unsubscribe from an event, then use -= operator.

The event will enable a class or object to notify other class or object when something special happens.

**16. Scop & Accessibility Modifiers**

* Access modifiers are used to specify the scope of a member of class or type of the class itself.
* Access modifiers are part of object oriented programming which used to implement encapsulation of oop.

1. **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.
* This access modifier has the most permissive access level in comparison to all other access modifiers.
* Syntax:

public Typename

1. **Private**

* Access is only granted to the containing class.
* Any other class inside the current or another assembly is not graned access to this members.
* Syntax:

private Typename

1. **Protected**

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

protected Typename

1. **Internal**

* Access is limited to only the current assembly, that is any class or type declared as internal is accessible anywhere inside the same namespace.
* Syntax:

Internal Typename

**17. Namespace & .Net Library**

**Namespace**

* Namespaces are used to organize your code and classes, The member of a namespace can be namespaces, interfaces, structure, and delegates.
* We can use the fully qualified name instead of the namespace but it will make code lengthy and complex, so use of namespace is more efficient.
* There is availability of many namespaces in .net library we can include them as per our requirements in our application.
* We can include the namespaces by using keyword.
* Syntax:

Using namespace-name;

* Example:

Using System;

**.Net Library**

* .NET Library include classes, interfaces, delegates, and value types that provide access to system functionality.
* .NET types are the foundation on which .NET applications, components, and controls are built. .NET implementations include types that perform the following functions:
* Represent base data types and exceptions.
* Encapsulate data structures.
* Perform I/O.
* Access information about loaded types.
* Invoke .NET Framework security checks.
* Provide data access, rich client-side GUI, and server-controlled, client-side GUI.
* .NET provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as it is or, in many cases, derive your classes from them. To use the functionality of an interface
* We can either create a class that implements the interface or derive a class from one of the .NET classes that implements the interface.

**18. Creating and adding ref. to assemblies**

* Normally, Manually adding assembly references to our project is not required since visual studio is able to automatically add a appropriate references to the project.
* However there will be times when we want to manually add a reference to an assembly to our project.
* By adding the reference of another project in our current project we can use the classes and methods of that project in our project.
* WE also need to add the namespace of that reference in our project to use that project.
* To adding the reference in current project ,
* First find the reference folder from visual studio solution explorer.
* Then right click on the reference folder and select the add reference menu option to open add reference dialog.
* Locate the assembly in the add reference dialog and click the select button.
* Once we have selected the assembly, click the Ok button to add the reference and close the dialog.
* We also need to add the namespace of the reference in our program to use that reference in our project.

**19. Working with collections**

* Collection is a set of related data that may or may not necessarily belongs to the same datatype.
* Collection is a class that is useful to manage a group of object in flexible manner to perform various operations like insert, update, delete etc .
* Collection is a dynamic array.
* Array has size limit but objects stored in collection can grow or shrink dynamically based on our requirements
* The collection is a class so we just need to declare an instance of the class before we perform any operations

**Types of collections**

1. **Non-Generic collections**

* Non-Generic classes are useful to store elements of different data types.
* This are provided by **System,Collection** namespace.

1. **ArrayList**

* It is useful to represent an array of objects whose size is dynamically increased as required.
* We need to declare an instance of ArrayList class before we perform any operation.
* Syntax:

ArrayList arrList = new ArrayList ()

* There is some methods and properties of array list like add, insert, remove, capacity, count, add range, sort by which we can perform operations on ArrayList.

;

1. **HashTable**

* In c#, HashTable is used to store a collection of key/value pairs of different [data types](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples) and those are organized based on the hash code of the key.
* In HashTable, keys are not pre-defined it means we can explicitly define user-define keys in hash table.
* Syntax:

Hashtable htbl = new Hashtable ();

* It includes properties like count and methods like add, clear, clone,contains, containskey, containsvalue, remove and GetHash.

**c) Stack**

* In c#, Stack is useful to represent a collection of objects which store elements in LIFO (Last in, First out) style i.e. the element which added last will be the first to come out.
* In the Stack Push() method is useful to add elements to the stack and the  Pop() / Peek() method is useful to retrieve elements from the stack.
* Syntax:

Stack my stack = new stack ();

* Stack includes properties and methods like count, push, pop, peeks, contains, clone, clear etc.

**d) Queue**

* In c#, Queue is useful to represent a collection of objects which stores elements in FIFO (First In, First Out) style i.e. the element which added first will come out first.
* By using Enqueue() and Dequeue() methods, we can add or delete an element from the queue. Here, the Enqueue() method is useful to add elements at the end of the queue and the Dequeue() method is useful to remove elements start from the queue.
* Syntax:

Queue myqueuqe = new Queue ();

* Queue includes properties and methods like count, Enqueue, Dequeue, peeks, contains, clone.

1. **Generic collections**

* In c#, generic collections will enforce a type safety so you can store only the elements which are having the same [data type](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples).
* It provides the auto-sizing.
* It Is typesefe.
* In Non-Genericcollection, we learned that  non-generic collections are useful to store elements of different [data types](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples) but the problem with that is while retrieving an elements we need to perform a typecasting (boxing and unboxing) and it will affect application performance so to solve this problem c# introduced a generic collections.
* This are provided by **System,Collection.Generic** namespace.

1. **List<T>**

* List is a generic type of collection so it will allow storing only strongly typed objects i.e. elements of the same [data type](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples) and the size of the list will vary dynamically based on our application requirements like adding or removing elements from the list.
* In c#, the list is same as an [ArrayList](https://www.tutlane.com/tutorial/csharp/csharp-arraylist) but the only difference is [ArrayList](https://www.tutlane.com/tutorial/csharp/csharp-arraylist) is a non-generic type of [collection](https://www.tutlane.com/tutorial/csharp/csharp-collections) so it will allow storing elements of different [data types](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples).
* Syntax:

List<T> MyList = new List<T>();

1. **Dictionary<Tkey, Tvalue>**

* The dictionary object is same as [hashtable](https://www.tutlane.com/tutorial/csharp/csharp-hashtable) object but the only difference is the dictionary object is used to store a key-value pair of same [data type](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples) elements.
* While storing the elements in the dictionary object, you need to make sure that the keys are unique because the dictionary object will allow us to store duplicate values but the keys must be unique.
* Syntax:

Dictionary<TKey, TValue> Mydct = new Dictionary<TKey, TValue>();

1. **SortedList<Tkey, Tvalue>**

* SortedList is a generic type of collection and it is used to store a collection of key/value pairs that are sorted by key based.
* By default, the sortedlist will sort a key/value pairs in ascending order of the key and the sortedlist will allow storing only the strongly-typed objects i.e. the key/value pairs of the specified [data type](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples).
* Tthe sortedlist will allow us to store duplicate values but the keys must be unique and cannot be null to identify the values in sortedlist and the size of sortedlist will vary dynamically so you can add or remove elements from the sortedlist based on our application requirements.
* Syntax:

SortedList<TKey, TValue> slist = new SortedList<TKey, TValue>();

1. **Stack<T>**

* Generic stack is same as non-Generic stack expect that this stack used to same datatype elements.
* Syntax:

Stack<T> mystack = new Stack<T>();

1. **Queue<T>**

* Generic queue is same as non-Generic queue expect that this stack used to same datatype elements.
* Syntax:

Queue<T> myqueue = new Queue<T>();

**PROGRAMS**

1. **Types of Classes**

using System;

namespace TypeOfClass

{

public static class Author

{

public static string A\_name;

public static int age;

public static void display()

{

Console.WriteLine("Name of the Author is: " + A\_name);

Console.WriteLine("Age of the Author is: " + age);

}

}

public sealed class Employee

{

public string E\_name;

public void display()

{

Console.WriteLine("Name of the employee is: "+ E\_name);

}

}

public abstract class person

{

public string F\_name;

public string L\_name;

public abstract void display();

}

public class student : person

{

public int Standard;

public override void display()

{

Console.WriteLine("Name of the student is: " +F\_name + " " + L\_name);

Console.WriteLine("Standard of the student is: " + Standard);

}

}

public class teacher : person

{

public int Salary;

public override void display()

{

Console.WriteLine("Name of the teacher is: " + F\_name + " " + L\_name);

Console.WriteLine("Salary of the teacher is: " + Salary);

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Example of static class");

Author.A\_name = "Tom";

Author.age = 21;

Author.display();

Console.WriteLine("----------------------------------");

Console.WriteLine("Example of Sealed class");

Employee e1 = new Employee();

e1.E\_name = "John";

e1.display();

Console.WriteLine("----------------------------------");

Console.WriteLine("Example of Abstract class");

student s1 = new student();

s1.F\_name = "Nirali";

s1.L\_name = "Lathiya";

s1.Standard = 6;

s1.display();

Console.WriteLine("");

teacher t1 = new teacher();

t1.F\_name = "Priya";

t1.L\_name = "Gosai";

t1.Salary = 300000;

t1.display();

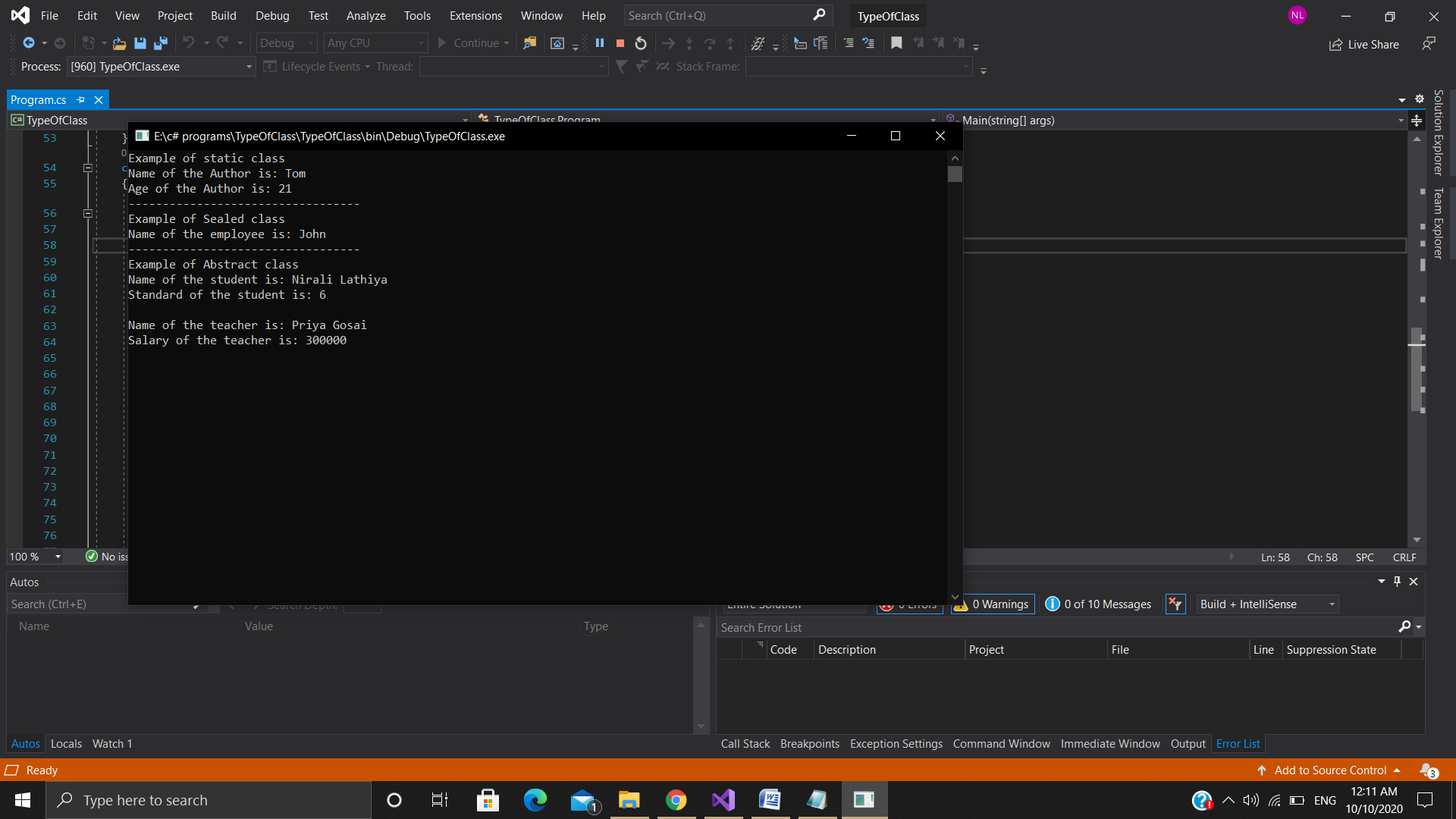
Console.ReadLine();

}

}

}

**Output**

****

1. **Properties**

using System;

namespace Properties

{

class student

{

private string name;

public string Name {

set

{

if (string.IsNullOrEmpty(value))

{

Console.WriteLine("Please enter the name...!!");

}

else

{

this.name = value;

}

}

get { return this.name; }

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Example of Read-Write Property");

student s1 = new student();

Console.WriteLine("Plaese enter your name: ");

s1.Name=Console.ReadLine();

Console.WriteLine("Getting the value by properties is: " + s1.Name);

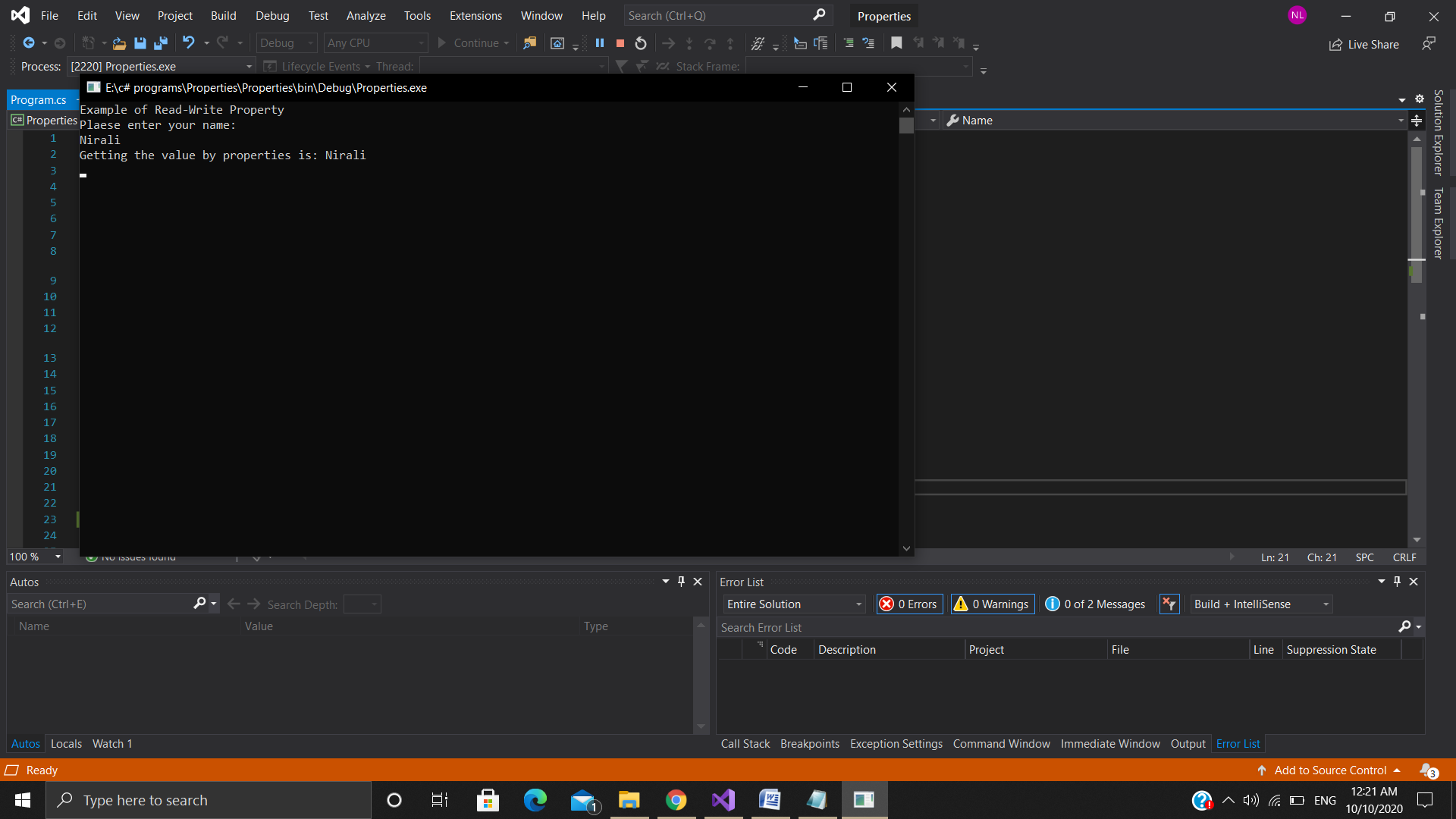
Console.ReadLine();

}

}

}

**Output**

****

1. **Events**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace simpleEvent

{

public delegate void EventHandler();

class Program

{

public static event EventHandler \_show;

static void Main(string[] args)

{

\_show += new EventHandler(Add);

\_show += new EventHandler(Sub);

\_show += new EventHandler(Div);

\_show.Invoke();

Console.ReadLine();

}

static void Add()

{

Console.WriteLine("This is Addition");

}

static void Sub()

{

Console.WriteLine("This is Subtraction");

}

static void Div()

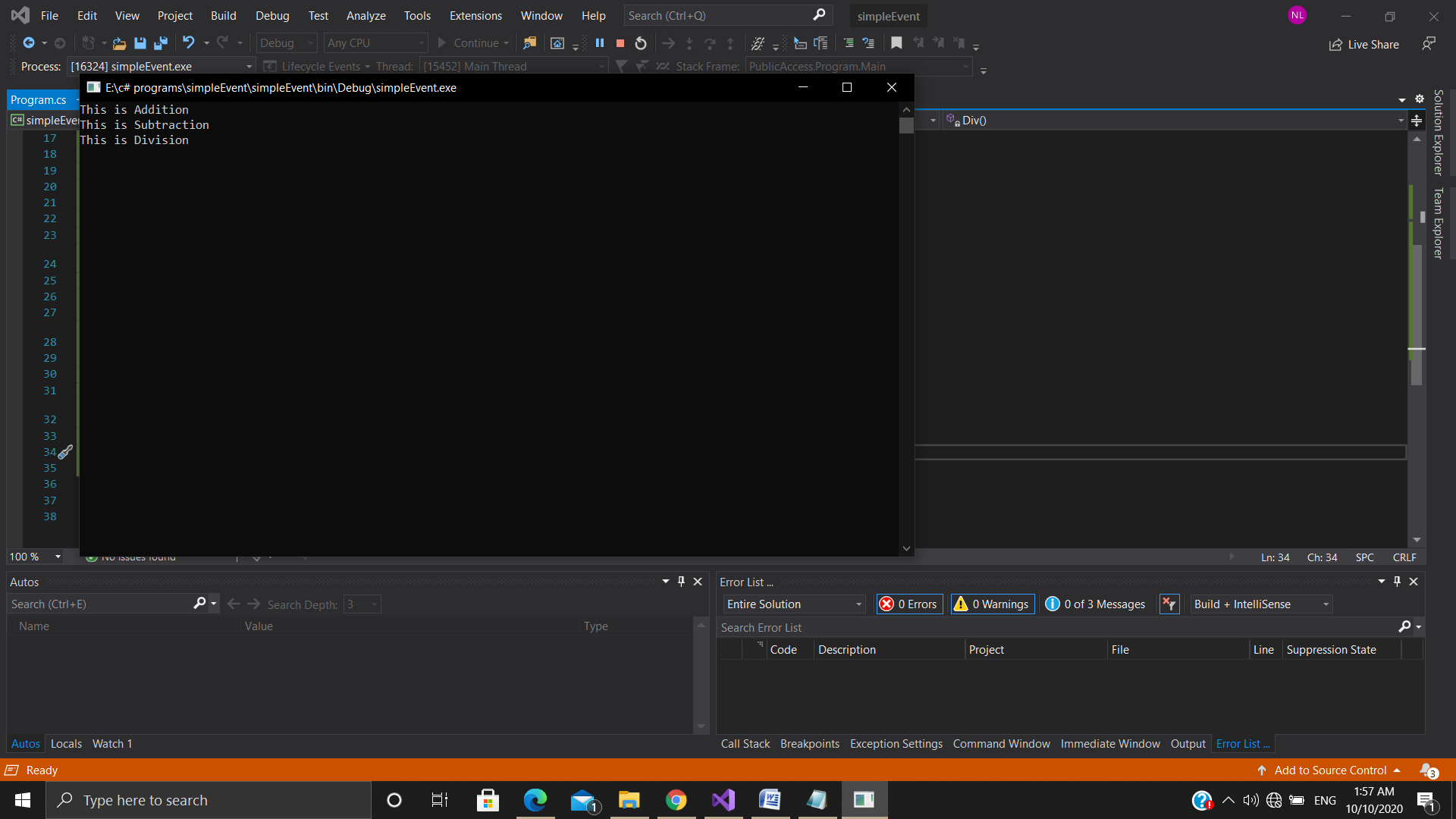
{

Console.WriteLine("This is Division");

}

}

**Output**

****

1. **Creating and adding reference to the assemblies**

**Creating a class Library**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MathLibrary

{

public class Class1

{

public float addition(float a, float b)

{

return a + b;

}

public float subtraction(float a, float b)

{

return a - b;

}

public float multiply(float a, float b)

{

return a \* b;

}

public float division(float a, float b)

{

return a / b;

}

}

}

**Adding Reference of this Library in another Assembly**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using MathLibrary;

namespace MathApplication

{

class Program

{

static void Main(string[] args)

{

Class1 math = new Class1();

float addition = math.addition(34.32f, 32.98f);

float substract = math.subtraction(84, 56);

float multiplication = math.multiply(5, 2);

float division = math.division(10, 2);

Console.WriteLine("addition is: " + addition);

Console.WriteLine("substraction is: " + substract);

Console.WriteLine("multiplication is: " + multiplication);

Console.WriteLine("division is: " + division);

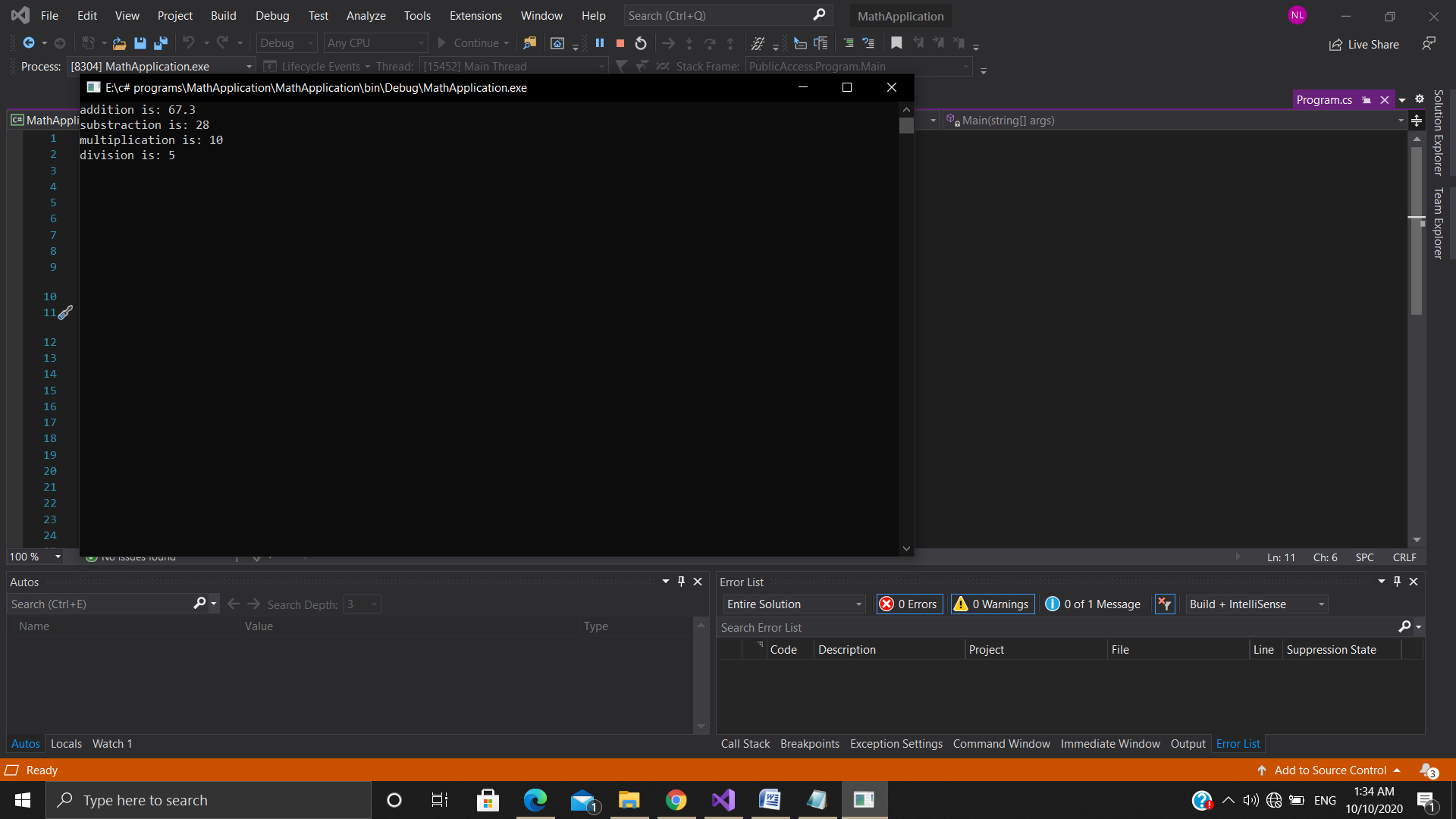
Console.ReadLine();

}

}

**}**

**Output**

****

1. **PublicAccessModifier**

using System;

namespace PublicAccess

{

class Student

{

public string name;

public int rollno;

public void Display()

{

Console.WriteLine("Name of the student is: {0}",name);

Console.WriteLine("Rollno of the student is {0}",rollno);

}

}

class Program

{

static void Main(string[] args)

{

Student s1 = new Student();

Console.WriteLine("Enter the student name: ");

s1.name = Console.ReadLine();

Console.WriteLine("Enter the student rollno: ");

s1.rollno = int.Parse(Console.ReadLine());

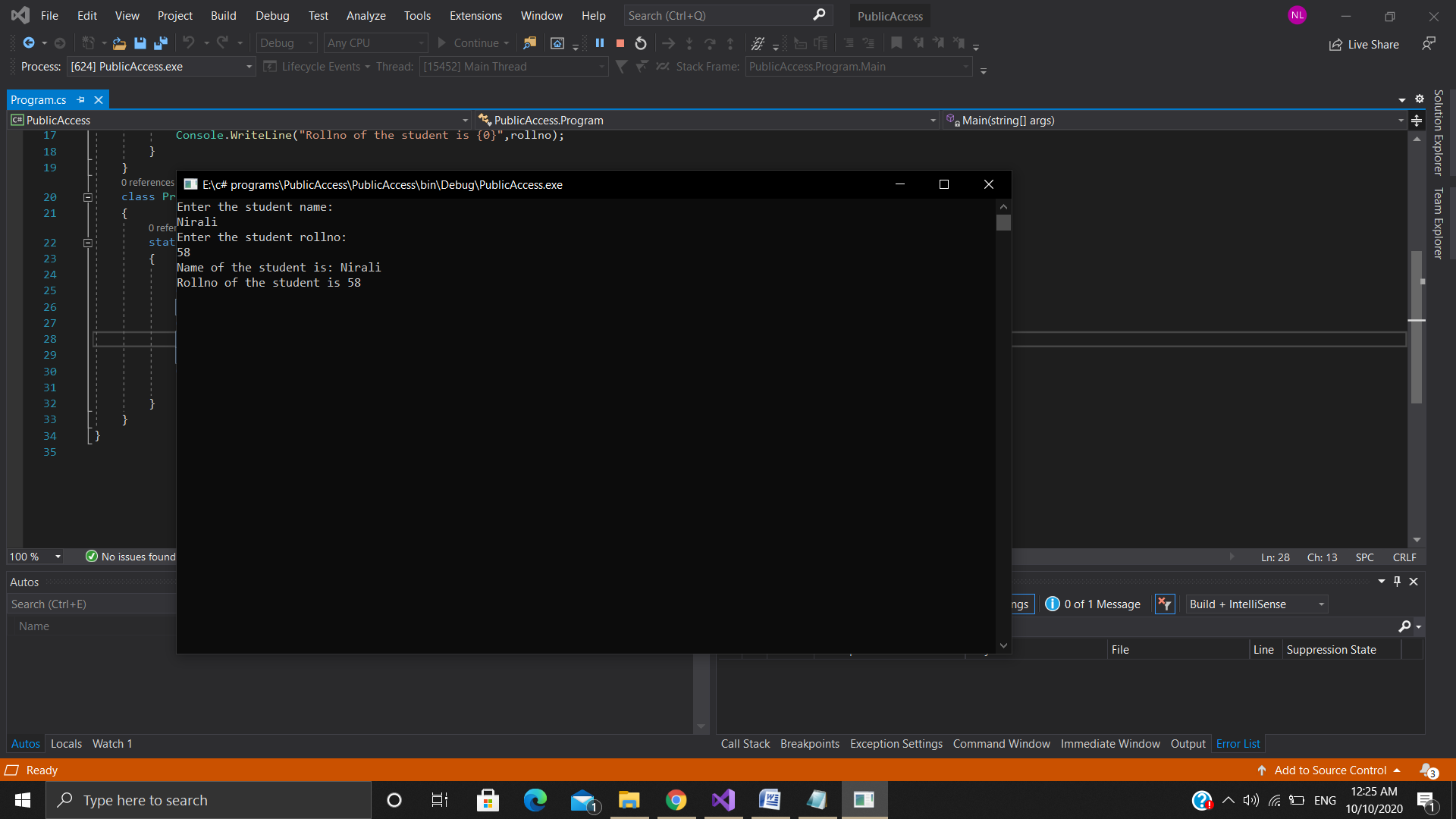
s1.Display();

Console.ReadLine();

}

}

**Output**

****

1. **PrivateAccessModifier**

using System;

namespace PrivateAccess

{

class student

{

private string name;

public string \_Name {

set {

this.name = value;

}

get

{

return this.name;

}

}

}

class Program

{

static void Main(string[] args)

{

student s1 = new student();

Console.WriteLine("Enter the name of the student: ");

s1.\_Name = Console.ReadLine();

Console.WriteLine("Name of the student is " + s1.\_Name);

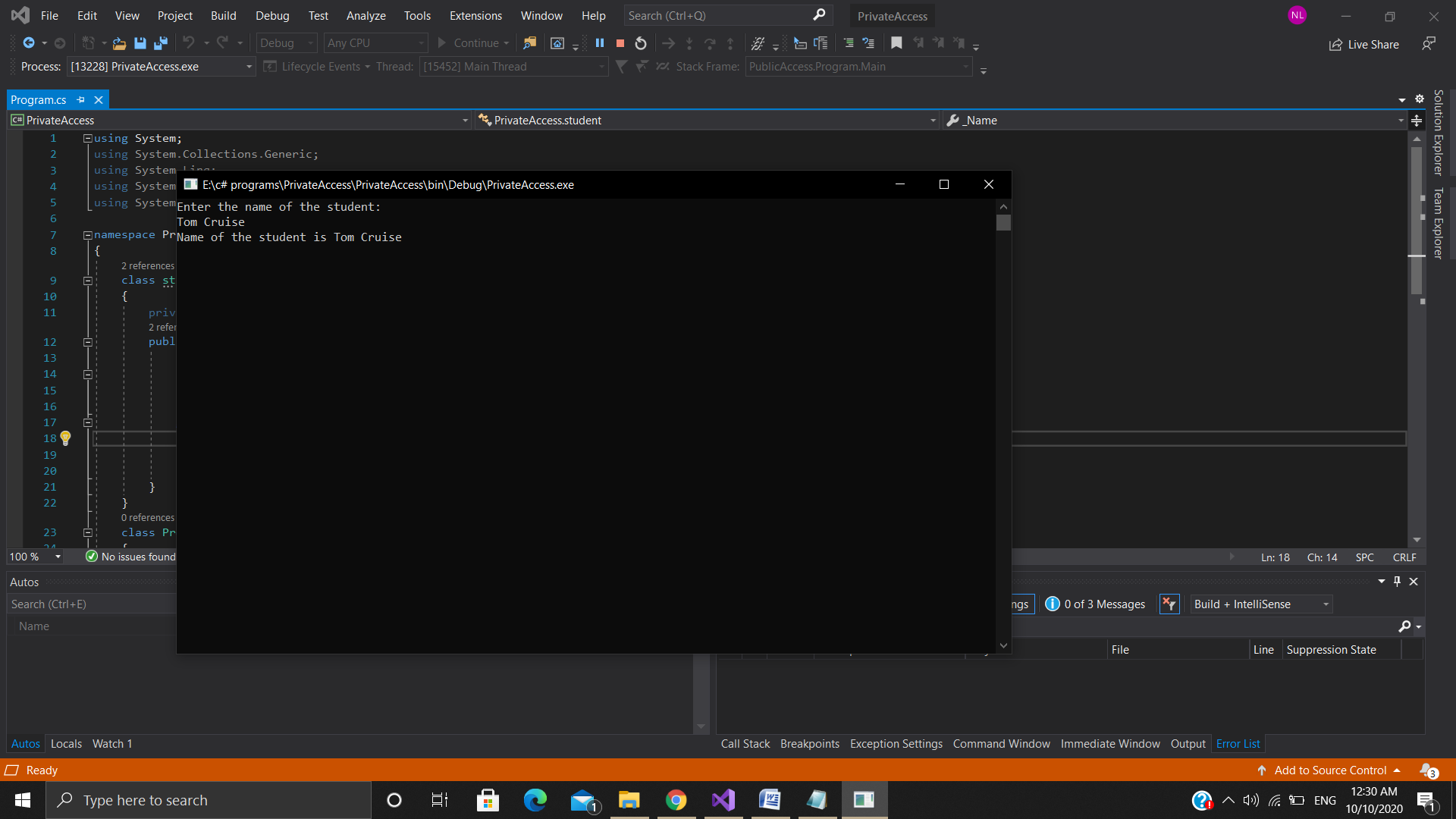
Console.ReadLine();

}

}

}

**Output**

****

1. **ProtectedAccessModifier**

using System;

namespace ProtectedAccess

{

class Base

{

protected void diaplay()

{

Console.WriteLine("This is Base class");

}

}

class Derived:Base

{

static void Main(string[] args)

{

Derived d1 = new Derived();

d1.diaplay();

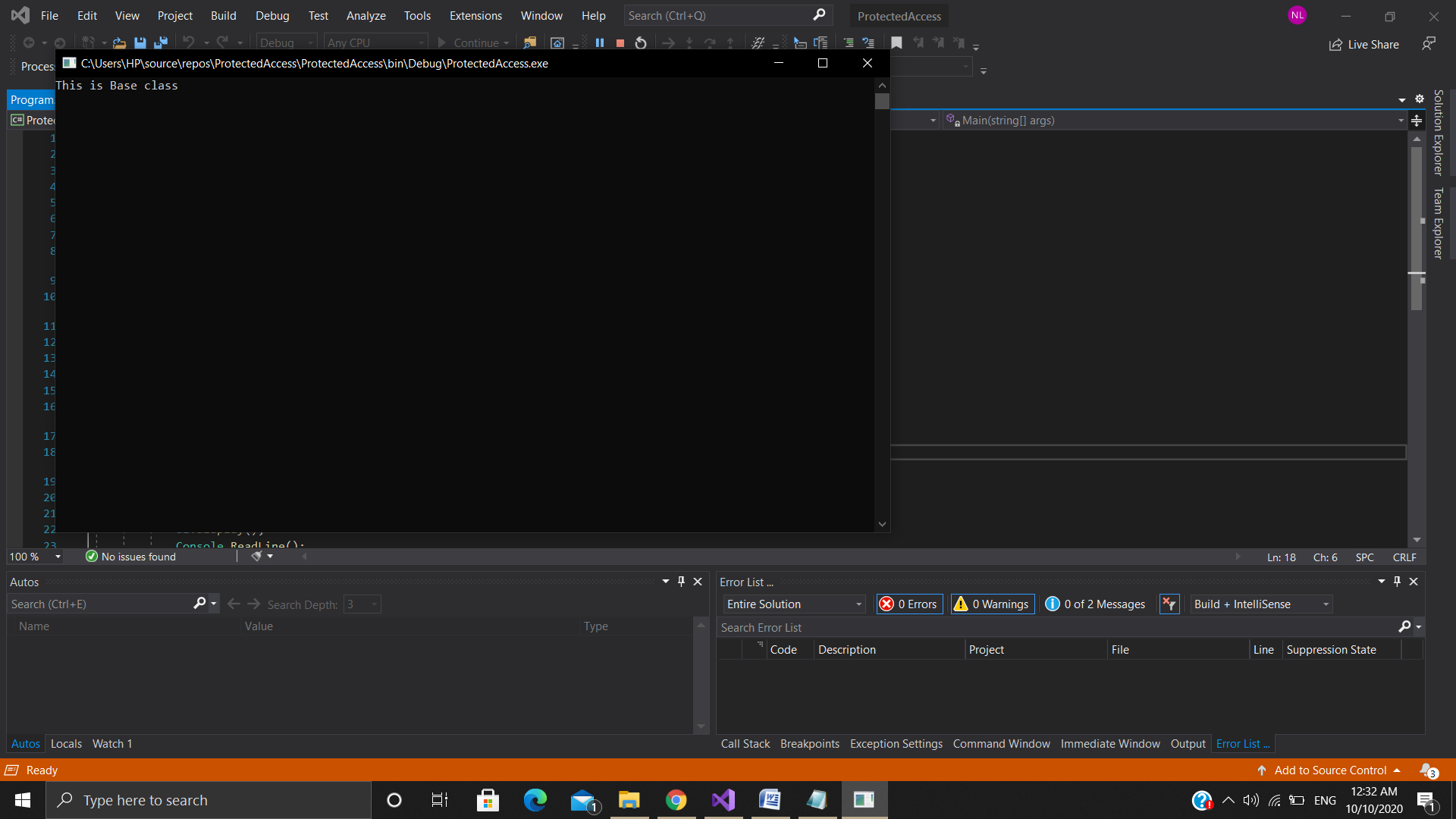
Console.ReadLine();

}

}

}

**Output**

****

1. **ArrayList**

using System;

using System.Collections;

namespace Collection1

{

class Program

{

static void Main(string[] args)

{

ArrayList mylist = new ArrayList();

mylist.Add("John");

mylist.Add(21);

mylist.Insert(1, "sinha");

int[] myarr = { 100, 200, 300 };

mylist.AddRange(myarr);

Console.WriteLine("Values of ArrayList");

foreach (object item in mylist)

{

Console.WriteLine(item);

}

Console.WriteLine("----------------------------------------------");

Console.WriteLine("Capacity of the List is : " + mylist.Capacity);

Console.WriteLine("Count of the List is : " + mylist.Count);

Console.WriteLine("List contains John: " + mylist.Contains("John"));

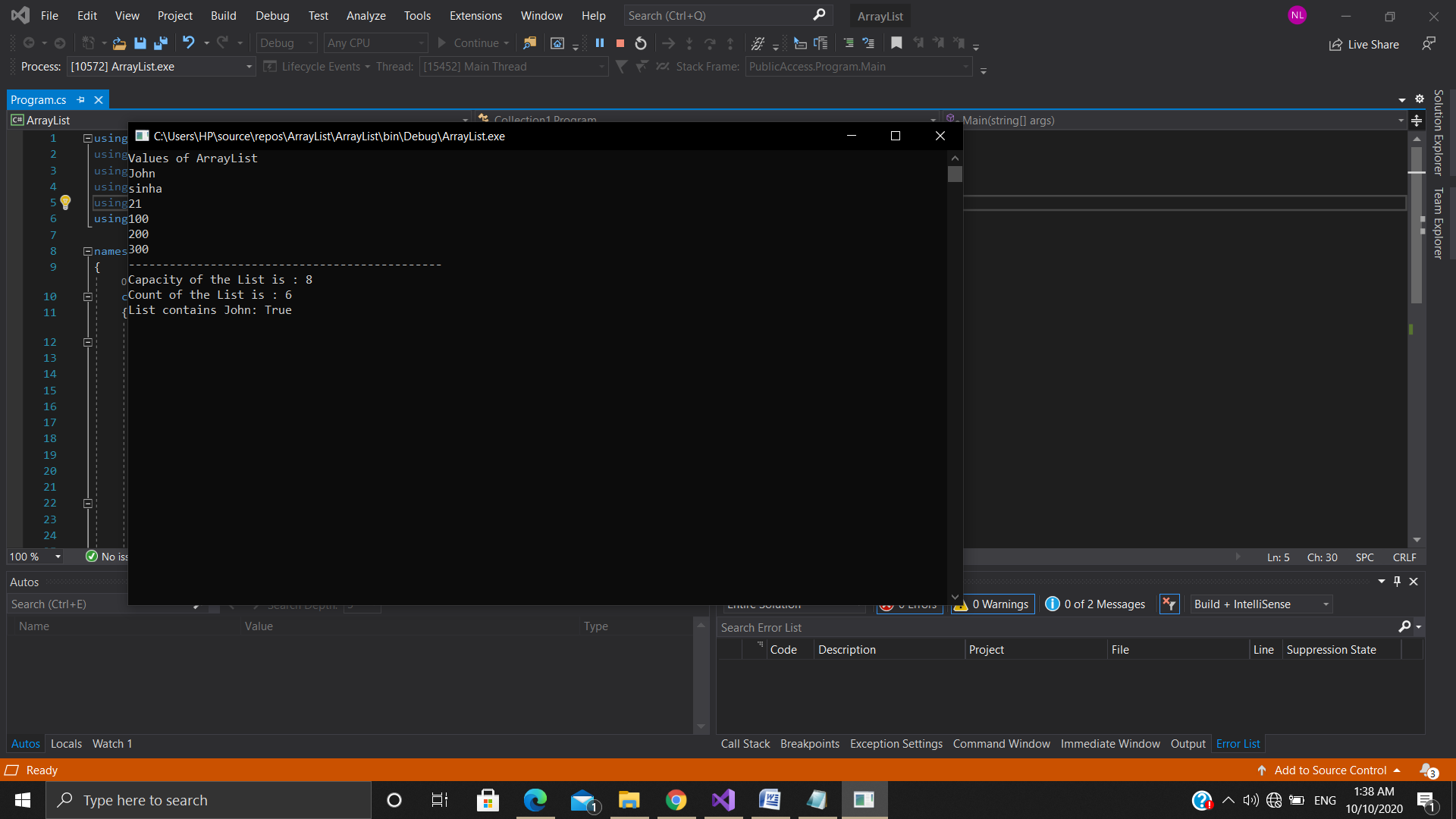
Console.ReadLine();

}

}

}

**Output**

****

1. **HashTable**

using System;

using System.Collections;

namespace Collection2

{

class Program

{

static void Main(string[] args)

{

Hashtable mytab = new Hashtable();

mytab.Add("Name", "Tom");

mytab.Add("Lastname", "Cruise");

mytab.Add("Age",32);

mytab.Add("Proffesion","Actor");

foreach (object item in mytab.Keys)

{

Console.WriteLine("Key is : " + item + " & Value is : "+ mytab[item]);

}

Console.WriteLine("Hashcode of Name key is: " + "Name".GetHashCode());

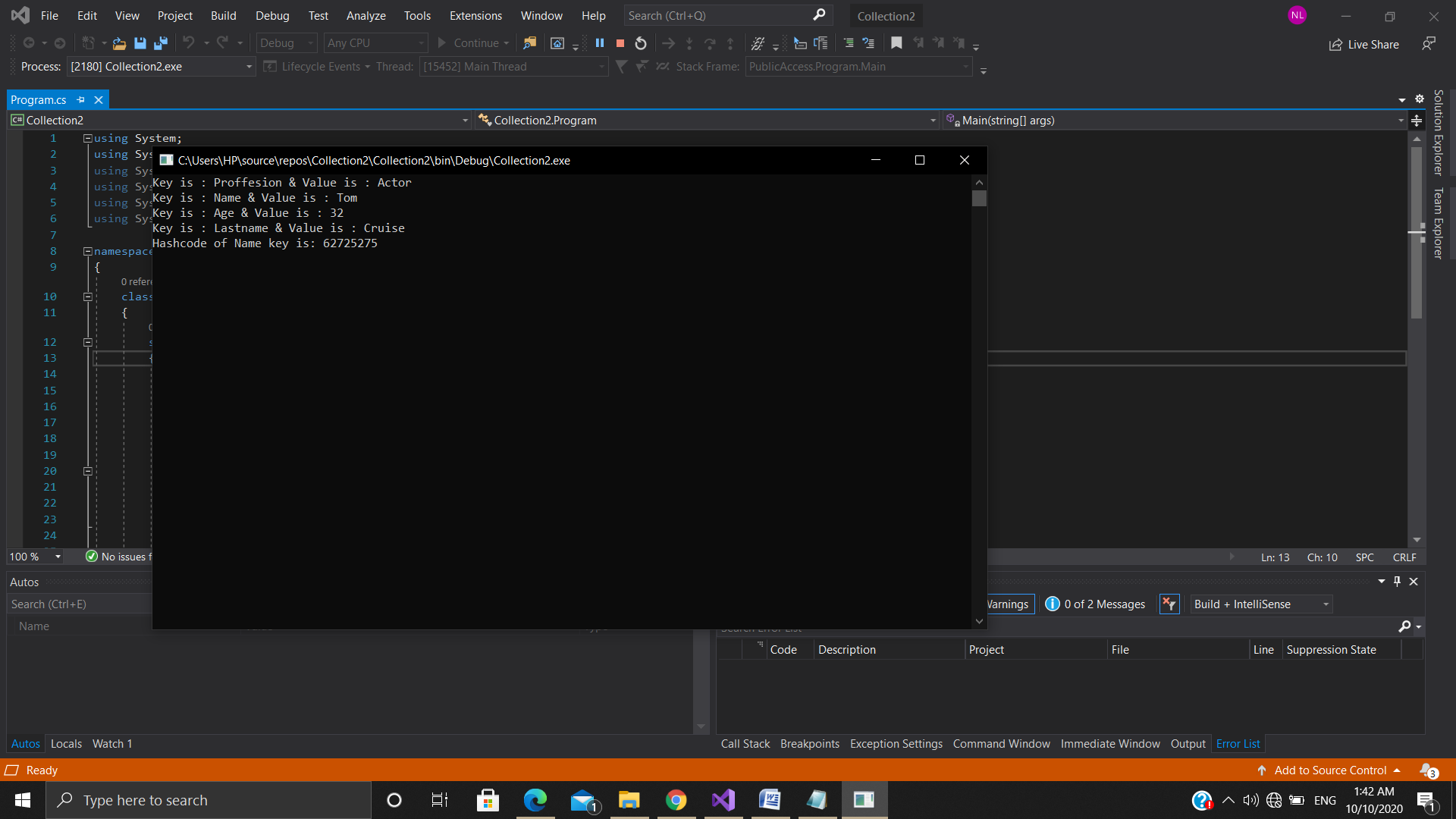
Console.ReadLine();

}

}

}

**Output**

****