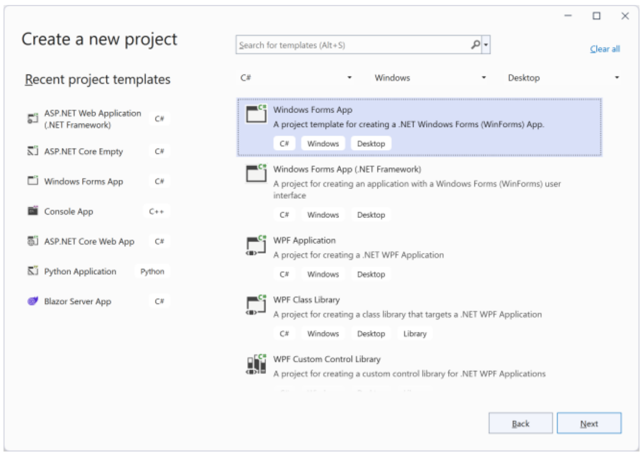
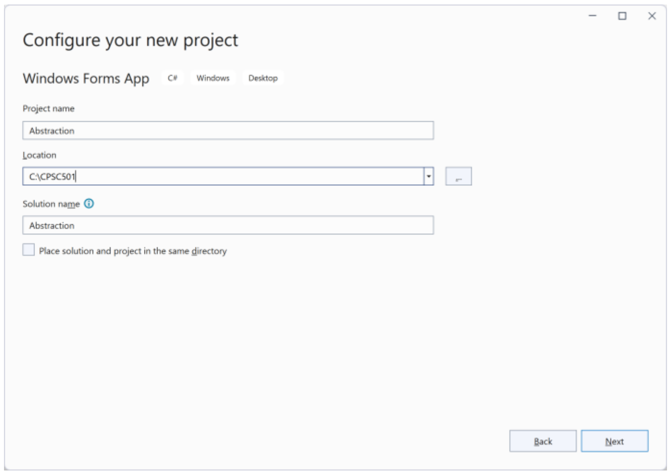
[redacted]

Abstraction – Custom and Standard Interfaces

One of the important concepts in OOP is abstraction where the key idea is to hide the implementation details of a computation from the end user. Abstraction mechanism in C# and Java is accomplished via the interface mechanism. An interface contains prototype of the function i.e., its name, the inputs and the type of output the function will return (but no public or private declaration before the function).

Create a new project called Abstraction as shown below.


Create a new project
Search for templates (Alt+S)
Clear all
Recent project templates
ASP.NET Web Application (.NET Framework) C#
ASP.NET Core Empty C#
Windows Forms App C#
Console App C++
ASP.NET Core Web App C#
Python Application Python
Blazor Server App C#
C#
Windows
Desktop
Windows Forms App
A project template for creating a .NET Windows Forms (WinForms) App.
C# Windows Desktop
Windows Forms App (.NET Framework)
A project for creating an application with a Windows Forms (WinForms) user interface
C# Windows Desktop
WPF Application
A project for creating a .NET WPF Application
C# Windows Desktop
WPF Class Library
A project for creating a class library that targets a .NET WPF Application
C# Windows Desktop Library
WPF Custom Control Library
A project for creating a custom control library for .NET WPF Applications
Back
Next



Configure your new project

Windows Forms App C# Windows Desktop

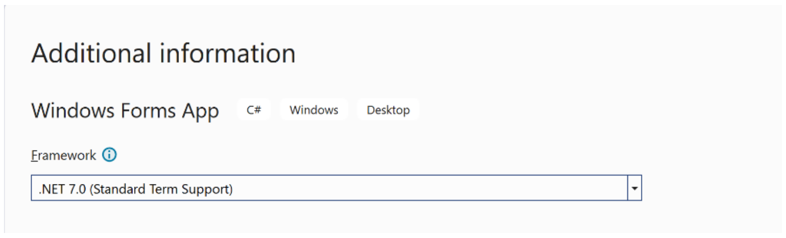
Project name: Abstraction

Location: C:\CPSC501

Solution name: Abstraction

☐ Place solution and project in the same directory

For example, if we wanted to create a class called AreaVolume that will provide methods such as ComputeCircleArea, and ComputeSphereVolume, these methods can be declared first in an interface.

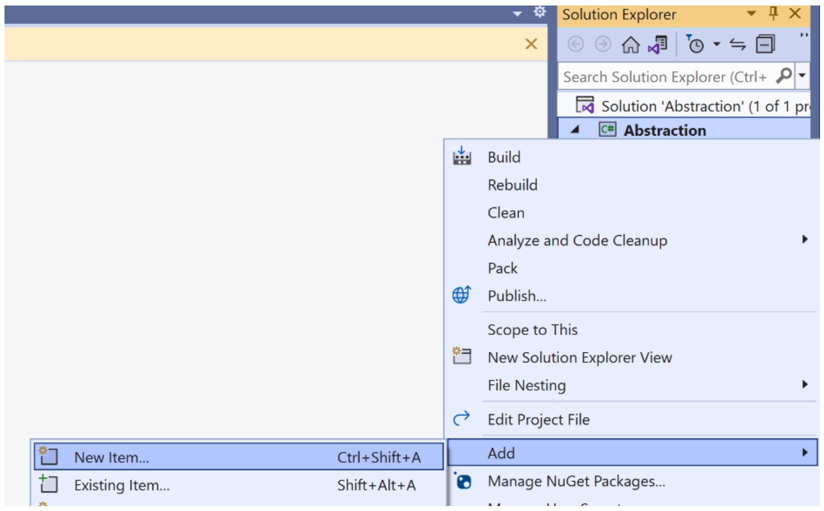


Additional information

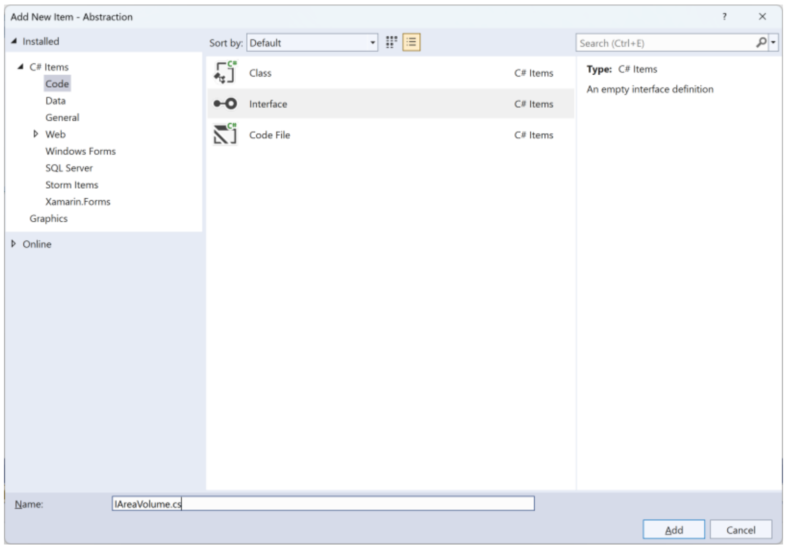
Windows Forms App C# Windows Desktop

Framework: .NET 7.0 (Standard Term Support)

Right click on the project name, and choose, “Add new item”.



Then select interface and name the interface IAreaVolume as:



Type the following code in IAreaVolume.cs.

```
internal interface IAreaVolume
{
    double ComputeCircleArea(double radius);
    double ComputeRectangleArea(double length, double width);
    double ComputeSphereVolume(double radius);
    double CylinderVolume(double radius, double length);
}
```

As you can see, the four functions declared above indicate what inputs they expect and what type of answer the function will produce, however, no code for the function is put in an interface. The idea is that the user of a class will examine only the interface to know how to call a particular function.

Add a class to the project called AreaVolume with the following code in it.
Note that after you have added the AreaVolume class, right click on the IAreaVolume and choose “Quick Actions-> Implement interface” so that Visual Studio types the skeleton code for all the four functions.

```
    internal class AreaVolume : IAreaVolume
    {
```

After Visual Studio has given you the skeleton code for the four functions as:

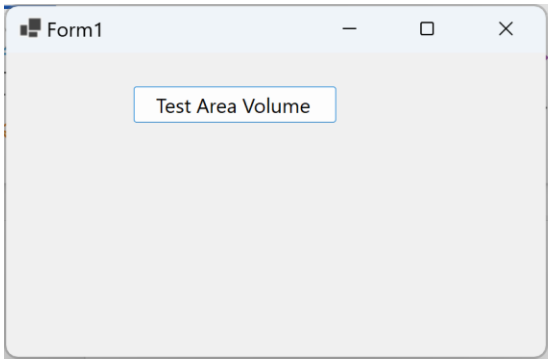
```
    internal class AreaVolume : IAreaVolume
    {
        public double ComputeCircleArea(double radius)
        {
            throw new NotImplementedException();
        }

        public double ComputeRectangleArea(double length, double width)
        {
            throw new NotImplementedException();
        }

        public double ComputeSphereVolume(double radius)
        {
            throw new NotImplementedException();
        }

        public double CylinderVolume(double radius, double length)
        {
            throw new NotImplementedException();
        }
    }
```

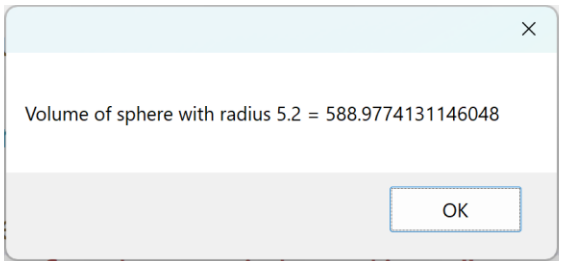
Modify the above code to have the proper computations as:

```
    internal class AreaVolume : IAreaVolume
    {
        public double ComputeCircleArea(double radius)
        {
            return Math.PI * radius * radius;
        }

        public double ComputeRectangleArea(double length, double width)
        {
            return length * width;
        }

        public double ComputeSphereVolume(double radius)
        {
            return 4/3.0 * Math.PI * Math.Pow(radius,3);
        }

        public double CylinderVolume(double radius, double length)
        {
            return 2 * Math.PI * radius * length;
        }
    }
```

Put a button the form with a name of btnTestAreaVolume and a text property of Test Area Volume. Then double click on the button and type the following code in the button click event handler.

```
        private void btnTestAreaVolume_Click(object sender, EventArgs e)
        {
            IAreaVolume iav = new AreaVolume();
            double radius = 5.2;
            double res = iav.ComputeSphereVolume(radius);
            MessageBox.Show("Volume of sphere with radius " + radius + " = " + res);
        }
```




Volume of sphere with radius 5.2 = 588.9774131146048

Note that interfaces act as the base type, and so an interface reference can point to the class object that implements the interface. Thus the following statement:

```
        IAreaVolume iav = new AreaVolume();
```

is perfectly OK according to the rules of OOP where a base class reference can point to a derived class object. Above is valid because the AreaVolume class implements the IAreaVolume interface as:

```
        internal class AreaVolume : IAreaVolume
```

The general guideline in OOP is that if you have a computation type of class, then define the interface for the computations needed first, and then write the class that will implement the interface. This way, we can just give the interface to the user of the class for them to know how to call the different methods in the class.

We also defined an interface called IProcessGrades in Assignment 1 before writing the code for the two functions.

```
    internal interface IProcessGrades
    {
        void ReadStudentData(string inputFileName);
        void ProcessAndWriteGrades(string outFileName);
    }
```

The above examples of creating interfaces is referred to as “custom interfaces” where we as a programmer decide what should be the name of the function in the interface and what inputs and outputs the different functions in the interface return.

There is another class of interfaces called “standard interfaces” that the language provides to us. The names of these interfaces and the inputs and outputs of the functions contained in these are fixed and cannot be changed.

Some of the popular standard interfaces that many programmers run into are:

ICloneable, IComparable, IComparer, IDispose, IEnumerator, IEnumerable, IAsyncResult etc.. Every good programmer should know the significance of these interfaces and the typical design patterns that are involved in coding these interfaces. We will take a look at each of these interfaces through simple examples.

ICloneable : The purpose of ICloneable is to provide a proper in memory copy of an existing object. Even though the base class *object* supports a *MemberwiseClone()* method, it only does a shallow copy of the object being copied.

Add a class called Student to the project. Type:

```
class Student : ICloneable
```

Then right click on ICloneable, and from quick actions, choose implement interface. Then modify the code in Student class to appear as:

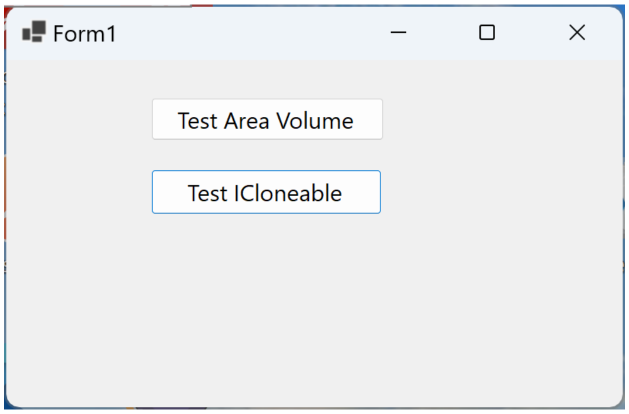
```
class Student : ICloneable
 {
     public Student() { }  // is needed because when we provide our
     // own constructor, the default constructor is no longer available
     public Student(string fnm, string lnm, int id)  // constructor
     {
         Fname = fnm;
         Lname = lnm;
         Id = id;
     }
     public Student(string fnm, string lnm, int id,
            int test1, int test2)  // overloaded constructor
     {
         Fname = fnm;
         Lname = lnm;
         Id = id;
         TestScores[0] = test1;
         TestScores[1] = test2;
     }

     public int[] TestScores { get; set; } = new int[2];
     protected string Fname { get; set; }
     protected string Lname { get; set; }
     public int Id { get; set; }

     public override string ToString()
     {
         return "Fname=" + Fname +
             " Lname=" + Lname + " ID=" + Id.ToString() +
             " Test1=" + TestScores[0].ToString() + " Test2=" +
             TestScores[1].ToString();
     }

     public virtual string GetGrade()
     {
         string grade = "";
```

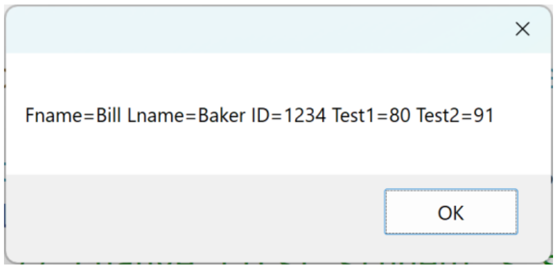
```
            double avg = 0.4 * TestScores[0] +
                0.6 * TestScores[1];
            if (avg > 90)
                grade = "A";
            else if (avg > 85)
                grade = "A-";
            else if (avg > 80)
                grade = "B+";
            else
                grade = "B";
            return grade;
        }

        public object Clone()
        {
            Student snew = new Student();
            snew = (Student)this.MemberwiseClone();
            return snew;
        }
    }
```

Add a button to the form with a name of btnTestICloneable and a text property of ICloneable. Double click on the button and type the following code in the event handler.

```
        private void btnTestICloneable_Click(object sender, EventArgs e)
        {
            Student s1 = new Student("Bill", "Baker", 1234, 87, 91);
            Student s2 = (Student)s1.Clone();
            s1.TestScores[0] = 80;  // change first student's score
            MessageBox.Show(s2.ToString());  // display second student
        }
```

When you run the project and click on the above button, you will notice that even though the code was changing the text score for the first student, the data for the second student also got changed.



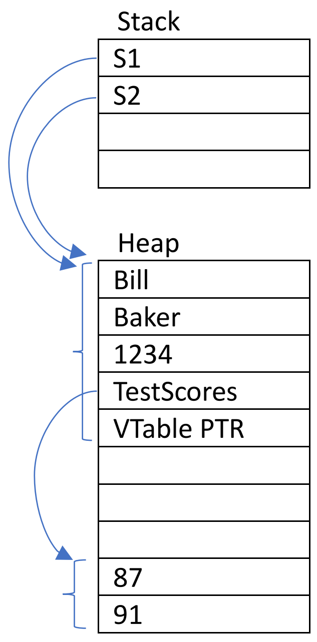


Fname=Bill Lname=Baker ID=1234 Test1=80 Test2=91

This is because arrays in C# and Java are reference types. For a reference type e.g., a class object of Student type, if we have the code:

```
        Student s1 = new Student("Bill", "Baker", 1234, 87, 91);
        Student s2 = s1
```

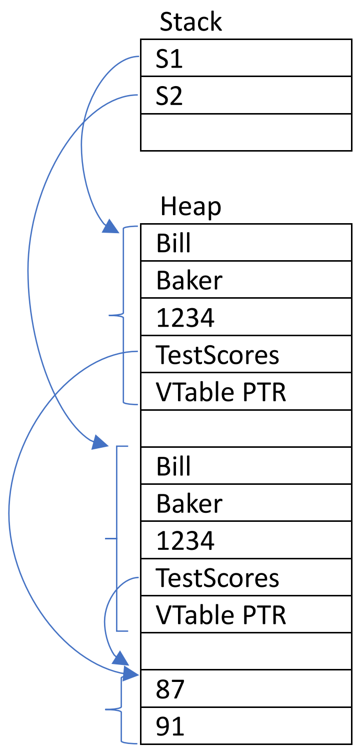
Then s2 is pointing to the exact same memory as s1 (see diagram below). If we modify any field of s1, s2 also gets the modification because it is pointing to the same memory.



If we execute:

```
	Student s1 = new Student("Bill", "Baker", 1234, 87, 91);
	Student s2 = (Student)s1.Clone();// implemented as MemberwiseClone
	s1.TestScores[0] = 80;
```

Then s2 gets new memory for all the fields except for the TestScores array because arrays in C# and Java are reference types, so the s2's TestScores is pointing to the same memory as s1's TestScores.



The important concept to note is that when we assign one reference type to another, then it points to the same memory, however when we do the same for a value type (such as integers or doubles), a new memory is created e.g.,

    int a = 5;
    int b = a;

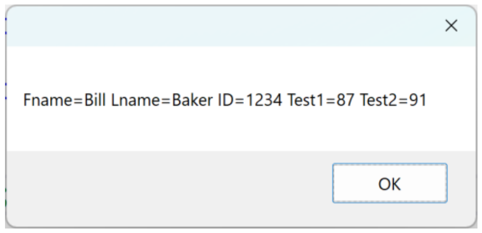
a	5
b	5

Note that in C# and Java, class objects are reference types, also arrays are reference types. Reference types are created on the heap, and when we assign one reference type to another, it points to the same memory as we depicted in the previous page.

The correct solution in copying one object to another is to have totally independent memories for each object. We cannot solely rely on the MemberwiseClone which does a shallow copy. For a proper copy such that each reference type inside an object is correctly copied, we have to call the Clone method on the field of the class object that is being cloned e.g., the correct implementation of ICloneable in previous example is to modify the clone method in Student class to appear as:

```
public object Clone()
 {
     Student snew = new Student();
     snew = (Student)this.MemberwiseClone();
     snew.TestScores = (int[])this.TestScores.Clone();
     return snew;
 }
```

Now the output will be better when you run the program and click on the button i.e., modifying the test score for the first student does not modify the test score of second student after the cloning is done.



```
Fname=Bill Lname=Baker ID=1234 Test1=87 Test2=91
```

What if the Student class contains a field of another class called Address. Add a class called Address to the project with the following code in it.

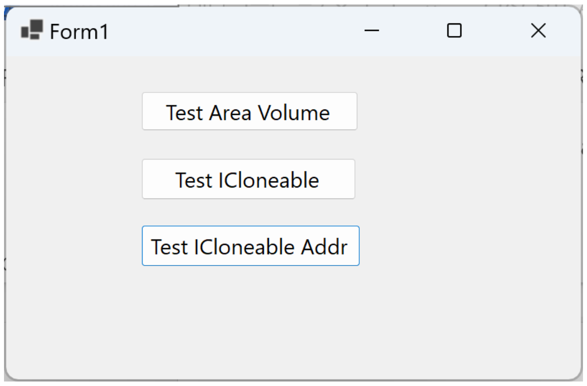
```
internal class Address
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

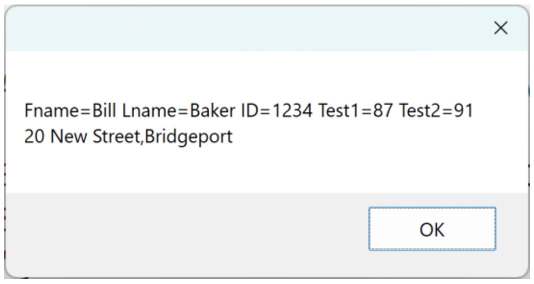
Modify the student class to include a field of Address class as (additions are shown in bold):

```
namespace Abstraction
{
    class Student : ICloneable
    {
        ….
        public int[] TestScores { get; set; } = new int[2];
        protected string Fname { get; set; }
        protected string Lname { get; set; }
        public int Id { get; set; }
        public Address Addr { get; set; } = new Address();

        public override string ToString()
        {
            return "Fname=" + Fname +
                " Lname=" + Lname + " ID=" + Id.ToString() +
                " Test1=" + TestScores[0].ToString() + " Test2=" +
                TestScores[1].ToString() +"\n" + Addr.Street +"," + Addr.City;
        }

….

        public object Clone()
        {
            Student snew = new Student();
            snew = (Student)this.MemberwiseClone();
            snew.TestScores = (int[])this.TestScores.Clone();
            return snew;
        }
    }

}
```

Add a button to the form with a name of btnTestICloneableAddr with a text property of Test ICloneable Addr. Then double click on the button and type the following code in its event handler.

```
        private void btnTestICloneableAddr_Click(object sender, EventArgs e)
        {
            Student s1 = new Student("Bill", "Baker", 1234, 87, 91);
            s1.Addr.Street = "50 Main Street";
            s1.Addr.City = "Bridgeport";
            Student s2 = (Student)s1.Clone();
            s1.TestScores[0] = 80;  // change first student's score
            s1.Addr.Street = "20 New Street"; // change first stydent's address
            MessageBox.Show(s2.ToString());  // display second student
        }
```





```
Fname=Bill Lname=Baker ID=1234 Test1=87 Test2=91
20 New Street,Bridgeport
```

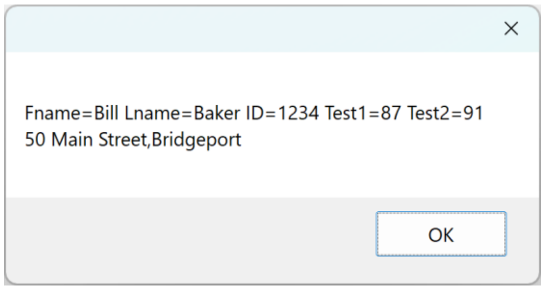
As you can see, after we clone s1, and change the address of s1, it is reflected in s2 which is wrong. Modifying s1 after cloning should not change the values in s2. Again this is because the address is a reference type field and Memberwise clone will end up pointing to the same memory for both objects. We need to further clone the Addr field of Student class for cloning to have two separate memories for the two objects. However, we cannot call clone on the Addr field unless we implement the ICloneable on the Address class.
Modify the Address class to contain ICloneable implementation as:

```
internal class Address : ICloneable
{
    public string Street { get; set; }
    public string City { get; set; }

    public object Clone()
    {
        return this.MemberwiseClone(); // OK as all fields are either
    }   // value or string type (which behaves as a value type)
}
```

Modify the Clone method in Student class to further call Clone on the Addr field.

```
    public object Clone()
    {
        Student snew = new Student();
        snew = (Student)this.MemberwiseClone();
        snew.TestScores = (int[])this.TestScores.Clone();
        snew.Addr = (Address)this.Addr.Clone();
        return snew;
    }
```

Now if you run the program, the cloning of s1 to s2 will be proper i.e., modifying either the testscore or the Addr in s1 after cloning will not affect s2.



```
Fname=Bill Lname=Baker ID=1234 Test1=87 Test2=91
50 Main Street,Bridgeport
```

To summarize, ICloneable interface provides a Clone method whose job is to make a proper in memory copy of an existing object. If the object contains further reference types such as arrays or class objects, then we need to call Clone on each of these in the implementation of ICloneable. If the field is of another class type, then we need to implement ICloneable in that class before calling Clone on the field.

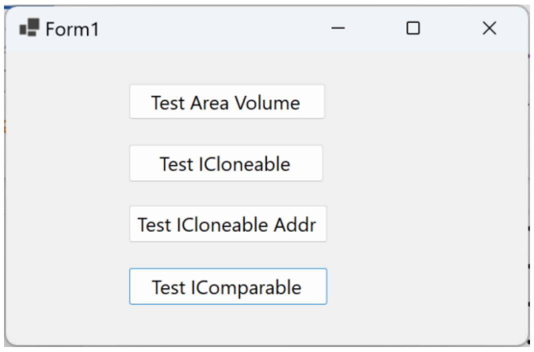
IComparable Interface: When a collection is being sorted, the parameterless sort method invokes the *IComparable.CompareTo* method on the objects in the collection to accomplish sorting. We can implement the IComparable interface to indicate which field in the class will be compared for sorting purposes. Thus *IComparable* allows us to specify a field of the class to sort collection of this class's objects on.

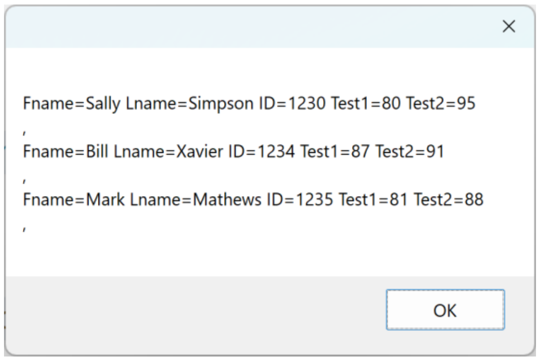
Implement the IComparable interface in the Student class as shown below.

```
class Student : ICloneable, IComparable<Student>
{
  ..
  ..
   public int CompareTo(Student? other)
   {
       int retVal = 0;
       if (other != null)
           retVal = this.Id.CompareTo(other.Id);
       return retVal;
   }
}
```

Add a button to the Form with a name of "btnTestIComparable" and a text property of "Test IComparable". Write the following code in the button's handler.

```
private void btnTestIComparable_Click(object sender, EventArgs e)
{
    List<Student> STList = new List<Student>();
    Student s1 = new Student("Bill", "Xavier", 1234, 87, 91);
    STList.Add(s1);
    Student s2 = new Student("Mark", "Mathews", 1235, 81, 88);
    STList.Add(s2);
    Student s3 = new Student("Sally", "Simpson", 1230, 80, 95);
    STList.Add(s3);
    STList.Sort(); // triggers IComparer on Student class
    string out1 = "";
    foreach (Student st in STList)
        out1 += st.ToString() + "\n";
    MessageBox.Show(out1);
}
```





```
Fname=Sally Lname=Simpson ID=1230 Test1=80 Test2=95
,
Fname=Bill Lname=Xavier ID=1234 Test1=87 Test2=91
,
Fname=Mark Lname=Mathews ID=1235 Test1=81 Test2=88
,
```

You can see that the Student list is correctly sorted via the Id field.

<u>Your Enhancement</u>:

- Modify the CompareTo function in the Student class so that it sorts on the Lname field.

- Modify the Student class so that the grades are output according to the highest grade. A grades should appear first.

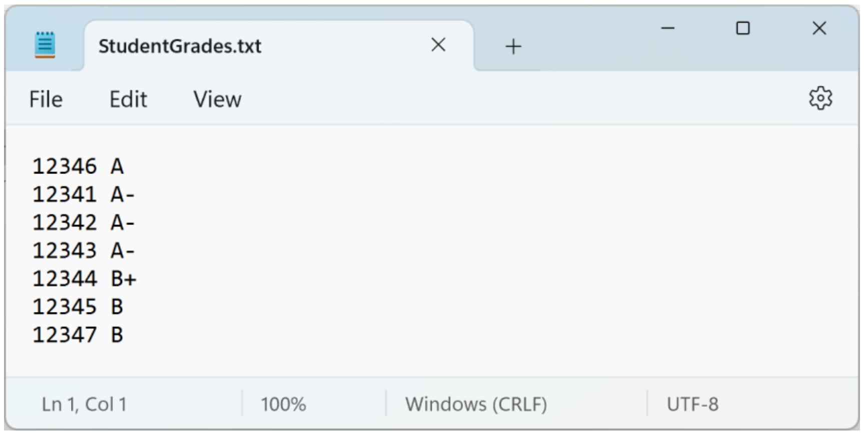
Modify the Student class in GradingApp to appear as:

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

namespace GradingApp
{
    abstract class Student : IComparable<Student>
    {
        public Student(int id, string fname, string lname, int test1, int test2)
        {  // constructor
            this.ID = id;
            this.FirstName = fname;
            this.LastName = lname;
            this.Test1 = test1;
            this.Test2 = test2;
        }
        public int ID { get; set; }
        public string FirstName { get; set; } = string.Empty;
        public string LastName { get; set; } = String.Empty;
        public int Test1 { get; set; }
        public int Test2 { get; set; }

        public int CompareTo(Student? other)
        {
            int ret = 0;
            if (other != null)
                return -
1*GetGradePoints(this.ComputeGrade()).CompareTo(GetGradePoints(other.ComputeGrade())
);
            return ret;
        }

        double GetGradePoints(string grade)
        {
            double points = 0;
            switch (grade)
            {
                case "A": points = 4.0;
                    break;
                case "A-":
                    points = 3.7;
                    break;
                case "B+":
                    points = 3.3;
                    break;
                case "B":
                    points = 3.0;
                    break;
                case "B-":
                    points = 2.7;
                    break;
                case "C+":
                    points = 2.3;
                    break;
                case "C":
                    points = 2.0;
                    break;
                default:
                    points = 1.0;
                    break;
            }
            return points;
        }

        public abstract string ComputeGrade(); // derived class will provide the
implemntation
    }
}
```

Modify the ProcessAndWriteGrades function in ProcessGrades to appear as (line to be added is shown in bold):

```
public void ProcessAndWriteGrades(string outFileName)
        {
            StreamWriter sw = new StreamWriter(outFileName);
            try
            {
                STList.Sort();
                foreach (Student st in STList)
                {
                    string grade = st.ComputeGrade();  // polymorphism, correct
ComputeGrade
                        // will be called depending upon the type of student in st
                    sw.WriteLine(st.ID + "\t" + grade);
                }
                sw.Close();
            }
            catch
            {
                throw;
            }
            finally
            {
                sw.Close();
            }
        }
```

If you run the project, the output file will be correctly sorted by the highest to lowest grade.



```
12346 A
12341 A-
12342 A-
12343 A-
12344 B+
12345 B
12347 B
```