Sorting a List of Objects:
Let us learn how to sort a list by using the predefined function.

Item	Description
For linear data	**Syntax:** **ListName.sort();** **Working:** Sorts a string, int, or float type list

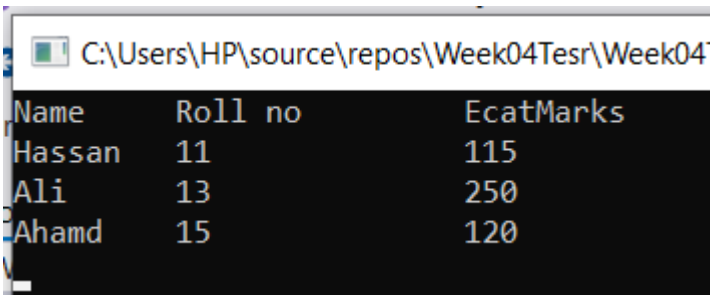
Code:	`List<int> integerList = new List<int>() { 1, 5, 4, 7, 2, 3, 6 };`
	`integerList.Sort();`
	`foreach (int i in integerList)`
	`  Console.Write(i + " ");`
	`Console.ReadKey();`

Solution:	C:\Users\HP\source\repos\Week04Tesr\Week04Tesr\bin\Debug\Week04Tesr.exe `1 2 3 4 5 6 7`
For Class data	**Syntax:** **newList = listName.OrderBy(o => o.classAttribtue).ToList();** **Working:** Sorts a list in ascending order based on the given attribute value **Syntax:** **newList = listName.OrderByDescending(o => o.classAttribtue).ToList();** **Working:** Sorts a list in descending order based on the given attribute value

Code:

```
Student s1 = new Student("Ahamd", 15, 120);
Student s2 = new Student("Hassan", 11, 115);
Student s3 = new Student("Ali", 13, 250);
List<Student> studentList = new List<Student> (){ s1, s2, s3 };
List<Student> sortedList = studentList.OrderBy(o => o.rollno).ToList();
Console.WriteLine("Name \t Roll no \t EcatMarks");
foreach (Student s in sortedList)
{
    Console.WriteLine("{0} \t {1} \t \t {2}", s.name, s.rollno, s.ecatMarks);
}
Console.Read();
```

Solution:



Ascending Sorting of the list based on Roll Number

Problem 0: UAMS SYSTEM

University Admission Management System (Case Study Solution)

Read the following question carefully.

Self Assessment

1. Identify the classes within the following case study.

Academic branch offers different programs within different departments each program has a degree title and duration of degree.
Student Apply for admission in University and provides his/her name, age, FSC, and Ecat Marks and selects any number of preferences among the available programs.
Admission department prepares a merit list according to the highest merit and available seats and registers selected students in the program.
Academic Branch also add subjects for each program. A subject have subject code, credit hours, subjectType. A Program cannot have more than 20 Credit hour subjects.A Student Registers multiple subjects but he/she can not take more than 9 credit hours.
Fee department generate fees according to registered subjects of the students.

Try out yourself.

Don't worry.

There is a solution on the next page.

Identification of Classes

By looking at the above-mentioned self-assessment you can extract the following possible class-like structures from the given statement.

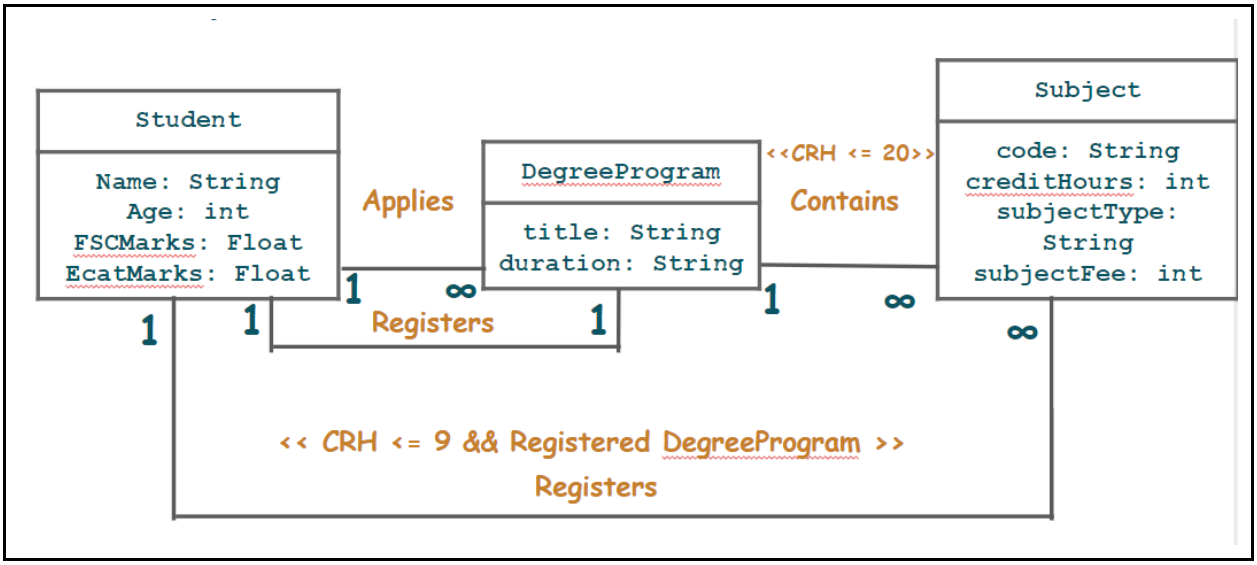
- Student Class
- Subject Class
- Program Class

Note: Create a separate class in the same BL(Business Logic) folder of your program.

Now Try to Build the Class Diagram/Domain Model of these classes.

Don't Worry. There is a solution ahead. First Try out yourself.

Class Diagram without the member functions



Let's Start with fun coding.

University Admission Management System (Through OOP)

Now that you have identified the classes in your program, it is time to start coding.

Solution:

Sr. #	Action	Description

1.	
	`class Student`
	`{`
	`public string name;`
	`public int age;`
	`public double fscMarks;`
	`public double ecatMarks;`
	`public double merit;`
	`public List<DegreeProgram> preferences;`
	`public List<Subject> regSubject;`
	`public DegreeProgram regDegree;`
	
	`public Student(string name, int age, double fscMarks, double ecatMarks, List<DegreeProgram> preferences)`
	`{`
	`this.name = name;`
	`this.age = age;`
	`this.fscMarks = fscMarks;`
	`this.ecatMarks = ecatMarks;`
	`this.preferences = preferences;`
	`regSubject = new List<Subject>();`
	`}`
		Creates a **Student** Class with one **Parameterized** Constructor.
		
		Important Note: Each student shall need a degree program preferences list and one registered subjects list and a selected Degree Program. These were determined through the relations between the Students Class and other Classes. Therefore, we need to include these attributes too.

1(a)	`class Subject`
	`{`
	`public string code;`
	`public string type;`
	`public int creditHours;`
	`public int subjectFees;`
	
	`public Subject(string code, string type, int creditHours, int subjectFees)`
	`{`
	`this.code = code;`
	`this.type = type;`
	`this.creditHours = creditHours;`
	`this.subjectFees = subjectFees;`
	`}`
	`}`
		In this code, we will create the **Subject** class. The attached code
		● Implements the **Subject class**
		● Provides Parameterized Constructor where the user must provide **subject code**, **subject type**, **subject fees,** and **credit hours** before creating a class object.

1(b)	`class DegreeProgram`
	`{`
	`public string degreeName;`
	`public float degreeDuration;`
	`public List<Subject> subjects;`
	`public int seats;`
	
	`public DegreeProgram(string degreeName, float degreeDuration, int seats)`
	`{`
	`this.degreeName = degreeName;`
	`this.degreeDuration = degreeDuration;`
	`this.seats = seats;`
	`subjects = new List<Subject>();`
	`}`
		In this code, we will create the **degree program** class. The attached code
		● Implements the **DegreeProgram**
		● Provides Parameterized Constructor where the user must provide **the degree name**, and **degree duration** before creating a class object.

<table>
<tr>
<td>1(c)</td>
<td>

```
public int calculateCreditHours()
{
    int count = 0;
    for (int x = 0; x < subjects.Count; x++)
    {
        count = count + subjects[x].creditHours;
    }
    return count;
}

public bool AddSubject(Subject s)
{
    int creditHours = calculateCreditHours();
    if(creditHours + s.creditHours <= 20)
    {
        subjects.Add(s);
        return true;
    }
    else
    {
        return false;
    }
}

public bool isSubjectExists(Subject sub)
{
    foreach (Subject s in subjects)
    {
        if (s.code == sub.code)
        {
            return true;
        }
    }
    return false;
}
```

</td>
<td>This code

- Includes member functions in the **degree program** class for adding **isSubjectExists** and **adding Subjects** and **caculateCreditHours().**

</td>
</tr>
<tr>
<td>3.</td>
<td>

```
public void calculateMerit()
{
    this.merit = (((fscMarks / 1100) * 0.45F) + ((ecatMarks / 400) * 0.55F)) * 100;

}
```

</td>
<td>Complete the **Student Class** by including the member function for performing the following tasks.

- Merit Calculator
- Registering Subjects for students

</td>
</tr>
</table>

	`public bool regStudentSubject(Subject s)`
	`{`
	`int stCH = getCreditHours();`
	`if (regDegree != null && regDegree.isSubjectExists(s) && stCH + s.creditHours <= 9)`
	`{`
	`regSubject.Add(s);`
	`return true;`
	`}`
	`else`
	`{`
	`return false;`
	`}`
	`}`	

3(a)	
	`public int getCreditHours()`
	`{`
	`int count = 0;`
	`foreach (Subject sub in regSubject)`
	`{`
	`count = count + sub.creditHours;`
	`}`
	`return count;`
	`}`
	
	`public float calculateFee()`
	`{`
	`float fee = 0;`
	`if (regDegree != null)`
	`{`
	`foreach (Subject sub in regSubject)`
	`{`
	`fee = fee + sub.subjectFees;`
	`}`
	`}`
	`return fee;`
	`}`
		Complete the **Student Class** by including the member function for performing the following tasks.
		● getCreditHours
		● calculateFee

Program.cs File: Let us now implement the Static Functions (in the program.cs file) for this project.		

<table>
<tr>
<td>4.</td>
<td>

```
static Student StudentPresent(string name)
{
    foreach (Student s in studentList)
    {
        if (name == s.name && s.regDegree != null)
        {
            return s;
        }
    }
    return null;
}

1 reference
static void calculateFeeForAll()
{
    foreach (Student s in studentList)
    {
        if (s.regDegree != null)
        {
                Console.WriteLine(s.name + " has " + s.calculateFee() + " fees");
        }
    }
}
```

</td>
<td>

Implement functions for

- Checking if a student exists in the list of students
- A function to show the “calculated fee” of all the students.

Note: The function call inside the **calculateFeeForAll()** function, written as **s.calculateFee()** is actually calling the function inside the **Student Class**.

</td>
</tr>
<tr>
<td>5.</td>
<td>

```
static void registerSubjects(Student s)
{
    Console.WriteLine("Enter how many subjects you want to register");
    int count = int.Parse(Console.ReadLine());
    for (int x = 0; x < count; x++)
    {
        Console.WriteLine("Enter the subject Code");
        string code = Console.ReadLine();
        bool Flag = false;
        foreach (Subject sub in s.regDegree.subjects)
        {
            if (code == sub.code && !(s.regSubject.Contains(sub)))
            {
                s.regStudentSubject(sub);
                Flag = true;
                break;
            }
        }
        if (Flag == false)
        {
            Console.WriteLine("Enter Valid Course");
            x--;
        }
    }
}
```

</td>
<td>

This code implements a function to allow users to **register any number of subjects** as they want.

</td>
</tr>
</table>

<table>
<tr>
<td>6.</td>
<td>

```
static List<Student> sortStudentsByMerit()
{
    List<Student> sortedStudentList = new List<Student>();
    foreach (Student s in studentList)
    {
        s.calculateMerit();

    }
    sortedStudentList = studentList.OrderByDescending(o => o.merit).ToList();
    return sortedStudentList;
}

1 reference
static void giveAdmission(List<Student> sortedStudentList)
{
    foreach (Student s in sortedStudentList)
    {
        foreach (DegreeProgram d in s.preferences)
        {
            if (d.seats > 0 && s.regDegree == null)
            {
                s.regDegree = d;
                d.seats--;
                break;
            }
        }
    }
}
```

</td>
<td>

This code creates functions for the following operations

- Sorting the **student list** based on the **student merit**
- Giving admission to user and setting the value of Data Member **regDegree**

</td>
</tr>
<tr>
<td>7.</td>
<td>

```
static void printStudents()
{
    foreach (Student s in studentList)
    {
        if (s.regDegree != null)
        {
            Console.WriteLine(s.name + " got Admission in " + s.regDegree.degreeName);
        }
        else
        {
            Console.WriteLine(s.name + " did not get Admission");

        }
    }
}

static void clearScreen()
{
    Console.WriteLine("Press any key to Continue..");
    Console.ReadKey();
    Console.Clear();
}
```

</td>
<td>

This code implements the functionality for

- **Printing** all the students who got admission as well as those who failed.
- Function to **clear screen**.

</td>
</tr>
</table>

8.	
	`static void viewStudentInDegree(string degName)`
	`{`
	`  Console.WriteLine("Name\tFSC\tEcat\tAge");`
	`  foreach (Student s in studentList)`
	`  {`
	`  if (s.regDegree != null)`
	`  {`
	`  if (degName == s.regDegree.degreeName)`
	`  {`
	`  Console.WriteLine(s.name + "\t" + s.fscMarks + "\t" + s.ecatMarks + "\t" + s.age);`
	`  }`
	`  }`
	`  }`
	`}`
	`static void viewRegisteredStudents()`
	`{`
	`  Console.WriteLine("Name\tFSC\tEcat\tAge");`
	`  foreach (Student s in studentList)`
	`  {`
	`  if (s.regDegree != null)`
	`  {`
	`  Console.WriteLine(s.name + "\t" + s.fscMarks + "\t" + s.ecatMarks + "\t" + s.age);`
	`  }`
	`  }`
	`}`
		Functions to
		● View the **registered students** in the system
		● **View registered students** in a **specific degree**

9.	`static void addIntoDegreeList(DegreeProgram d)`
	`{`
	`  programList.Add(d);`
	`}`
	`static DegreeProgram takeInputForDegree()`
	`{`
	`  string degreeName;`
	`  float degreeDuration;`
	`  int seats;`
	`  Console.Write("Enter Degree Name: ");`
	`  degreeName = Console.ReadLine();`
	`  Console.Write("Enter Degree Duration: ");`
	`  degreeDuration = float.Parse(Console.ReadLine());`
	`  Console.Write("Enter Seats for Degree: ");`
	`  seats = int.Parse(Console.ReadLine());`
	
	`  DegreeProgram degProg = new DegreeProgram(degreeName, degreeDuration, seats);`
	`  Console.Write("Enter How many Subjects to Enter: ");`
	`  int count = int.Parse(Console.ReadLine());`
	`  for (int x = 0; x < count; x++)`
	`  {`
	`  degProg.AddSubject(takeInputForSubject());`
	`  }`
	`  return degProg;`
	`}`
		Functions for
		● **Creating new degree**
		● **Adding a degree** into the **Program List**

10.	
	`static Subject takeInputForSubject()`
	`{`
	`  string code;`
	`  string type;`
	`  int creditHours;`
	`  int subjectFees;`
	`  Console.Write("Enter Subject Code: ");`
	`  code = Console.ReadLine();`
	`  Console.Write("Enter Subject Type: ");`
	`  type = Console.ReadLine();`
	`  Console.Write("Enter Subject Credit Hours: ");`
	`  creditHours = int.Parse(Console.ReadLine());`
	`  Console.Write("Enter Subject Fees: ");`
	`  subjectFees = int.Parse(Console.ReadLine());`
	`  Subject sub = new Subject(code, type, creditHours, subjectFees);`
	`  return sub;`
	`}`
	`static void addIntoStudentList(Student s)`
	`{`
	`  studentList.Add(s);`
	`}`
		Functions for
		● **Creating** new **Subject**
		● Adding student to **Students List**

11.	
	`static Student takeInputForStudent()`
	`{`
	`  string name;`
	`  int age;`
	`  double fscMarks;`
	`  double ecatMarks;`
	`  List<DegreeProgram> preferences = new List<DegreeProgram>();`
	`  Console.Write("Enter Student Name: ");`
	`  name = Console.ReadLine();`
	`  Console.Write("Enter Student Age: ");`
	`  age = int.Parse(Console.ReadLine());`
	`  Console.Write("Enter Student FSc Marks: ");`
	`  fscMarks = double.Parse(Console.ReadLine());`
	`  Console.Write("Enter Student Ecat Marks: ");`
	`  ecatMarks = double.Parse(Console.ReadLine());`
	`  Console.WriteLine("Available Degree Programs");`
	`  viewDegreePrograms();`
		Function for
		● Creating a new student by taking information from the user

```
    Console.Write("Enter how many preferences to Enter: ");
    int Count = int.Parse(Console.ReadLine());
    for (int x = 0; x < Count; x++)
    {
        string degName = Console.ReadLine();
        bool flag = false;
        foreach (DegreeProgram dp in programList)
        {
            if (degName == dp.degreeName && !(preferences.Contains(dp)))
            {
                preferences.Add(dp);
                flag = true;
            }

        }
        if (flag == false)
        {
            Console.WriteLine("Enter Valid Degree Program Name");
            x--;
        }
    }
    Student s = new Student(name, age, fscMarks, ecatMarks, preferences);
    return s;
}
```

12.

```
static void viewDegreePrograms()
{
    foreach (DegreeProgram dp in programList)
    {
        Console.WriteLine(dp.degreeName);
    }
}
static void header()
{
    Console.WriteLine("****************************************");
    Console.WriteLine("                    UAMS                ");
    Console.WriteLine("****************************************");
}
static void viewSubjects(Student s)
{
    if (s.regDegree != null)
    {
        Console.WriteLine("Sub Code\tSub Type");
        foreach (Subject sub in s.regDegree.subjects)
        {
            Console.WriteLine(sub.code + "\t\t" + sub.type);
        }
    }
}
```

Functions for

- **View all degrees**
- **View Subjects**
- Print **Header**

13.	
	`static int Menu()`
	`{`
	`  header();`
	`  int option;`
	`  Console.WriteLine("1. Add Student");`
	`  Console.WriteLine("2. Add Degree Program");`
	`  Console.WriteLine("3. Generate Merit");`
	`  Console.WriteLine("4. View Registered Students");`
	`  Console.WriteLine("5. View Students of a Specific Program");`
	`  Console.WriteLine("6. Register Subjects for a Specific Student");`
	`  Console.WriteLine("7. Calculate Fees for all Registered Students");`
	`  Console.WriteLine("8. Exit");`
	`  Console.Write("Enter Option: ");`
	`  option = int.Parse(Console.ReadLine());`
	`  return option;`
	`}`
		function to print the **main menu**

Let us now implement the **Main Driver Program** for this project.		

14.	`public class Program`
	`{`
	`  static List<Student> studentList = new List<Student>();`
	`  static List<DegreeProgram> programList = new List<DegreeProgram>();`
	`  static void Main(string[] args)`
	`  {`
		Create the following global lists.
		● List for all Students
		● List of all Programs

14(a)	`static void Main(string[] args)`
	`{`
	`  int option;`
	`  do`
	`  {`
	`  option = Menu();`
	`  clearScreen();`
	`  if (option == 1)`
	`  {`
	`  if (programList.Count > 0)`
	`  {`
	`  Student s = takeInputForStudent();`
	`  addIntoStudentList(s);`
	`  }`
	`  }`
	`  else if (option == 2)`
	`  {`
	`  DegreeProgram d = takeInputForDegree();`
	`  addIntoDegreeList(d);`
	`  }`
		Implement the Main Menu

14(b)

```
else if (option == 3)
{
    List<Student> sortedStudentList = new List<Student>();
    sortedStudentList = sortStudentsByMerit();
    giveAdmission(sortedStudentList);
    printStudents();
}
else if (option == 4)
{
    viewRegisteredStudents();
}
else if (option == 5)
{
    string degName;
    Console.Write("Enter Degree Name: ");
    degName = Console.ReadLine();
    viewStudentInDegree(degName);
}
```

14(c)

```
        else if (option == 6)
        {
            Console.Write("Enter the Student Name: ");
            string name = Console.ReadLine();
            Student s = StudentPresent(name);
            if (s != null)
            {
                viewSubjects(s);
                registerSubjects(s);
            }

        }
        else if (option == 7)
        {
            calculateFeeForAll();
        }
        clearScreen();
    }
    while (option != 8);

    Console.ReadKey();
}
```

Problem 1:

Case Study: Ocean Navigation

In ocean navigation, locations are measured in degrees and minutes of latitude and longitude. Thus if you're lying off the mouth of Papeete Harbor in Tahiti, your location is 149 degrees 34.8 minutes west longitude, and 17 degrees 31.5 minutes south latitude. This is written as 149°34.8' W, 17°31.5' S. There are 60 minutes in a degree. (An older system also divided a minute into 60 seconds, but the modern approach is to use decimal minutes instead.) Longitude is measured from 0 to 180 degrees, east or west from Greenwich, England, to the international dateline in the Pacific. Latitude is measured from 0 to 90 degrees, north or south from the equator to the poles.
Create a class **angle** that includes three member variables: an int for degrees, a float for minutes, and a char for the direction letter (N, S, E, or W). This class can hold either a latitude variable or a longitude variable.
Write one member function to change the angle value (in degrees and minutes) and a direction given from the user, and a second to display the angle value in 179°59.9' E in string format. Also write a three-argument constructor.
You can use this to print a degree (°) symbol.

```
Console.WriteLine("\u00b0");
```

Create a class called **ship** that incorporates a ship's number and location. Use two variables of the angle class to represent the ship's latitude and longitude for the ship's location. Write a parameterized constructor to initialize the attributes of the ship class. A member function of the ship class should print the position (latitude and longitude) of the ship; another should report/print the serial number.

Your Tasks:

Task 1: Identify the Classes and Make the Class Diagram by adding the relation, multiplicity and collaboration Step by Step
Task 2: Maintain a list of ships in the main.
Driver Program Menu:

1. Add Ship
2. View Ship Position
3. View Ship Serial Number
4. Change Ship Position
5. Exit

If the user Enters 1 then
Enter Ship Number: "123TG"

Enter Ship Latitude:
Enter Latitude's Degree: 149
Enter Latitude's Minute: 34.8
Enter Latitude's Direction: W
Enter Ship Longitude:
Enter Longitude's Degree: 17
Enter Longitude's Minute: 31.5
Enter Longitude's Direction: S

If the user Enters 2 then
Enter Ship Serial Number to find its position: "123TG"
Ship is at 149°34.8' W and 17°31.5' S

If the user Enters 3 then
Enter the ship latitude: "149°34.8' W"
Enter the ship longitude: "17°31.5' S"
Ship's serial number is 123TG

If the user Enters 4 then
Enter Ship's serial number whose position you want to change: "123TG"
Enter Ship Latitude:
Enter Latitude's Degree: 170
Enter Latitude's Minute: 3.8
Enter Latitude's Direction: E
Enter Ship Longitude:
Enter Longitude's Degree: 12
Enter Longitude's Minute: 39.5
Enter Longitude's Direction: W

Problem 2: Magical Duel Challenge

Having gotten rather sick of always being paired together in sciency literature, Alice and Bob have decided to finally settle their differences with a magical duel. They'll each learn some skills and then battle it out.

Your Goal

Your job is to write the 2 Classes **Player and Stats** which will handle all the combat mechanics.

1. **Stats Class**

Data Members:
Your Stats class has the following properties:

1. name: Skill Name
2. damage: the raw damage done (assuming 0 effective armor),
3. description: the description of the attack (for humans to read),
4. penetration: Armor penetration amount (see "Armor" below),
5. cost: Cost, in energy points,
6. heal: Optional heal value (some skills heal the caster on cast!)

Constructor:

- Your Stats instances will be constructed as new Stats(damage, penetration, heal, cost, description)

2. **Player Class**

Data Members:

Let's look at the Player class's properties first. You'll need:

- A health variable **hp**.
- A maxHealth variable **maxHp**.
- An energy variable **energy**.
- A maxEnergy variable **maxEnergy**.
- An armor value with variable named **armor**.
- A **name** variable for player name.
- An object of statistics named **skillStatistics**.

Constructor:

- Your Player instances will be constructed as new Player(name,health,energy,armor)

Member Functions:

1. Write the functions to update Health, Energy, Armor and name. (Make sure that you cannot have less than 0 health or energy, and your health and/or energy cannot be greater than their respective "max" values.)
2. **learnSkill()** Method
   Your class must implement a method called **learnSkill**.
   Basics/Functionality
   This method takes 1 parameter: an object containing skill statistics.

   Most importantly, after adding a skill:
   alice.learnSkill(skillStats)

3. You should then be able to call that skill using the function **attack()**. This function will take 1 parameter as input which is of type Player

   alice.attack(bob);

   In general, the **attack()** function should return a string describing what happened, as well as changing the relevant numbers on both the target and "caster".

   Logic
   Armor: Your attack() method will start out by subtracting an armor penetration stat from the target's armor value to get an "effective armor" value. That is, if Alice attacks Bob with a skill with 5 armor penetration, and Bob's armor is 50, then Bob's effective armor for this attack is 50-5 = 45.

   Energy: If the skill costs more energy than the character currently has, return (player name) attempted to use (skill name), but didn't have enough energy!. Otherwise, subtract the energy cost from the character's energy, and continue.

   Damage: Damage here is pretty easy. Consider that the minimum armor value is 0, the maximum is 100, and each percent effective armor decreases damage by one percent. An example:
   Alice attacks Bob for 50 damage. Bob's effective armor rating is 25.
   Alice does 50*((100-25)/100) = 37.5 damage.

   Attack String: You'll need to return a string describing what happened. The first part of the returned string should describe the attack itself, and should look like this: (attacking player name) used skill (skill name), (skill description), against (target name), doing (calculated damage) damage!
   Next, if the skill healed, append (attacking player name) healed for (heal amount) health.
   Finally, if the target player died, append (target name) died. Otherwise, append (target name is at (targ hpPerc) % health.
   Return this string, and don't forget to actually apply the damage/health changes!

Example

```
Player alice = new Player("Alice", 110, 50, 10);
Player bob = new Player("Bob", 100, 60, 20);

Stats fireball = new Stats(“fireball”, 23, 1.2, 5, 15, "a firey magical attack");
```

```
alice.learnSkill(fireball);

Console.WriteLine(alice.attack(bob));
// Alice used fireball, a firey magical attack, against Bob, doing
// 18.68 damage! Alice healed for 5 health! Bob is at 81.32% health.

Stats superbeam = new Stats("superbeam", 200, 50, 50, 75, "an overpowered attack, pls nerf")

Console.WriteLine(bob.attack(alice))
// Bob attempted to use superbeam, but didn't have enough energy!
```

Problem 3: Play Magical Duel

Make a complete menu driven program to play the above game. Make a list of Stats objects as well as the list of Player objects.
Following can be the menu options

1. **Add Player:**
    - Enter details such as player name, health, maxHealth, energy, maxEnergy and armor.
    - Store this object in the list of Players.
2. **Add Skill Statistics:**
    - Enter details such as skill name, damage, penetration, heal, cost, and description.
    - Store this in the list of SkillsStatistics.
3. **Display Player Info:**
    - This option displays the current information for both players, including their health, energy, and armor.
4. **Learn a Skill:**
    - Take the name of the player from the user.
    - Take the name of the skill from the user.
    - Choose this option to teach a new skill to the player.
5. **Attack:**
    - Engage in combat by selecting this option.
    - Choose the player who is going to attack from the list.
    - Choose the target player to attack from the list.
    - The attack will be executed, and the outcome, including damage dealt and healing received, will be displayed.
6. **Exit:**
    - Choose this option to exit the program.

Problem 4: Business Application

Identify the Classes from your Business Application and Draw the Class Diagram and Implement your Business Application with Separate Classes in BL Folder.

Problem 5: Game Project

Identify the Classes from your Project and Draw the Class Diagram and Implement your Game with Separate Classes in BL Folder.