Practical Task 2.1

(Pass Task)

Submission deadline: Monday, March 24
Discussion deadline: Friday, April 11

Instructions

The objective of this task is to allow the user to sort elements stored in the Vector<T> data collection, which is the class completed by you in Task 1.1. This task does not need writing too much code; it is to help you to learn the standard interfaces that enable sorting within the .NET Framework. Your task is to apply these interfaces to a sample class and implement the required comparison methods.

The following steps indicate what you need to do.

1. Create a copy of your solution for Task 1.1 and replace the Tester.cs file by that attached to this task. The Main method of the new Tester class is to be compatible with the Vector<T>. Again, it contains a series of tests that will verify if the functionality that you need to develop in this task works correctly. See Appendix A for an example of what the program will output when working correctly.
2. Extend your Vector<T> class by adding the following two overloaded methods:

   - **void Sort( )**

     Sorts the elements in the entire Vector<T> collection. The order of elements is determined by the default comparer for type T, which is Comparer<T>.Default.

   - **void Sort(IComparer<T> comparer)**

     Sorts the elements in the entire Vector<T> collection using the specified comparer. If the **comparer** is null, the order of elements is determined by the default comparer for type T, which is Comparer<T>.Default.

   In this task, the implementation of both methods does not require you to implement any sorting algorithm. Instead, you should delegate sorting of data elements to Array.Sort, which implements the Introspective sorting algorithm as the sorting method for the Array class. Similar to the two methods above, Array.Sort is overloaded and has two versions: One that accepts an array along with the starting index and the number of elements to define the range for sorting, and the other, which additionally accepts a comparer object for the purpose of determining the order of elements. Read about these two methods at

   https://learn.microsoft.com/en-us/dotnet/api/system.array?redirectedfrom=MSDN&view=net-7.0#methods

   Once you have your both Sort methods set up, you should be able to sort integers and succeed with tests A, B, C, and D in the attached Tester class. To enable these tests, uncomment the respective code lines in Tester.cs file.
3. Explore the Student class provided in Tester.cs file. To make two students comparable (and thus sorted in the Vector<T> collection like in the List<T>), you must enable a mechanism that automatically compares two elements of type Student. Keep in mind that in this case T is a blueprint type, which during the runtime will be replaced by the Student class.
4. One way to make two elements comparable in the .NET Framework is to introduce a comparison method right in the type, for example, in the Student class. This is how we can 'teach' the type to compare one element (referred to as **this**) with the other element (referred to as **another**) of the same type. In the code, this should be done by implementing the special IComparable<T> interface in the class, which in its turn requires implementation of the CompareTo(T *another*) method that compares **this** (current) element with **another** element and returns an integer value. The integer values is

   - **less than zero**, when **this** (current) element must precede **another** element specified as the argument,

- **zero**, when **this** (current) element is seen as equal to **another** element,
- **greater than zero**, when **this** (current) element must succeed **another** element.

Your task now is to modify the Student class and implement the IComparable<Student> interface along with its underlying CompareTo(Student another) method so that it can sequence students in the ascending order of the ID attribute. When this is done, you should be able to sort objects of Student class via the Sort( ) method of the Vector<T> class. Complete this part to pass Test E by activating its respective code lines in the program code.

5. The other way to make two elements comparable is to develop a so-called comparator class, whose purpose is to compare a pair of elements of specific type. Such comparator class must implement the IComparer<T> interface. The IComparable<T> is similar to the IComparer<T>, except that the CompareTo(T another) method belongs to the type, while Compare(T a, T b) is part of an independent class implementing the IComparer<T>. The Compare(T **a**, T **b**) method implied by the Comparer<T> returns an integer value indicating whether argument **a** is less than, equal to, or greater than argument **b**, exactly as the CompareTo method does.

   There are a few examples of classes implementing the IComparer<int> interface in Tester.cs file that sort integers in ascending, descending, or in the order that places even integers prior to odd integers. You should learn from these examples to develop your own comparator for the Student class. Specifically, your task is to provide two comparators for the Student class:

   - **AscendingIDComparer** to sort a sequence of students stored in an instance of the Vector<Student> class in the ascending order of IDs;
   - **DescendingNameDescendingIdComparer** to sort a sequence of students stored in an instance of the Vector<Student> class in the descending order of names, breaking ties by the descending order of IDs.

   Once this is done, you should be able to pass tests F and G. Make sure that students appear in the desired order. Keep in mind that any of these comparators can be the argument of the Sort(IComparer<T> comparer) method of the Vector<T> class; so, test how they work with your Vector<Student> class.

6. Submit the new version of the Vector<T> class and the updated Tester.cs file that now constains the extended Student class and both AscendingIDComparer and DescendingNameDescendingIdComparer classes.

Further Notes

- Read Chapter 1.7 of SIT221 Workbook available in CloudDeakin in Content → Learning Resources → SIT221 Workbook. It provides a brief discussion of the IComparable<T> interface.
- If you still struggle with such OOP concept as Generics, you may wish to read Chapter 11 of SIT232 Workbook available in Content → Learning Resources → SIT232 Workbook. You may also have to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you will need an excellent understanding of these topics in order to progress well through the practical tasks of the unit. Make sure that you are proficient with them as they form a basis to design and develop programming modules in this and all subsequent tasks.
- We will test your code using the .NET Core 6.0. You are free to use any IDE (for example, Visual Studio Code), though we recommend you work from Microsoft Visual Studio 2022 due to its simple installation process and good support of debugging. You can find the necessary instructions and installation links by navigating in CloudDeakin to Content → Learning Resources → Software.

Submission Instructions and Marking Process

To get your task completed, you must finish the following steps strictly on time.

- Make sure your programs implement the required functionality. They must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your programs thoroughly before submission. Think about potential errors where your programs might fail.
- **Submit** the expected code files as a solution to the task via OnTrack submission system. You must **record a short video explaining your solution** to the task. Remember to explain the implementation issues of the IComparer<T> and the IComparable<T> interfaces and the difference between them. Upload the video to one of accessible resources and refer to it for the purpose of marking. You must provide a private working link to the video to your marking tutor in OnTrack. Note that the video recording must be made in the **camera on mode**; that is, the tutor must see both the presenter and the shared screen.
- Once your solution is accepted by the tutor, you will be invited to **continue its discussion and answer relevant theoretical questions** through the Intelligent Discussion Service in OnTrack. Your tutor will record several audio questions. When you click on these, OnTrack will record your response live. You must answer straight away in your own words. As this is a live response, you should ensure you understand the solution to the task you submitted. Answer all additional questions that your tutor may ask you. Questions will cover the lecture notes; so, attending (or watching) the lectures should help you with this **compulsory discussion part**. You should start the discussion as soon as possible as if your answers are wrong, you may have to pass another round, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after **the submission deadline** and will not discuss it after **the discussion deadline**. If you fail one of the deadlines, you fail the task, and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When grading your achievements at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and the quality of your solutions.

Appendix A: Expected Printout

The following provides an example of the output generated from the testing module (Tester.cs) once you have correctly implemented all methods of the Vector<T> class.

```
Test A: Run a sequence of operations:
Create a new vector by calling 'Vector<int> vector = new Vector<int>(50);'
Add a sequence of numbers 2, 6, 8, 5, 5, 1, 8, 5, 3, 5, 7, 1, 4, 9
Sort the integers in the default order defined by the native CompareTo() method
Resulting order: [1,1,2,3,4,5,5,5,5,6,7,8,8,9]
 :: SUCCESS

Test B: Run a sequence of operations:
Create a new vector by calling 'Vector<int> vector = new Vector<int>(50);'
Add a sequence of numbers 2, 6, 8, 5, 5, 1, 8, 5, 3, 5, 7, 1, 4, 9
Sort the integers in the order defined by the AscendingIntComparer class
Resulting order: [1,1,2,3,4,5,5,5,5,6,7,8,8,9]
 :: SUCCESS

Test C: Run a sequence of operations:
Create a new vector by calling 'Vector<int> vector = new Vector<int>(50);'
Add a sequence of numbers 2, 6, 8, 5, 5, 1, 8, 5, 3, 5, 7, 1, 4, 9
Sort the integers in the order defined by the DescendingIntComparer class
```

```
Resulting order: [9,8,8,7,6,5,5,5,5,4,3,2,1,1]
 :: SUCCESS

Test D: Run a sequence of operations:
Create a new vector by calling 'Vector<int> vector = new Vector<int>(50);'
Add a sequence of numbers 2, 6, 8, 5, 5, 1, 8, 5, 3, 5, 7, 1, 4, 9
Sort the integers in the order defined by the EvenNumberFirstComparer class
Resulting order: [2,6,8,8,4,5,5,1,5,3,5,7,1,9]
 :: SUCCESS

Test E: Run a sequence of operations:
Create a new vector of Student objects by calling 'Vector<Student> students = new Vector<Student>();'
Add student with record: 0[Vicky]
Add student with record: 1[Cindy]
Add student with record: 2[Tom]
Add student with record: 3[Simon]
Add student with record: 4[Richard]
Add student with record: 5[Vicky]
Add student with record: 6[Tom]
Add student with record: 7[Elicia]
Add student with record: 8[Richard]
Add student with record: 9[Cindy]
Add student with record: 10[Vicky]
Add student with record: 11[Guy]
Add student with record: 12[Richard]
Add student with record: 13[Michael]
Sort the students in the default order defined by the native CompareTo() method
Print the vector of students via students.ToString();
[0[Vicky],1[Cindy],2[Tom],3[Simon],4[Richard],5[Vicky],6[Tom],7[Elicia],8[Richard],9[Cindy],10[Vicky],11[Guy],12[Richard],13[Michael]]
 :: SUCCESS

Test F: Run a sequence of operations:
Create a new vector of Student objects by calling 'Vector<Student> students = new Vector<Student>();'
Add student with record: 0[Vicky]
Add student with record: 1[Cindy]
Add student with record: 2[Tom]
Add student with record: 3[Simon]
Add student with record: 4[Richard]
Add student with record: 5[Vicky]
Add student with record: 6[Tom]
Add student with record: 7[Elicia]
Add student with record: 8[Richard]
Add student with record: 9[Cindy]
Add student with record: 10[Vicky]
Add student with record: 11[Guy]
Add student with record: 12[Richard]
Add student with record: 13[Michael]
```

```
Sort the students in the order defined by the AscendingIDComparer class
Print the vector of students via students.ToString();
[0[Vicky],1[Cindy],2[Tom],3[Simon],4[Richard],5[Vicky],6[Tom],7[Elicia],8[Richard],9[Cindy],10[Vicky],11[Guy],12[Richard],13[Michael]]
 :: SUCCESS

Test G: Run a sequence of operations:
Create a new vector of Student objects by calling 'Vector<Student> students = new Vector<Student>();'
Add student with record: 0[Vicky]
Add student with record: 1[Cindy]
Add student with record: 2[Tom]
Add student with record: 3[Simon]
Add student with record: 4[Richard]
Add student with record: 5[Vicky]
Add student with record: 6[Tom]
Add student with record: 7[Elicia]
Add student with record: 8[Richard]
Add student with record: 9[Cindy]
Add student with record: 10[Vicky]
Add student with record: 11[Guy]
Add student with record: 12[Richard]
Add student with record: 13[Michael]
Sort the students in the order defined by the DescendingNameDescendingIdComparer class
Print the vector of students via students.ToString();
[10[Vicky],5[Vicky],0[Vicky],6[Tom],2[Tom],3[Simon],12[Richard],8[Richard],4[Richard],13[Michael],11[Guy],7[Elicia],9[Cindy],1[Cindy]]
 :: SUCCESS



 ---------- SUMMARY ----------
Tests passed: ABCDEFG
```