Lab: Generics, Iterators and Comparators

This document defines the lab overview for the "C# OOP Advanced" course @ Software University. Please submit your solutions (source code) of all below described problems at the end of the course at softuni.bg.

Introduction

In the last few labs all we've done is refactoring **BashSoft** with all kinds of good practices of **OOP** and **OOD** (object oriented design). We have to admit that although necessary, that has been kind of boring. Now that we've learned about **Generics**, **Iterators** and **Comparators** it's a good moment to finally add some new functionality to our project. We will create our own data structure called **SimpleSortedList.** It will have some normal methods for a list but it will be different because the elements inside it will always be sorted. Kind of like a **SortedSet** with repeating elements. Such an abstract data structure is called an **OrderedBag**.

Note that our implementation will be very slow in terms of performance because that is not the focus of the current material. If you are interested in making it fast you may learn about that and much more in the Data Structures course at SoftUni.

Problem 1. Implementing the SimpleSortedList Data Structure

Let's start with the interface so we know what features our data structure should have in case we (or someone else) decide to do a different implementation of it. Call the interface **ISimpleOrderedBag.** The simple part is because we don't plan to add much functionality to it for now. If we do decide to add more functionality we can extend the interface. It is also simple because as mentioned it will not have the best implementation. Our interface should have methods: **add**, **addAll**, **size** and **joinWith** (a method to help printing). It should also keep a **generic type** that is **comparable** and the bag itself should be **iterable.** This is how it should look (don't forget about good formatting of empty lines and spaces):

```
public interface ISimpleOrderedBag<T> : IEnumerable<T> where T : IComparable<T>
{
    void Add(T element);

    void AddAll(ICollection<T> elements);

    int Size { get; }

    String JoinWith(String joiner);
}
```

Now that we've created the interface it's time for the class itself. Let's start with making a new folder called **DataStructures.** In it make a new class called **SimpleSortedList**. We will need three fields - one for keeping the **internal collection** (a generic array), one for holding the size of our list, and one for the comparator of our sorted list. Don't forget to also write the correct class signature. We will also need a constant field to keep a default field for our list.

```
class SimpleSortedList<T> : ISimpleOrderedBag<T> where T : IComparable<T>
{
    private const int DefaultSize = 16;

    private T[] innerCollection;
    private int size;
    private IComparer<T> comparison;
```

Notice how we have unimplemented methods?

Well we will implement them later because we should do the constructors first. We will have **four** of them, so let's do the first two and you can do the others yourself. The first one will accept the most parameters:

```
private void InitializeInnerCollection(int capacity)
{
    if (capacity < 0)
    {
        throw new ArgumentException("Capacity cannot be negative!");
    }

    this.innerCollection = new T[capacity];
}
```

Don't forget to validate the capacity!

The second constructor will have a default comparer:

```
public SimpleSortedList(int capacity)
    : this(Comparer<T>.Create((x, y) => x.CompareTo(y)), capacity)
{
}
```

The **third** will have a **default capacity** and the **fourth** one will have both **capacity and comparer to be default**. Implement them yourself.

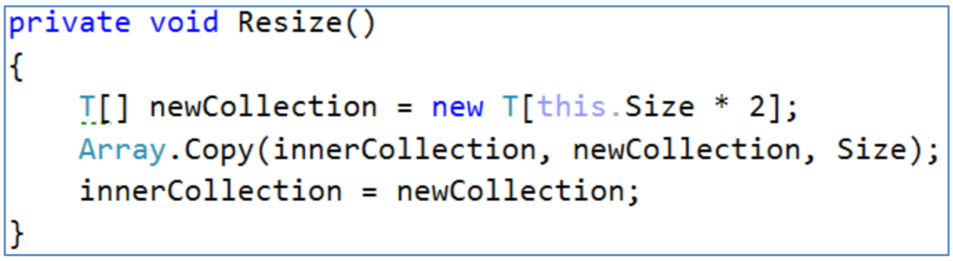
OK, now let's start implementing the F… heck out of those methods! The **Size** property should just return our size field as is:

```
public int Size
{
    get { return this.size; }
}
```

The **add** method, should set the element at the current **size index of our inner collection** to the generic element passed to it. Then **increment the size** and finally **sort the inner collection**, because after all we're creating a **sorted list.**

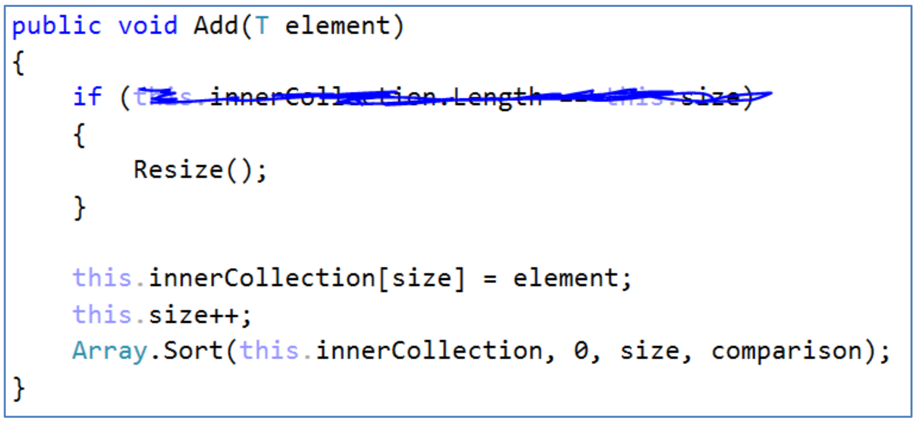
```
this.innerCollection[size] = element;
this.size++;
Array.Sort(this.innerCollection, 0, size, comparison);
```

Well as Nakov says this should work like a dude, except… not always. What happens when our inner array is full? The answer is - **IndexOutOfRangeException.** To prevent this we need to resize our array. We basically need to copy our array into a new one that is twice as big and leave the empty values to be null. The **Arrays.Copy** method can do this for us but let's wrap it in our own private method.



```
private void Resize()
{
    T[] newCollection = new T[this.Size * 2];
    Array.Copy(innerCollection, newCollection, Size);
    innerCollection = newCollection;
}
```

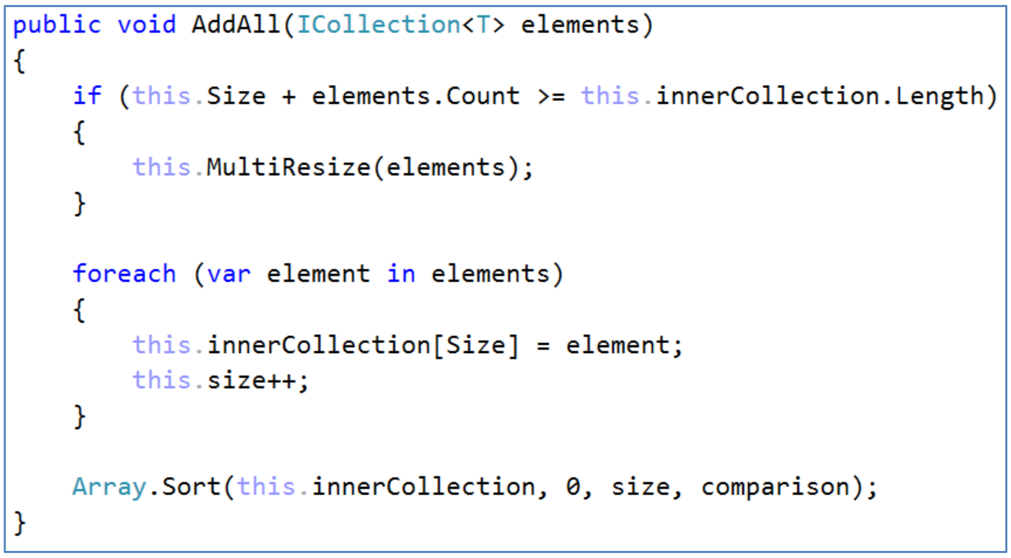
Now all that is left is to call this method when our **size** is more or equal to the inner array's length. This is how the add method should look in the end:



```
public void Add(T element)
{
    if (~~this.innerCollection.Length == this.size)~~
    {
        Resize();
    }

    this.innerCollection[size] = element;
    this.size++;
    Array.Sort(this.innerCollection, 0, size, comparison);
}
```

The **AddAll** method will work in a similar fashion as the **add** method. We could even implement it by calling **add** but that would trigger sorting at each element to be added, so a better approach would be to add all of the elements and only sort at the end:



```
public void AddAll(ICollection<T> elements)
{
    if (this.Size + elements.Count >= this.innerCollection.Length)
    {
        this.MultiResize(elements);
    }

    foreach (var element in elements)
    {
        this.innerCollection[Size] = element;
        this.size++;
    }

    Array.Sort(this.innerCollection, 0, size, comparison);
}
```

However resizing in this case might not be so simple because our current elements + the ones we want to add might be more than the inner collection's length * 2. Thus we will have a slightly different resize approach (think about the logic behind this):

```
private void MultiResize(ICollection<T> elements)
{
    int newSize = this.innerCollection.Length * 2;
    while (this.Size + elements.Count >= newSize)
    {
        newSize *= 2;
    }

    T[] newCollection = new T[newSize];
    Array.Copy(this.innerCollection, newCollection, size);
    this.innerCollection = newCollection;
}
```

Now that we're done with the main functionality it's a good time to override the `IEnumerator<T> GetEnumerator()` method. In it we should create loop through the inner collection and yield return the current element.

```
public IEnumerator<T> GetEnumerator()
{
    for (int i = 0; i < this.Size; i++)
    {
        yield return this.innerCollection[i];
    }
}
```

The last thing we need to implement is the **JoinWith** method. It will connect all the elements in our structure with the given **joiner** string. Since now we have the iterator, we can reuse it.

```
public string JoinWith(string joiner)
{
   StringBuilder builder = new StringBuilder();
    foreach (var element in this)
    {
        builder.Append(element);
        builder.Append(joiner);
    }

    builder.Remove(builder.Length - 1, 1);
    return builder.ToString();
}
```

Problem 2. Making Students and Courses Comparable

As always start by altering the interfaces:

```
public interface Course : IComparable<Course>
```

Do the same for the Student interface. Then go to the classes. This would be a good time to override the **ToString** method of our SoftUniStudent/Course classes too:

```
public int CompareTo(Student other) => String.Compare(this.UserName, other.UserName, StringComparison.Ordinal);

public override string ToString() => this.UserName;
```

Do the same for the **SoftUniCourse** class.

Problem 3. Adding functionality to the StudentsRepository

The methods we are going to add are most akin to the **Requester** interface which the **Database** interface extends. Thus we will add our two new method signatures here:

```
public interface IRequester
{
    void GetStudentScoresFromCourse(string courseName, string username);

    void GetAllStudentsFromCourse(string courseName);

    ISimpleOrderedBag<Course> GetAllCoursesSorted(IComparer<Course> cmp);

    ISimpleOrderedBag<Student> GetAllStudentsSorted(IComparer<Student> cmp);
}
```

Now implement them in the **StudentRepository:**

```
public ISimpleOrderedBag<Student> GetAllStudentsSorted(IComparer<Student> cmp)
{
    SimpleSortedList<Student> sortedStudents = new SimpleSortedList<Student>(cmp);
    sortedStudents.AddAll(students.Values);

    return sortedStudents;
}
```

Notice how we can add the **values** from our **Dictionary** with the **addAll** method of our **SimpleSortedList** because they are a **Collection**. The other one is exactly the same but with **Students,** implement it **yourself**.

Problem 4. Adding the new DisplayCommand

Start off by adding a case to our **CommandInterpreter's parseCommand** method:

```
case "display":
    return new DisplayCommand(input, data, this.judge, this.repository,
                              this.downloadManager,this.inputOutputManager);
```

Then create the matching class with the usual constructor. You can try to figure out an implementation for its execute command on your own. Try to reuse as much code as you can and use our SimpleSortedList's **JoinWith** command for printing.

Alternatively you can look at our implementation. We won't go into much detail about it because it is not the subject of this lab:

```
public override void Execute()
{
    string[] data = this.Data;

    if (data.Length != 3)
    {
        throw new InvalidCommandException(this.Input);
    }

    string entityToDisplay = data[1];
    string sortType = data[2];
    if (entityToDisplay.Equals("students", StringComparison.OrdinalIgnoreCase))
    {
        IComparer<Student> studentComparator = this.CreateStudentComparator(sortType);
        ISimpleOrderedBag<Student> list = this.Repository.GetAllStudentsSorted(studentComparator);
        OutputWriter.WriteMessageOnNewLine(list.JoinWith(Environment.NewLine));
    }
```

```
else if (entityToDisplay.Equals("courses", StringComparison.OrdinalIgnoreCase))
{
    IComparer<Course> courseComparator = this.CreateCourseComparator(sortType);
    ISimpleOrderedBag<Course> list = this.Repository.GetAllCoursesSorted(courseComparator);
    OutputWriter.WriteMessageOnNewLine(list.JoinWith(Environment.NewLine));
}
else
{
    throw new InvalidCommandException(this.Input);
}
```

This one of the **CreateComparator** commands, implement the other **yourself:**

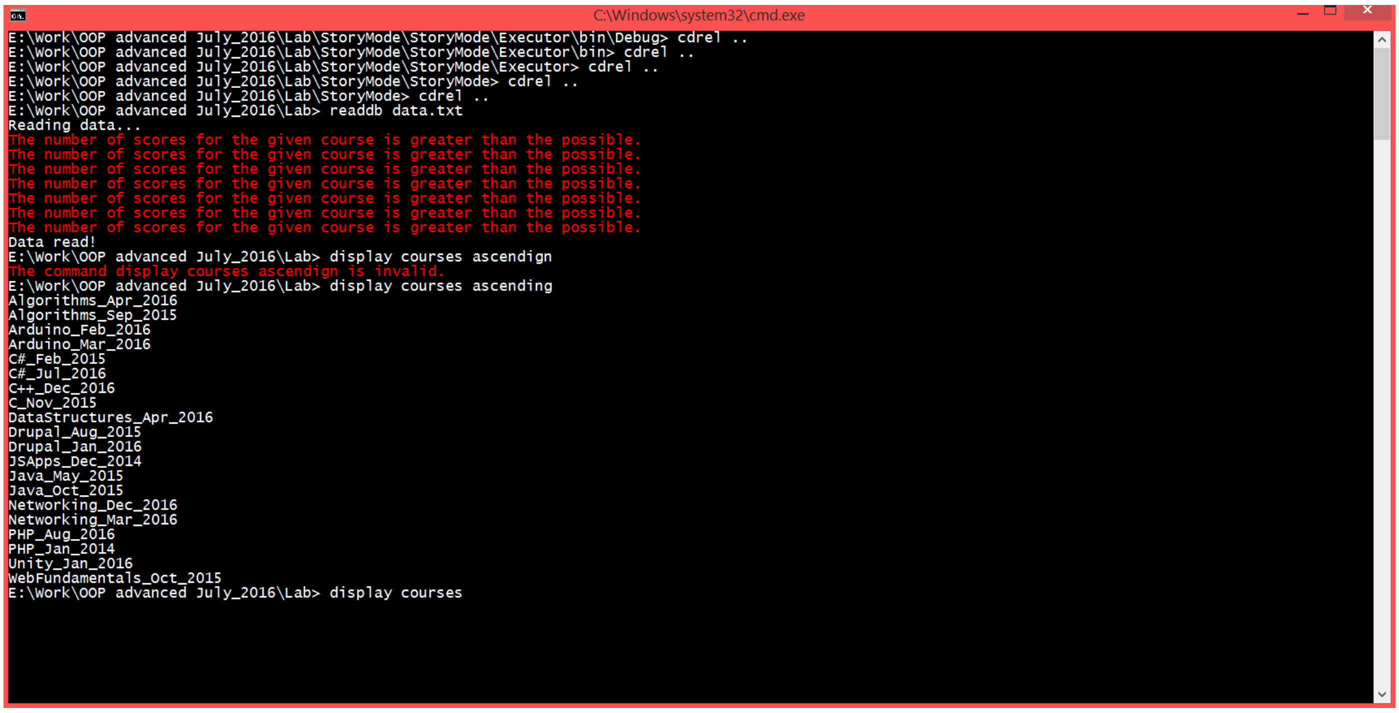
```
private IComparer<Student> CreateStudentComparator(string sortType)
{
    if (sortType.Equals("ascending", StringComparison.OrdinalIgnoreCase))
    {
        return Comparer<Student>.Create((student, student1) => student.CompareTo(student1));
    }
    else if (sortType.Equals("descending", StringComparison.OrdinalIgnoreCase))
    {
        return Comparer<Student>.Create((student, student1) => student1.CompareTo(student));
    }
    else
    {
        throw new InvalidCommandException(this.Input);
    }
}
```

Finally go to the **DisplayHelpCommand** and add some more help info about our new command:

```
stringBuilder.AppendLine(string.Format("|{0, -98}|",
    "display data entities - display students/courses ascending/descending"));
```

Problem 5. Testing our new functionality

If you've implemented all the new functionality correctly you should get such a result:



```
E:\Work\OOP advanced July_2016\Lab\StoryMode\StoryMode\Executor\bin\Debug> cdrel ..
E:\Work\OOP advanced July_2016\Lab\StoryMode\StoryMode\Executor\bin> cdrel ..
E:\Work\OOP advanced July_2016\Lab\StoryMode\StoryMode\Executor> cdrel ..
E:\Work\OOP advanced July_2016\Lab\StoryMode\StoryMode> cdrel ..
E:\Work\OOP advanced July_2016\Lab\StoryMode> cdrel ..
E:\Work\OOP advanced July_2016\Lab> readdb data.txt
Reading data...
The number of scores for the given course is greater than the possible.
The number of scores for the given course is greater than the possible.
The number of scores for the given course is greater than the possible.
The number of scores for the given course is greater than the possible.
The number of scores for the given course is greater than the possible.
The number of scores for the given course is greater than the possible.
The number of scores for the given course is greater than the possible.
Data read!
E:\Work\OOP advanced July_2016\Lab> display courses ascendign
The command display courses ascendign is invalid.
E:\Work\OOP advanced July_2016\Lab> display courses ascending
Algorithms_Apr_2016
Algorithms_Sep_2015
Arduino_Feb_2016
Arduino_Mar_2016
C#_Feb_2015
C#_Jul_2016
C++_Dec_2016
C_Nov_2015
DataStructures_Apr_2016
Drupal_Aug_2015
Drupal_Jan_2016
JSApps_Dec_2014
Java_May_2015
Java_Oct_2015
Networking_Dec_2016
Networking_Mar_2016
PHP_Aug_2016
PHP_Jan_2014
Unity_Jan_2016
WebFundamentals_Oct_2015
E:\Work\OOP advanced July_2016\Lab> display courses
```

Problem 6. * BONUS TASK: Implement your own sorting algorithm for the SimpleSortedList

Instead of using the state **Arrays.sort** method you can make your own **generic** sorting method inside our class. Use one of the following algorithms:

- **BubbleSort (easy difficulty)**
- **SelectionSort (easy difficulty)**
- **InsertionSort (medium difficulty)**
- **QuickSort (hard difficulty)**

You can research about them all around the internet, but here are some more interesting sources:

http://visualgo.net/sorting - great site with visualization of all the algorithms and more + pseudo code. A good programmer should never be **limited** to using just one language and should always be able to read pseudo code.

https://softuni.bg/trainings/resources/video/8437/video-12-april-2016-atanas-rusenov-algorithms-april-2016 - a lecture by **Atanas Rusenov** about sorting algorithms.