# CMPE 1666
# Intermediate Programming

Lecture 3- Introduction To Lists

# Storing Multiple Entities

- So far, we have used arrays for storing multiple data entities.
- Advantages of the array:
  - Can store multiple entities at a given time.
  - Can access any of its element using an index
- Drawback of the array,
  - Size has to be known before we create the array
  - Cannot insert an element at any random position
  - Cannot delete an element from an array

# List- Overview

- The List is similar to the array in that:
  - it can store multiple entities of a given type
  - We can access an element using the name of the List variable and an index.
  - The List variable is a reference type

# List- Overview

- However, the List is more flexible than the array.
  - We can create an empty List at the beginning, then add elements as and when required (No need to know the size beforehand).
  - The List can grow and shrink as required (It's a vector type).
  - We can inert an element before existing elements or any position in between existing elements and the List will grow.
  - We can delete existing elements from a List and the List will shrink

# List- Overview

- The **List** type is generic, meaning when you create a List, there is a place holder for the type:

```
List<int> myData = new List<int>();
```

- In this case, the List is being created for type int. A List is a reference type, so it is created with the new keyword.
- The empty constructor (the empty round brackets), means a new, empty collection of int is to be created.
- The object reference, **myData**, is bound to the List<int>

- Create a console application called **lecture3Demo1** to work wit the examples in the upcoming slides

# Adding Data to a list

- The **Add** method adds data to the end of the list.
- The **Insert** method adds the data to a specific index

```
List<string> myData = new List<string>();
  myData.Add("Monkey");
  myData.Add("Elephant");
  myData.Insert(0, "Goat");
```

# Accessing Elements of the List

- The List is maintained in the order that you add or insert items. You may use the subscript operator [] to access individual elements in the List, and just like an array, you must take care not to exceed the index range of the List!

```
List<string> myData = new List<string>();
myData.Add("Monkey");
myData.Add("Elephant");
myData.Insert(0, "Goat");
Console.WriteLine(myData[0]); // OK (Goat)
Console.WriteLine(myData[55]); // exception: Index was out of range.
```

- **The Count** property gives the number of elements in the list.
- Remember: the index range is 0 to Count – 1!

```
Console.WriteLine($"There are {myData.Count} elements in the list!");
```

# Iterating Through a list

- Arrays and List fall into a special category in C# -> they implement the IEnumerable class
- Anything in C# that is IEnumerable may be used with **foreach** to enumerate (or visit all elements):
- A **foreach** requires that you specify the type and name for a range variable, that will become each element in the collection in the iterations of the foreach. The range variable type must match your collection type:

```
List<string> myData = new List<string>();
myData.Add("Monkey");
myData.Add("Elephant");
myData.Insert(0, "Goat");
foreach (string s in myData)
        Console.WriteLine(s);
```

# Iterating Through a list

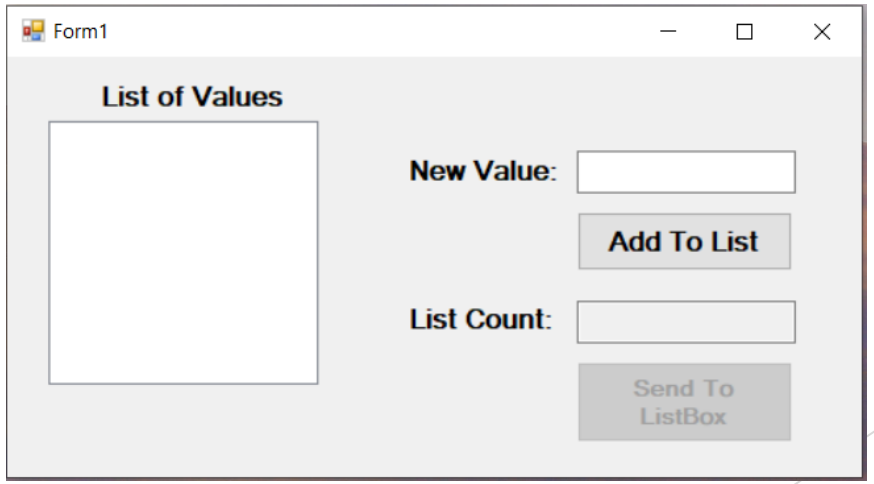
- A **foreach** will visit each element in the collection and automatically stop looping when it has visited all elements
- The **foreach** construct is nice when you just want to iterate over the elements. It does not, however, let you know where you are in the collection by index.
- For that, a for loop is more appropriate:

```
for (int i = 0; i < myData.Count; ++i)
        Console.WriteLine($"Animal #{i}: {myData[i]}");
```

- Also, be aware that a **foreach** loop is read-only. You cannot modify the elements.

# Lecture3-Demo 2

- Write a Form-based program that has the controls shown below. (Name the controls **UI_ValuesList_Lbx, UI_NewValue_Tbx, UI_AddToList_btn, UI_Count_Tbx** and **UI_SendToListBox_Btn**
- Initially, the "**Add To List**" button is enabled while the "**Send To List Box**" button is disabled.
- The Form must contain a member-variable of type **List<string>** and it also creates the corresponding object.
- When the user clicks on the "**Add To List**" button, the text in the text box must be added to the **List<string>** object. The "List Count" textbox must display the number of strings in the List object. When List object has 10 strings, the "**Add To List**" button must be disabled and the "**Send To List Box**" button enabled.
- When the user clicks on the "**send To List Box**" button, all the strings in the List<string> object must be added to the list box.



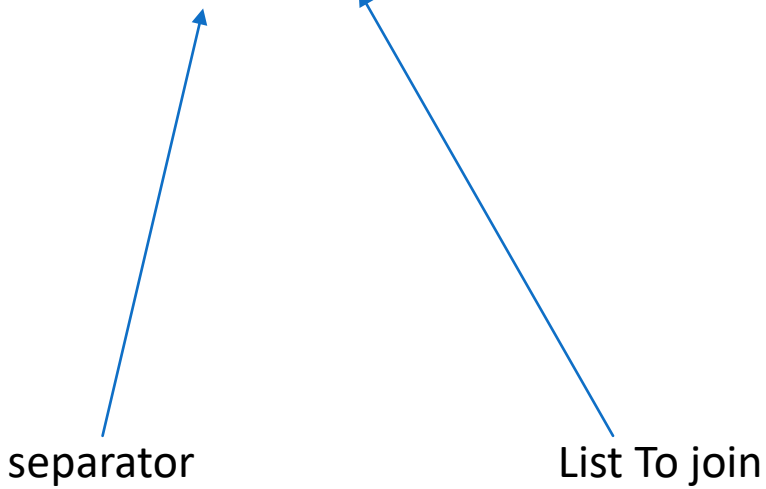
# Building a List from another collection

- We can create a new list and assign its values from an array or from anotherList
- E.g. List<int> list1=new List<int>{59,23,56,24,35,12,21}

  List<int> list2 = new List<int>(List1)

# Using string.join on a list

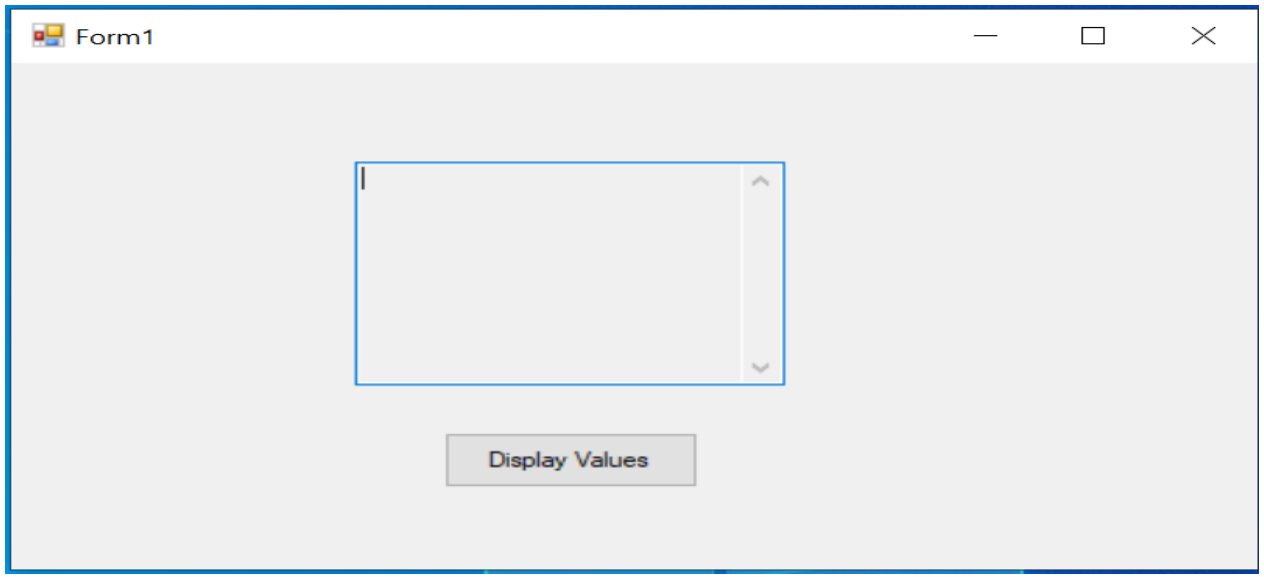
- We can convert a list of primitive values into a string with a separator by using the string.join() method.
- string.Join() take as arguments a separator and the list to join, E.g. string.join(“, ”, myList);




- This is handy for displaying a list in a multiline textbox.

# Lecture3-Demo3

- Create a UI with the controls shown below.
- Name the multiline textbox as **UI_Display_Tbx** and the button as **UI_Display_Btn**




- To the form class, add a list of **int** type and initialize it with the values: 50, 26, 28, 49, 72, 13, 15, 92, 17, 43, 18, 33, 12,  30, 25
- To the click event handler of the button, add the code to use string.join() to display the values in the multiline textbox. Use a “,” and a space as separator.

# Other List Methods

- So far, we have looked at the Add() and Insert() methods.
- The **List** class offers many other useful methods, for performing common tasks.
- At this point, we are interested in the following methods: Average, Clear, Contains, Max, Min, Remove, RemoveAt, and Reverse.
- We'll create a console application called Lecture3-Demo4 to test these methods

# Methods of the List class

- Average() returns the average value from a List of numerical Data
- Min() and Max() return respectively the minimum and maximum values from a List.
- Clear() removes all data from the List
- Contains() takes as argument an item and returns true if the List contains this item already. Otherwise, it returns false.
- Remove() takes as argument an item and removes the first occurrence of the item from the List
- RemoveAt() takes as argument an index and removes the element at that index in the List
- Reverse() reverses the order of the elements in the List

# Creating a list of structs

- We can create a list of any of the basic types, but we can also create a list of more complex structures, such as struct types or objects.
- Eg, consider the struct type (Point):

```
struct Point
   {
      public int _X;
      public  int _Y;
    }
```

- We can create a list of Point structs

**List<Point> pointList= new List<Point>()**

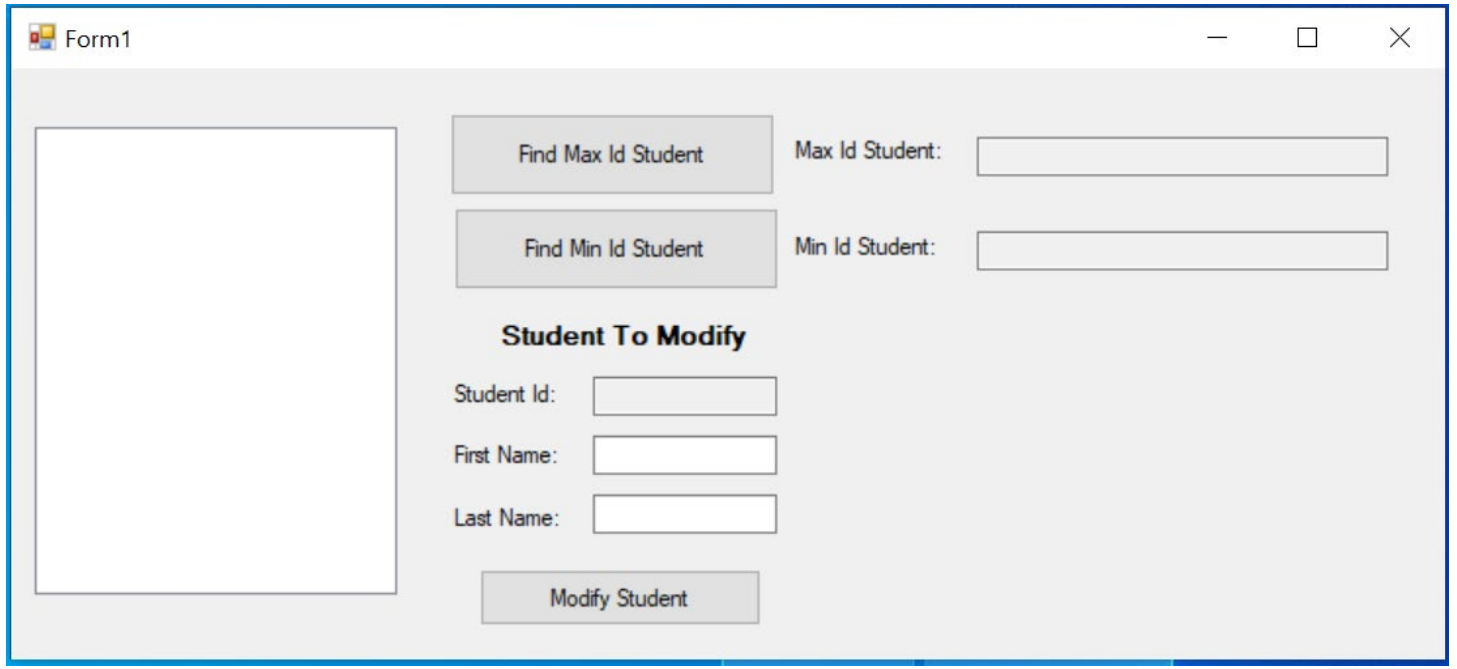
# Creating a list of structs

- With the list defined on the previous slide, we can create Point structs and add to the list
- E.g. List<Point> pointList= new List<Point>()

```
Point P1=new Point{_X=2,_Y=3};
pointList.Add(P1);
```

# Lecture 3 Demo5

- You have been provided with a starter program with the UI shown below.




- To the program, add a struct **Student** having as members a student id (an integer) a string for the student first name and a string for the student's last name. Then follow the instructions in the next slide

# Lecture 3 Demo 5- contd

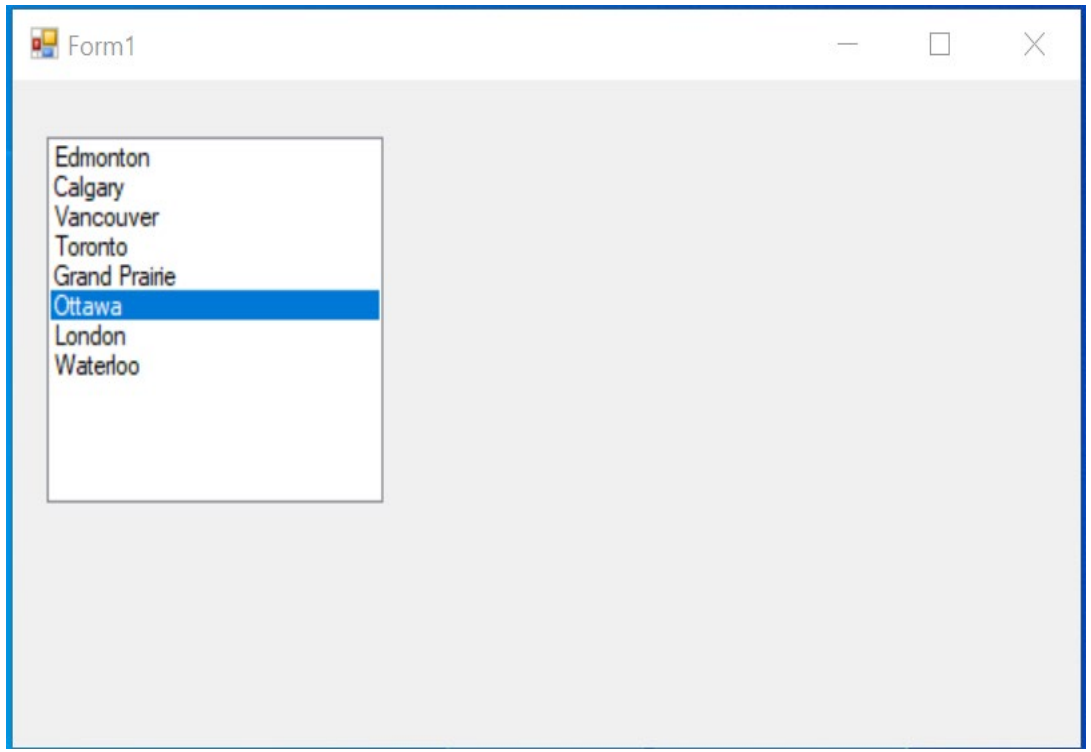
- The program must also contain the following methods:
  - A method **StudentToString**(). It has as parameter a student struct and it returns a string containing the student data as follows: <studentId>, <firstName> <lastName>
  - A method MaxId() that has as parameter a list of Student structs and it returns the Student struct having the highest ID value
  - A method MinId() that has a parameter a list of Student structs and it returns the Student struct with the minimum Id Value
- In the Form class, create an empty List of **Student** structs.
- In the form class create 3 Lists each containing 10 values. One must be for student Ids, one for first names and one for last names.
- Add event handlers as per the next slide.

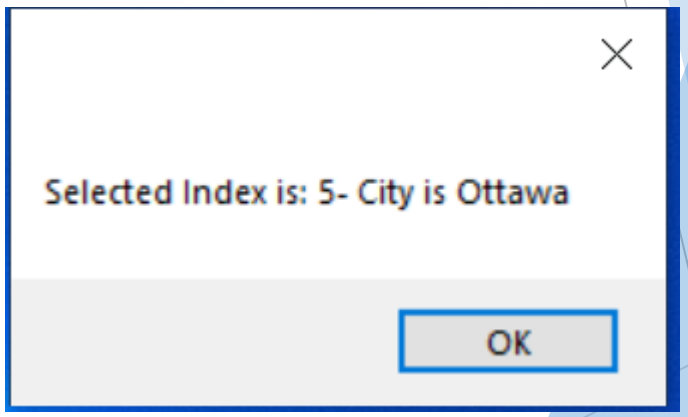
# Lecture 3 Demo 5- contd

- The Form_Load event handler must create 10 Student structs, from the 3 class-level lists and add them to the list of structs. It must then display them in the listbox using the **StudentToString()** method.
- The click event handlers for the “Find Max Id Student” and “Find Min Id Student” buttons must respectively display all information for the student with maximum and minimum id in read-only textboxes next to the button.

# Obtaining the index of a selected element

- We can use the **SelectedIndex** property of the ListBox, to obtain the index of a selected item.
- We also have the **SelectedIndexChanged** event that we can use to trigger an action



# Lecture 3Demo 5B Modifying a list element

- We will now try to modify the list elements
- To Demo5, add a "**SelectedIndexChanged**" event handler for the listbox such that it picks the selected index and display that student information in the textboxes under the "Student To Modify" label.
- The user can modify the information in the 2 non-read-only textboxes. Add a click event handler of the "Modify Student" button such that it modifies the information for the corresponding struct in the list, then redisplay the listbox.

# Modifying members of a List Element

- From the previous demo, we note that when our List element is a struct type, we cannot modify it in-place.
- Instead, we have to create another instance of the struct, then assign it again to the List element.