■ Student Grade Tracker — Study Guide Version

Purpose: Demonstrate variables, lists, loops, functions, I/O, validation, exceptions, classes, inheritance, and simple polymorphism in one coherent example.

Program Overview

You create a Student (actually an HonorsStudent) and then collect scores in a loop. The class encapsulates calculations (average, highest, lowest) and exposes a simple DisplayReport(). Invalid input is handled with try/catch; invalid ranges are rejected by AddScore().

Annotated Code (with line-by-line comments)

```
using System;
using System.Collections.Generic;

namespace GradeTracker
{
    // ■ Class 'Student' models a student and their scores.
    // Encapsulation: fields are private, access via methods.
    public class Student
    {
        private string _name;                  // holds the student's name
        private List<double> _scores = new(); // dynamic list for scores (List<T>)

        // Constructor: requires a name when creating a student
        public Student(string name)           // parameter 'name' is assigned to _name
        {
            _name = name;                      // store the provided name
        }

        // AddScore: validates and adds a score to the list
        public void AddScore(double score)
        {
            // Validation: only accept 0..100 inclusive
            if (score < 0 || score > 100)
            {
                Console.WriteLine("■■  Invalid score. Enter 0–100.");
                return;                        // leave early if invalid
            }
            _scores.Add(score);                // add to List<double>
        }

        // GetAverage: sums all scores and divides by count (guard for empty list)
        public double GetAverage()
        {
            if (_scores.Count == 0) return 0; // avoid divide-by-zero
            double sum = 0;                    // accumulator variable
            foreach (double s in _scores)      // foreach loop to add up items
            {
                sum += s;                      // arithmetic operator +
            }
            return sum / _scores.Count;        // average calculation
        }

        // GetHighest: find maximum value by scanning the list
        public double GetHighest()
        {
            if (_scores.Count == 0) return 0;
            double max = double.MinValue;      // start lower than any real score
            foreach (double s in _scores)
            {
                if (s > max) max = s;          // comparison operator >
            }
            return max;
        }
    // GetLowest: find minimum value by scanning the list
    public double GetLowest()
    {
        if (_scores.Count == 0) return 0;
        double min = double.MaxValue;     // start higher than any real score
        foreach (double s in _scores)
        {
            if (s < min) min = s;         // comparison operator <
        }
        return min;
    }

    // GetPerformance: conditional logic classifying by average
    public virtual string GetPerformance()
    {
        double avg = GetAverage();
        if (avg >= 90) return "Excellent";          // if branch
        else if (avg >= 75) return "Good";          // else-if
        else if (avg >= 60) return "Needs Improvement";
        else return "Failing";                       // else branch
    }

    // DisplayReport: outputs a summary—abstraction over details
    public void DisplayReport()
    {
        Console.WriteLine($"\n■ Student: {_name}");            // string interpolation
        Console.WriteLine($"Scores Entered: {_scores.Count}");  // property Count
        Console.WriteLine($"Average Score: {GetAverage():F2}"); // format to 2 decimals
        Console.WriteLine($"Highest Score: {GetHighest()}");
        Console.WriteLine($"Lowest Score: {GetLowest()}");
        Console.WriteLine($"Performance: {GetPerformance()}");
    }
}

// ■ HonorsStudent inherits Student (Inheritance)
// Polymorphism: overrides performance expectations.
public class HonorsStudent : Student
{
    public HonorsStudent(string name) : base(name) { } // call base constructor

    // 'override' changes behavior of virtual method in base class
    public override string GetPerformance()
    {
        double avg = GetAverage();        // reuse base computation
        if (avg >= 95) return "Outstanding (Honors)";
        return base.GetPerformance();     // defer to base classification
    }
}

class Program
{
    // Entry point of the console app
    static void Main(string[] args)
    {
        Console.WriteLine("■ Welcome to the Student Grade Tracker!\n"); // Output

        Console.Write("Enter the student's name: ");                      // Prompt
        string name = Console.ReadLine();                                 // Input

        // Demonstrate inheritance via HonorsStudent instance (is-a Student)
        Student student = new HonorsStudent(name);

        // Loop for collecting scores until the user types 'done'
        while (true)
        {
            Console.Write("Enter a score (or 'done' to finish): ");
            string input = Console.ReadLine();

            // Convert to lower-case and compare strings (==)
```

```
                if (input.ToLower() == "done")
                {
                    break;                           // exit loop
                }

                try
                {
                    // Parse to double (may throw on invalid input)
                    double score = double.Parse(input);
                    student.AddScore(score);         // encapsulated add with validation
                }
                catch (Exception)                    // Exception Handling
                {
                    Console.WriteLine("■■  Please enter a numeric value (e.g., 87.5).");
                }
            }

            // Final summary
            student.DisplayReport();

            Console.WriteLine("\n■ Program complete. Press any key to exit.");
            Console.ReadKey();                       // pause so output is visible
        }
    }
}
```

Flow Summary

1) Main prompts for name → 2) Create HonorsStudent → 3) Loop: read scores until 'done' → 4) AddScore validates and stores → 5) After loop, call DisplayReport() which calls calculations internally.