CPSC 501 – Fall 2023 - Assignment #1

Problem #1: Recreate the example in C# in Visual Studio described in Lecture 2 (see the kiwi web site for CPSC 501 for recording of lecture 2).

Problem #2: Simple Grading Application using OOP principles in C#.

We will be reading the student information from a tab delimited text file (each field in the text file will be separated by a tab). Then assign grades to each student. The grading formula for UNDERGRAD, GRAD and PHD students will be different. Each student will have the following info:

ID FirstName LastName Major DegreeLevel Test1 Test2 Thesis/Dissertation Advisor

The degree level can be GRAD or UNDERGRAD or PHD. For Grad students, there will be a thesis title, for PhD students, there will be dissertation title.

Here is an example of data for UNDERGRAD student:

1234 Bill Baker UNDERGRAD 85 91

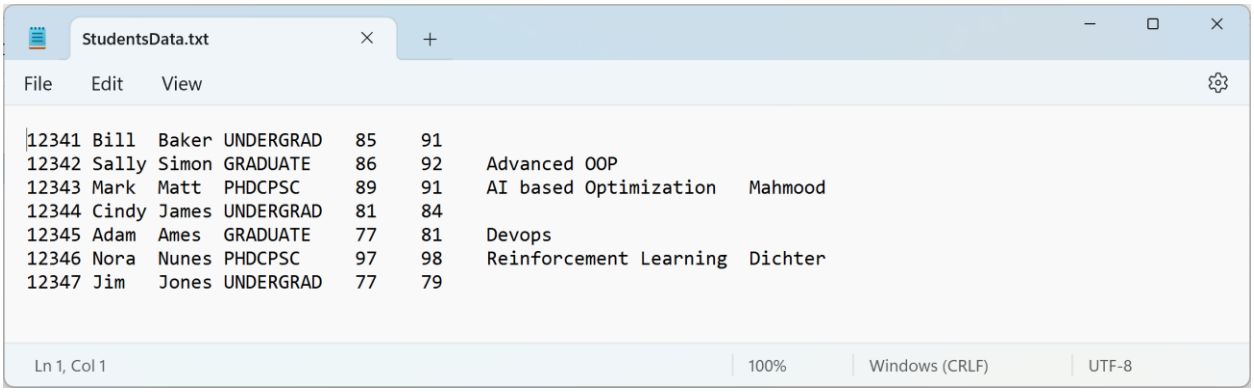
Here is an example of data for GRAD student:

1234 John Jacobs GRADUATE 88 90 Cloud Scalability

Here is an example of data for PhD student:

1234 Sally Simpson PHDCPSC 91 93 AI Diffusion Models Mahmood

Using Notepad, create a tab delimited file with the following data and save it in a folder called data under the C:/CPSC501 folder. Name the file StudentsData.txt



In an OOP based design, one of the first things we decide in creating a software application is the classes and the fields and methods that we will need in each class. Class provides the fundamental encapsulation mechanism where the related data and fields are grouped together in a virtual box.

Since we have three types of students, we will be creating an UnderGradStudent, GradStudent, and PhDStudent classes. The fields and methods in each of these classes will be as:

class UnderGradStudent:
- Fields:
  - ID
  - FirstName
  - LastName
  - Test1
  - Test2
- Methods:
  - ComputeGrade

class GradStudent:
- Fields:
  - ID
  - FirstName
  - LastName
  - Test1
  - Test2
  - **Thesis**
- Methods:
  - **ComputeGrade**

class PhdStudent:
- Fields:
  - ID
  - FirstName
  - LastName
  - Test1
  - Test2
  - **Dissertation**
  - **Advisor**
- Methods:
  - **ComputeGrade**

The next thing we identify in an OOP implementation is “are there any common things between the different classes that we will need?”. If so, we move the common things (both fields and methods) in a base or the parent class. The different classes then inherit from the base class and add the additional fields or methods that are needed. Recall that in OOP, a derived class inherits all fields and methods from the base class. The derived class can add a field or modify an inherited method by overriding it, but it cannot remove fields or methods from the base class.

Thus our class hierarchy may appear as:

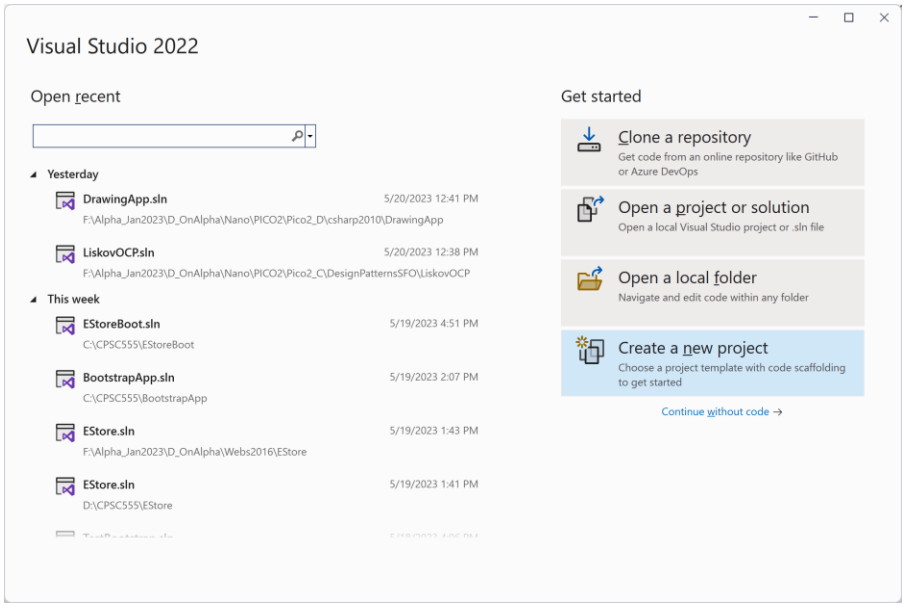
class Student: // base class
    Fields:
        ID
        FirstName
        LastName
        Test1
        Test2
    Methods:
        ComputeGrade // virtual or abstract so that each derived class can modify this function

class UnderGradStudent: Student // derived class
    Fields: // all fields are inherited from base class
    Methods:
        ComputeGrade // override and provide code for computing grade for UndeGraduate student

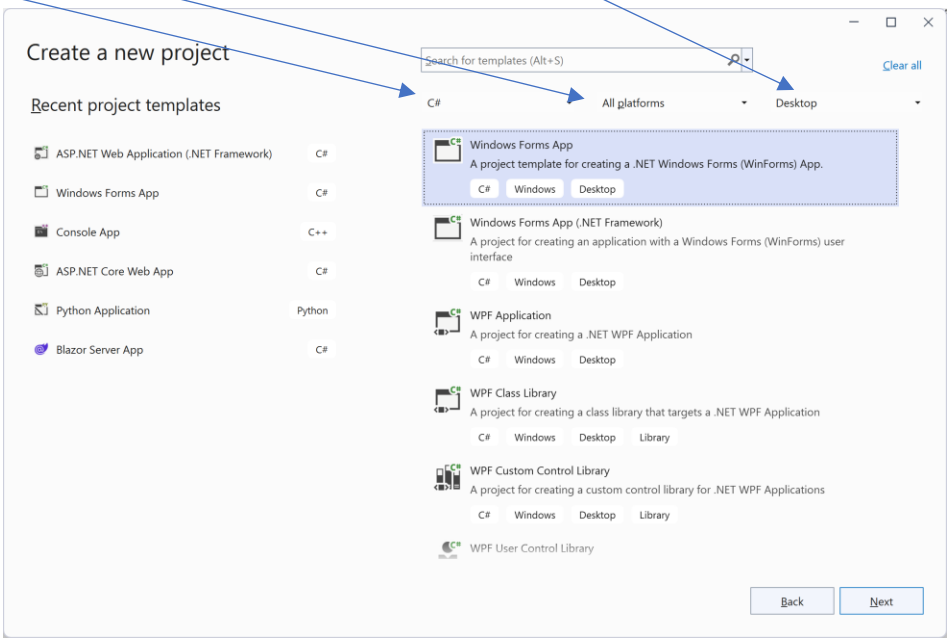
class GraduateStudent: Student // derived class
    Fields:
        Thesis   // other fields are inherited from Student
    Methods:
        ComputeGrade // override and provide code for computing grade for Graduate Student

class PhdStudent: Student // derived class
    Fields:
        Thesis
        Advisor   // other fields are inherited from Student
    Methods:
        ComputeGrade // override and provide code for computing grade for Phd Student

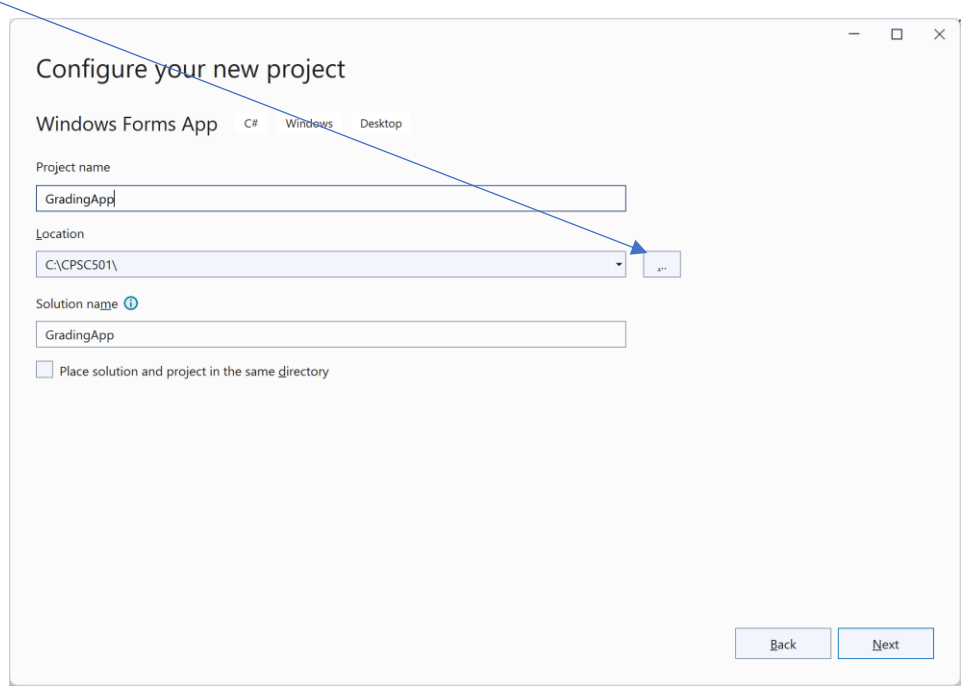
Create a new project in Visual Studio called GradingApp as shown below.

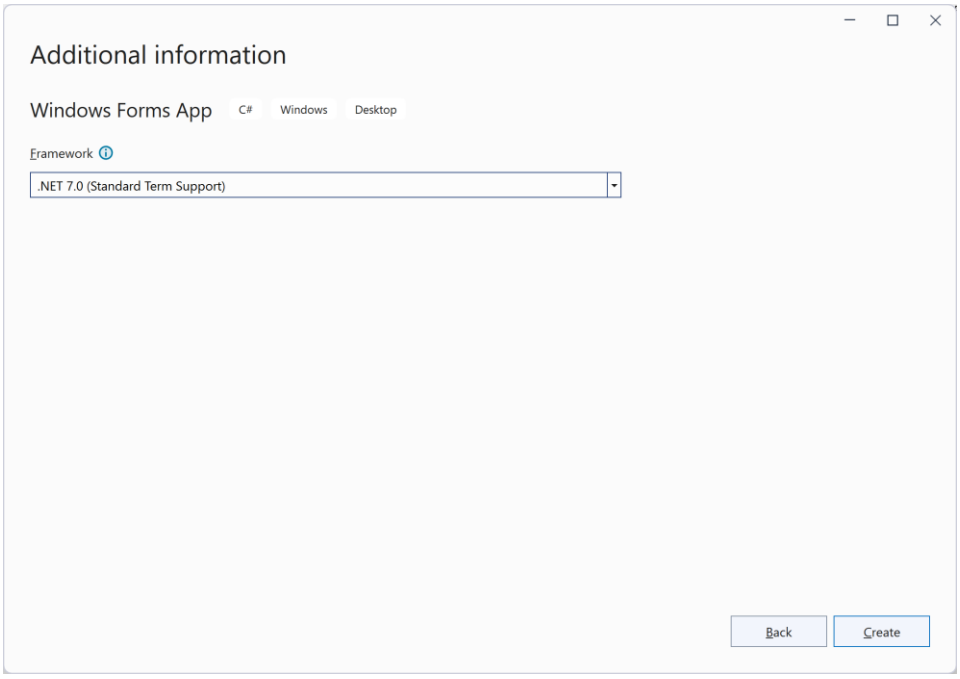


Select C#, All Platforms and Windows Forms Application, Desktop as shown below.



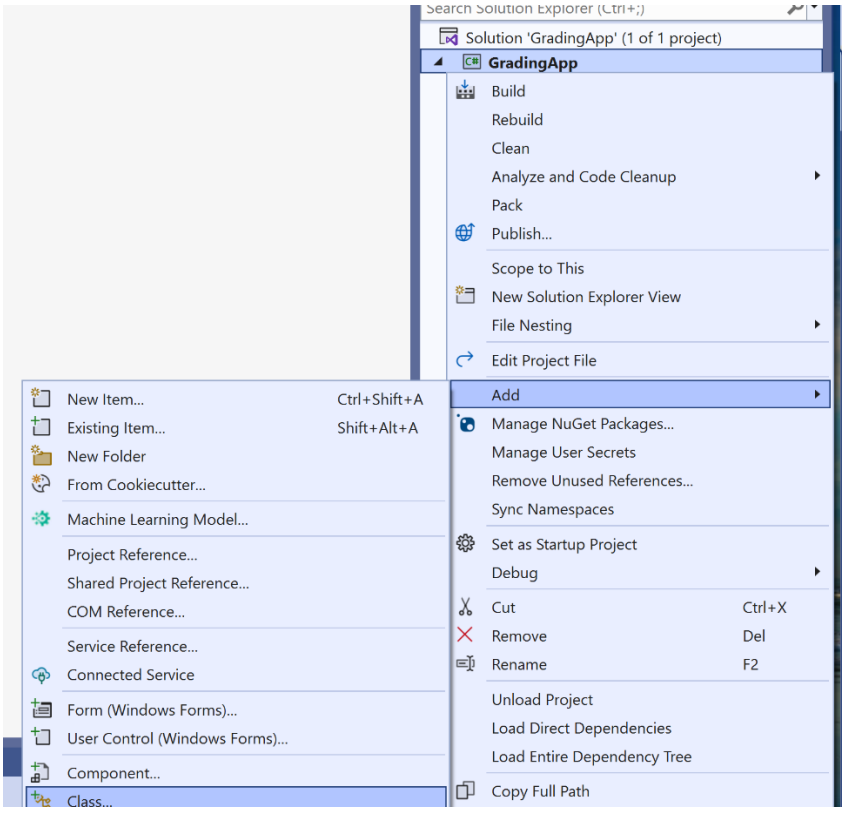
Once you click next, name the prok=ject GradingApp and browse to the C:/CPSC501 folder by clicking on the button (create CPSC501 folder if it does not already exist).

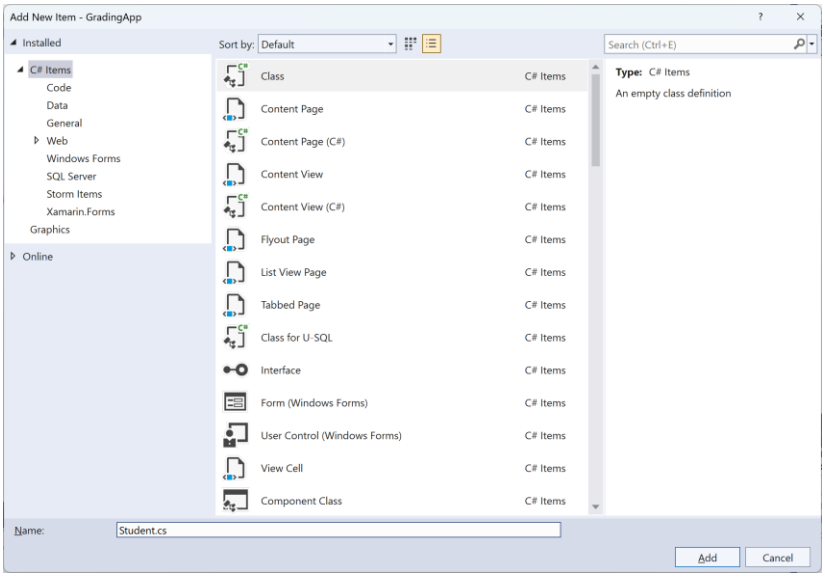




Then click Create.

Right click on name of the project name and click “Add Class”, name the class Student as shown below.





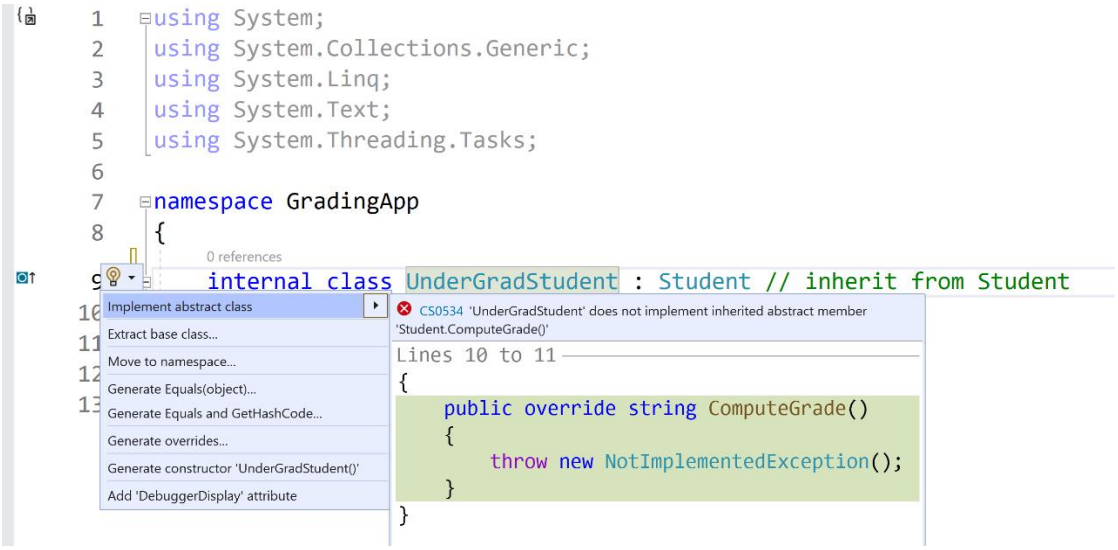
Click on the Create button and type the following code in it. You are typing the code shown in bold.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace GradingApp
{
    abstract class Student // base or parent class for different student types
    {
        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 abstract string ComputeGrade(); // derived class will provide the code
    }
```

Note that the Student class is marked as “abstract” which means that we cannot create an object of this class but can use it in inheritance to reuse code. A class has to be marked abstract if it contains one method that is abstract. In this case, the ComputeGrade method is marked as abstract because we do not know the code for it as each type of student will have a different formula for assigning the grade.

Similarly, add another class called UnderGradStudent to the project with the following code in it.

Once you type UnderGradStudent : Student, right click on the UnderGradStudent and choose “implement abstract method” as show below.



Then modify the code in the UnderGradStudent class to appear as:

```
internal class UnderGradStudent : Student // inherit from Student
    {
        public UnderGradStudent(int id, string fname, string lname, int test1, int
test2)
        :base(id,fname,lname,test1,test2){  // delegate initialization to base class
constructor
        }
        // ID, FirstName, LastName, Test1, Test2 are inherited from Student
        public override string ComputeGrade()
        {
            double avg = 0.4 * Test1 + 0.6 * Test2;
            string grade = "";
            if (avg > 90)
                grade = "A";
            else if (avg > 85)
                grade = "A-";
            else if (avg > 80)
                grade = "B+";
            else if (avg > 70)
                grade = "B";
            else
                grade = "C";
            return grade;
        }
    }
```

Notice that ComputeGrade has been marked as “override” in above code which means that we are inheriting the base class ComputeGrade method but for UnderGradStudent we want to change the inherited code. In OOP, the base class has to give us permission to override the code by declaring the method as abstract (if there is no code in it), or virtual (if there is code in it) in the base class.

Add another class called GradStudent with the following code in it.

```
internal class GradStudent : Student
    {
        public GradStudent(int id, string fname, string lname, int test1, int test2,
string thesis) : base(id, fname, lname, test1, test2)
        {  // delegate init to base class constructor for first 5 fields
            this.Thesis= thesis; // initialization of extra field
        }
        // ID, FirstName, LastName, Test1, Test2 are inherited from Student
        public string Thesis { get; set; } // added extra field in GradStudent class
        public override string ComputeGrade()
        {
            double avg = 0.4 * Test1 + 0.6 * Test2;
            string grade = "";
            if (avg > 92)  // more than 92 is an A for a GradStudent
                grade = "A";
            else if (avg > 87)
                grade = "A-";
            else if (avg > 83)
                grade = "B+";
            else if (avg > 75)
                grade = "B";
            else
                grade = "C";
            return grade;
        }
    }
```

Add another class called PhdStudent with the following code in it.

```
internal class PhdStudent : Student
    {
        public PhdStudent(int id, string fname, string lname, int test1, int test2,
string dissertation, string advisor)
        : base(id, fname, lname, test1, test2)
        {  // delegate initialization to base class constructor for first 5 fields
            this.Dissertation = dissertation; // initialization of extra field
            this.Advisor = advisor; // initialization of extra field
        }
        // ID, FirstName, LastName, Test1, Test2 are inherited from Student
        public string Dissertation { get; set; } // added field in PhdStudent class
        public string Advisor { get; set; } // added extra field in PhdStudent class
        public override string ComputeGrade()
        {
            double avg = 0.4 * Test1 + 0.6 * Test2;
            string grade = "";
            if (avg > 95)  // more than 95 is an A for a PhdStudent
                grade = "A";
            else if (avg > 90)
                grade = "A-";
            else if (avg > 87)
                grade = "B+";
            else if (avg > 80)
                grade = "B";
            else
                grade = "C";
            return grade;
        }
    }
```

Since our goal in this app is to read student data from a file and assign grades to students, and store the ID and the grade for each student in a grades file, we will create one more class to encapsulate the reading of student data and writing of the grades to a file. If we name this class ProcessGrades, then one possibility for this class is to have the following fields and mthods.

class ProcessGrades:

Fields:

List of student objects

Methods:

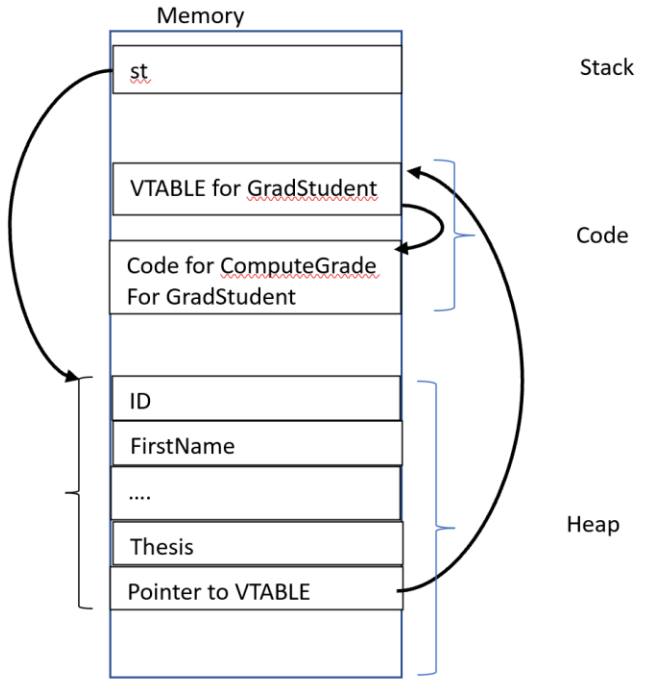
ReadStudentData(string studentdatafilename)

AssignAndWriteGradesToFile(string outputfilename)

In good OOP, we have the concepts of Encapsulation, Inheritance, Polymorphism and Abstraction. When we created the Student base class and the derived classes of UnderGradStudent, GradStudent and PhdStudent, we saw how an individual class is used to encapsulate related fields and methods, and how base and derived classes implement inheritance so that less code is needed in each of the student type classes. Further, when we declared the ComputeGrade as an abstract method in Student class and used override for it in each of the different students types, it helped us achieve Polymorphism. This basically means that if we use the Student class (which is our base class) reference to point to a derived class object, and call the ComputeGrade from the base class reference, it will call the correct ComputeGrade based on the type of object that was created, by traversing the pointer to the VTABLE (which is table of function pointers maintained in the code area for each class e.g.,

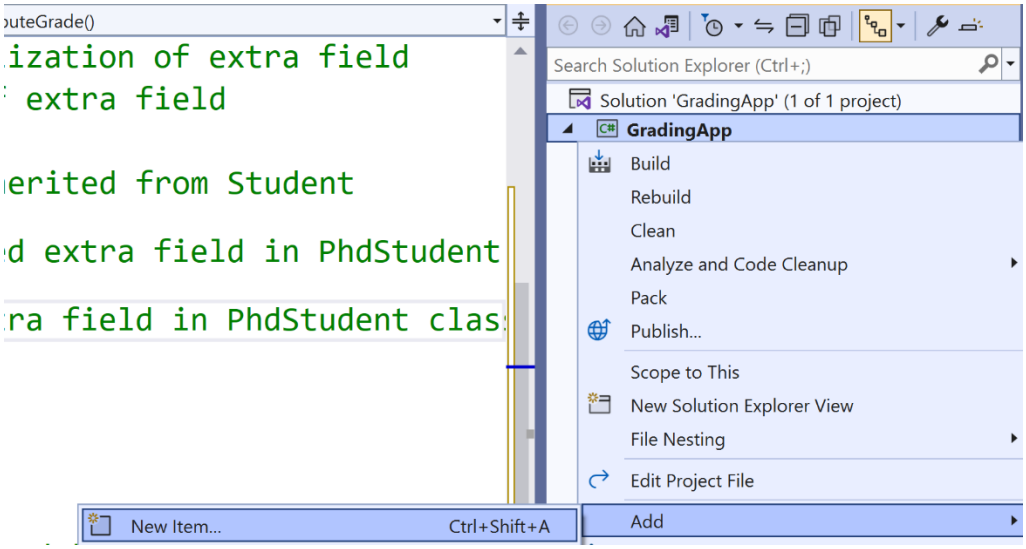
```
Student st = new GradStudent();
String grade = st.ComputeGrade();
```

In the above example, we are calling st.ComputeGrade but st is of type student. However, since st is pointing to a GradStudent type of object, the call will track the vtable and call the ComputeGrade on the GradStudent class. This is the meaning of Polymorphism i.e., call the correct function at runtime.

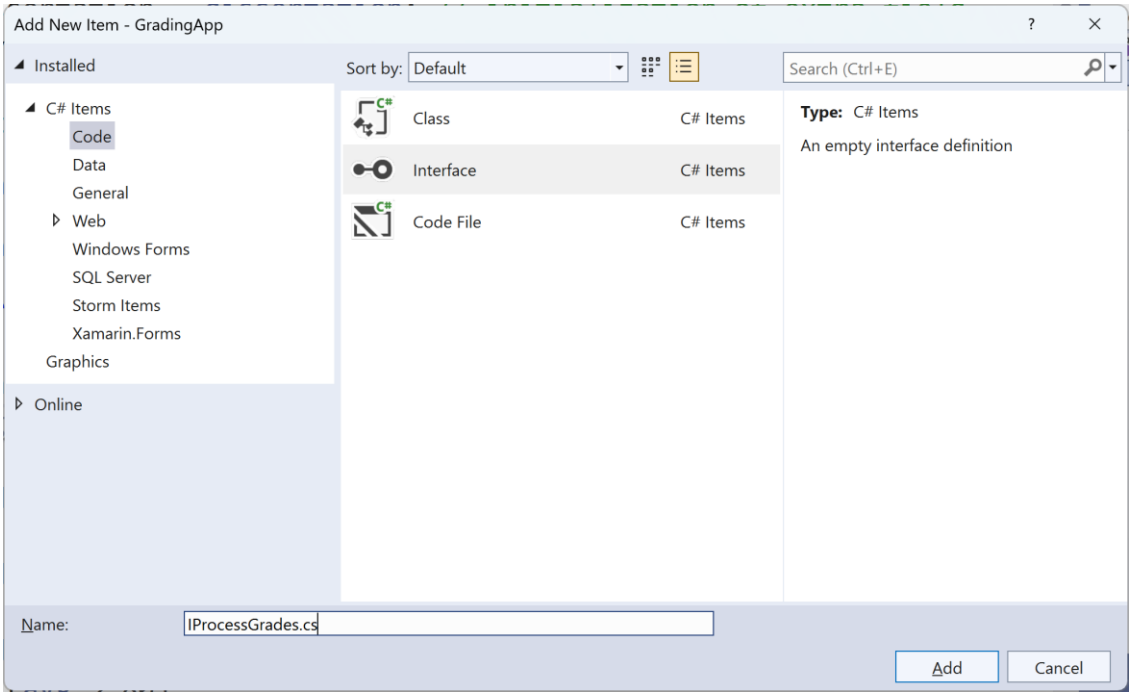


The purpose of abstraction in OOP is to hide the details of the implementation (i.e., code for the methods) from the user of a class. This way, the user only cares about how to call a method of a class and not how it is implemented. In C# (and in design patterns), we accomplish abstraction via interface declarations. An interface simply describes the name of the method, what input it takes and what output it returns. In Design Patterns, we often write the interface first, and then write the class that implements the interface. For example, the ProcessGrades class needs two methods that we can first describe in an interface.

Add a new item to the project by right clicking on the project name, choosing Add New Item as shown below.



Then select interface and name it as IProcessGrades.cs as shown below.

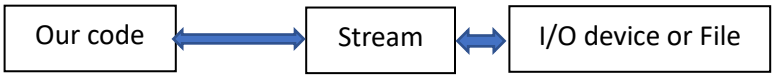


Type the following code in the IProcessGrades.cs file.

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

Note that when declare the prototype for the function in an interface, there is no public or private declaration. Once a class implements the interface, it will usually mark the method as public.

For reading and writing to files, all OOP languages use the concept of streams. We associate a stream object with the file first and then read from or write to the stream as shown below.



Depending upon the type of file, the stream can be binary, or a text type of stream. For reading/writing to/from text files, the prebuilt stream class is called StreamReader or StreamWriter.

Add a class to the project called ProcessGrades with the following code in it.

```
    internal class ProcessGrades: IProcessGrades
    {
    }
```

The above declaration indicates that the ProcessGrades class will implement the interface IProcessGrades and provide the code for the methods in the interface. Right click on the IProcessGrades and from quick actions, select implement interface. Then modify the code in the ProcessGrades class to appear as:

```
    internal class ProcessGrades : IProcessGrades
    {
        public List<Student> STList { get; set; } = new List<Student>();

        public void ProcessAndWriteGrades(string outFileName)
        {
            StreamWriter sw = new StreamWriter(outFileName);
            try
            {
                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; // if error, send error back to the calling code
            }
```

```
            Finally // always triggered even if there is no error
            {
                sw.Close();
            }
        }

        public void ReadStudentData(string inputFileName)
        {   // will read students into the STList
            try
            {
                STList.Clear(); // clear the list of students
                StreamReader sr = new StreamReader(inputFileName);
                string sline = sr.ReadLine();
                while (sline != null)
                {
                    Student st = null; // base class reference
                    string[] parts = sline.Split(new char[] { '\t' });
                    if (parts.Length == 6) // undergrad student
                    {
                        if (parts[3].ToUpper() == "UNDERGRAD")
                        {
                            st = new UnderGradStudent(int.Parse(parts[0]), parts[1],
                                parts[2], int.Parse(parts[4]), int.Parse(parts[5]));
                        }
                    }
                    if (parts.Length == 7) // grad student
                    {
                        if (parts[3].ToUpper() == "GRADUATE")
                        {
                            st = new GradStudent(int.Parse(parts[0]), parts[1],
                                parts[2], int.Parse(parts[4]), int.Parse(parts[5]),
parts[6]);
                        }
                    }
                    if (parts.Length == 8) // Phd student
                    {
                        if (parts[3].ToUpper() == "PHDCPSC")
                        {
                            st = new PhdStudent(int.Parse(parts[0]), parts[1],
                                parts[2], int.Parse(parts[4]), int.Parse(parts[5]),
parts[6], parts[7]);
                        }
                    }
                    if (st != null)
                        STList.Add(st);  // add student to the list
                    sline = sr.ReadLine();  // read next line
                }
            }
            catch
            {
                throw;  // if error, send error back to the calling code
            }
        }
    }
```

You will notice that the above code use try{ } catch { } to enclose all the code inside a method. In all OOP languages, the error handling is properly handled via the the try, catch mechanism. If any line of code runs

into an error, the remaining lines in the try block (after the line that caused error) are skipped and the code in the catch part is executed.

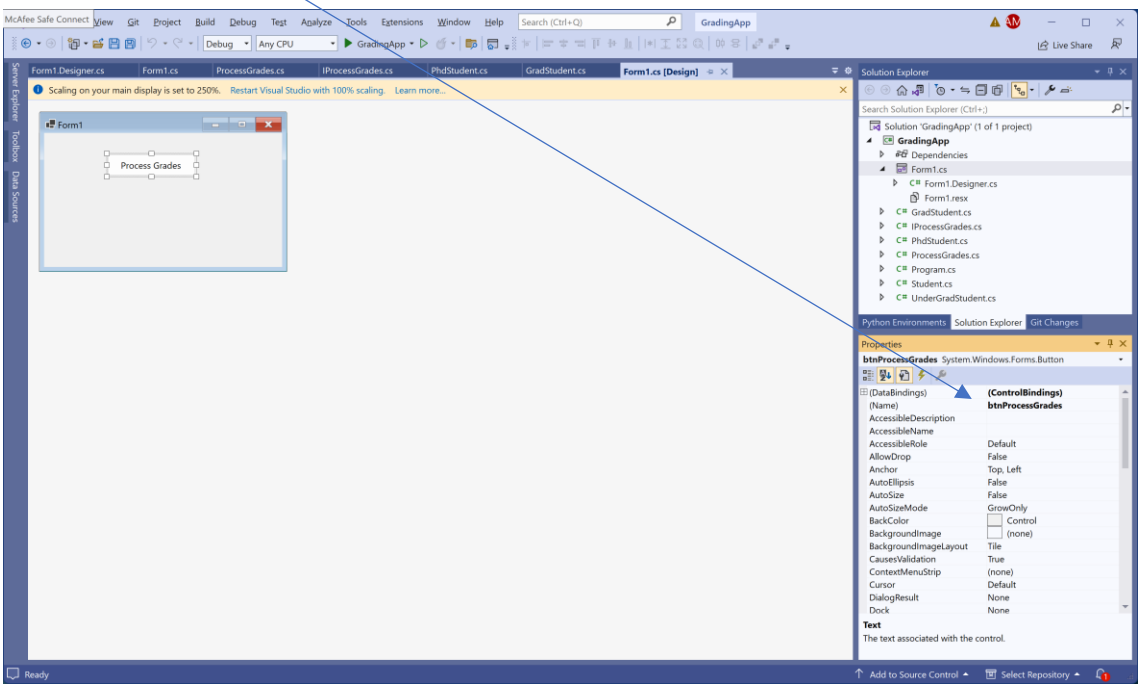
```
try
{
        -----
        -----
        Some code that causes an error
        -----




        -----
}
catch
(
        // error handling code
}
```

If the function being written is in a class that is not directly tied to a user interface, then the error handling code typically has a throw statement, which means send the error to whoever called this function. If the function that has the try catch block is part of the user interface class, then typically we will display the error in the catch part. If no error occurs, then all the code in the try block is executed and the code in the catch block is skipped.

Double click on the Form1 in the solution explorer and add a button to the form from the toolbox with a name of btnProcessGrades and a text property of “Process Grades”.

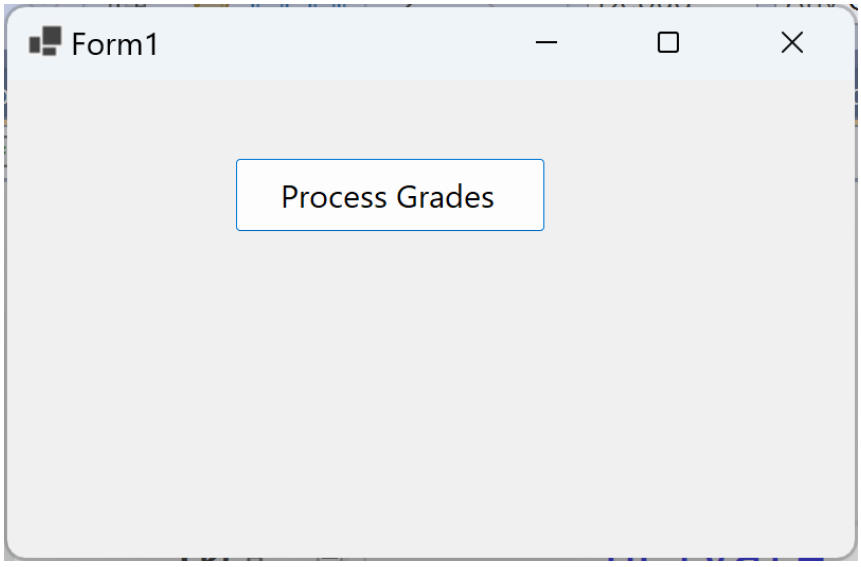


Double click on the button and type the following code in the button handler. The code you are typing is shown in bold.

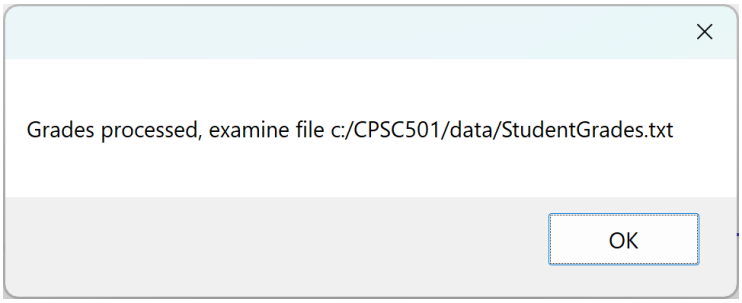
```
namespace GradingApp
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnProcessGrades_Click(object sender, EventArgs e)
        {
            try
            {
                string inputFile = "c:/CPSC501/data/StudentsData.txt";
                string outputFile = "c:/CPSC501/data/StudentGrades.txt";
                ProcessGrades pg = new ProcessGrades();
                pg.ReadStudentData(inputFile);
                pg.ProcessAndWriteGrades(outputFile);
                MessageBox.Show("Grades processed, examine file " + outputFile);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
    }
}
```

Run the project by Debug->Start without Debugging. Click on the Process Grades button.



The output will appear as:



If you examine the StudentGrades.txt, you will see the grades assigned as: