**IComparable and IComparer interface in C#**

* IComparable and IComparer interface is used when a class has a data member as an array of object of any other class.
* For the ordered data types like numbers or strings, comparison can be done easily.
* As there can be many different data membersfor the object, comparison of two objects cannot be done directly.
* In that situation, that class, whose objects are stored in an array, should implement IComparable and/or IComparer interface.
* While implementing methods of IComparable or IComparer, you can choose any one of the fields for comparison, as per business requirement.
* The same implemented method will be used for sorting of the array or collection.

### IComparable Interface:

* This interface is used to sort elements and compare the current instance with another object of same type.
* Method in IComparable interface is System.IComparable.CompareTo(System.Object)
* The CompareTo method returns an int value that shows how two elements are related.

there are three common ways to check the orders of class fields:

* For string fields you can use the String.Compare method to check for alphabetical order.
* For numeric fields you can use the ==, > and < operators.
* For string fields that have a non-alphabetical order, a good strategy is to write a pair of helper methods such as the RegionCompare and RegionValue methods

public interface IComparable

{

int CompareTo(object obj);

}

|  |  |
| --- | --- |
| Value | Meaning |
|  |  |
| Less than zero | The current instance precedes the object specified by the CompareTo method in the sort order. |
| Zero | This current instance occurs in the same position in the sort order as the object specified by the CompareTo method. |
| Greater than zero | This current instance follows the object specified by the CompareTo method in the sort order. |

Let’s look at the below example.  
For example, if you are creating an Order class that has a DateTime Created property that you want to sort on, you can implement IComparable on the Order class and compare the Created dates of both orders.

class Order : IComparable

{

    public DateTime Created { get; set; }

public int CompareTo(object obj)

    {

        if (obj == null) return 1;

        Order o = obj as Order;

    if (o == null)

        {

             throw new ArgumentException(“Object is not an Order”);

        }

        return this.Created.CompareTo(o.Created);

     }

}

   List<Order> orders = new List<Order>

    {

        new Order { Created = new DateTime(2015, 12, 1 )},

        new Order { Created = new DateTime(2015, 1, 6 )},

        new Order { Created = new DateTime(2015, 7, 8 )},

        new Order { Created = new DateTime(2016, 2, 20 )},

    };

    orders.Sort();

}

}

The call to orders.Sort() calls the CompareTo method to sort the items. After sorting, the list contains the ordered Orders.  
IComparable also has a generic version: IComparable. Especially when dealing with methods from the .NET Framework, it’s a good idea to implement both IComparable and IComparable.

**Scenario 2: More than One Sort Order**  
To define a class that can be sorted in two or more ways and that has no default ordering, you should use the IComparer interface.

Notice that the City class doesn't directly inherit from either IComparable or IComparer. The City class in [**Listing 4**](https://visualstudiomagazine.com/articles/2011/10/01/multilevel-sorting-with-icomparable-and-icomparer/listing4.aspx) has three sort order routines, each of which is defined in a nested subclass. Each nested subclass inherits from the IComparer interface and implements a Compare method that defines a sort order.

Sorting a collection of City class objects as defined in [**Listing 4**](https://visualstudiomagazine.com/articles/2011/10/01/multilevel-sorting-with-icomparable-and-icomparer/listing4.aspx) could look like:

List<City> list = new List<City>();

list.Add(new City("san diego", 1.2, "west"));

list.Add(new City("phoenix", 1.7, "west"));

(etc.)

list.Sort(new City.SortByNameAscending());

foreach (City c in list) { Console.WriteLine(c.ToString()); }

The List.Sort method is passed an instance of one of the subclasses that defines a sort order, SortByNameAscending in this case; this ordering is used to sort the list collection. If the Compare method in SortByNameAscending is coded so that City objects are ordered solely by name in alphabetical order without regard to population or region, the output produced by the foreach statement would be:

boston 1.0 east

chicago 1.2 north

houston 1.7 west

nashville 1.2 south

phoenix 1.7 west

san diego 1.2 west

The SortByNameAscending subclass is defined as:

public class SortByNameAscending : IComparer<City>

{

public int Compare(City c1, City c2)

{

return String.Compare(c1.name, c2.name);

}

}

Unlike IComparable CompareTo, which accepts one parameter, the IComparer Compare method accepts two parameters.

Compare returns a value of 0 if the two parameters are equal with regard to sort order, a value greater than 0 (usually +1) if the left parameter comes after the right parameter in sort order, and a value less than 0 (usually -1) if the left parameter comes before the right parameter in sort order. I could've coded the Compare method like so:

if (c1.name == c2.name) return 0;

else if (c1.name > c2.name) return 1;

else if (c1.name < c2.name) return -1;

But because this is exactly how String.Compare works, I can just call String.Compare.

The SortByNameDescending subclass looks like this:

public class SortByNameDescending : IComparer<City>

{

public int Compare(City c1, City c2)

{

return -1 \* String.Compare(c1.name, c2.name);

}

}

Returning -1 times an ascending order statement is a standard trick when coding a descending order method. The SortByPopulationName subclass is given in [**Listing 5**](https://visualstudiomagazine.com/articles/2011/10/01/multilevel-sorting-with-icomparable-and-icomparer/listing5.aspx).

The SortByPopulationName subclass houses a Compare method that orders first by city population (ascending), then by city name (alphabetical order). You can implement as many non-default sort-ordering subclasses as you wish. For a class with three data fields, if using both ascending and descending orders on all fields there are a total of 78 distinct sort orderings.

**Scenario 3: Multiple Sort Orders and a Default Order**  
If you want to define a class that has a default sort ordering, and also one or more additional sort orders, you should inherit the class from the IComparable interface, implement CompareTo in the class, and define nested subclasses, each of which inherits from IComparer and implements the Compare method. The outline for a City class that has both a default sort order and three non-default sort orders is shown in [**Listing 6**](https://visualstudiomagazine.com/articles/2011/10/01/multilevel-sorting-with-icomparable-and-icomparer/listing6.aspx).

There's no new code in [**Listing 6**](https://visualstudiomagazine.com/articles/2011/10/01/multilevel-sorting-with-icomparable-and-icomparer/listing6.aspx) that didn't appear in the Scenario 1 and Scenario 2 code. Essentially, the code for the two scenarios is simply combined.

I did make one small organizational change: I created a wrapper subclass named SortRoutines to hold the non-default sort order subclasses. As a rule of thumb, if my class has three or more sort subclasses, I create a wrapper class for modularity and improved code readability. Using the City class defined in [**Listing 6**](https://visualstudiomagazine.com/articles/2011/10/01/multilevel-sorting-with-icomparable-and-icomparer/listing6.aspx) could look like this:

List<City> list = new List<City>();

list.Add(new City("san diego", 1.2, "west"));

(etc.)

list.Sort();

foreach (City c in list) { Console.WriteLine(c.ToString()); }

list.Sort(new City.SortRoutines.SortByNameDescending());

foreach (City c in list) { Console.WriteLine(c.ToString()); }

After instantiating and loading the list of City objects, you can use the default sort ordering by calling the Sort method without an argument, or you can use one of the non-default sort orders by passing an instance of the appropriate subclass to Sort.

**Putting It Together**  
You've seen examples of how to deal with three common object sorting scenarios: a default sort order, multiple sort orderings with no default order, and multiple sort ordering plus a default order. Understanding these three scenarios will allow you to handle other scenarios.

Consider Scenario 1 again. The code example derived a City class from the IComparable interface and implemented the required CompareTo method inside the class. A perfectly reasonable alternative is to define a single subclass that inherits from IComparer, which houses and implements the Compare method, and pass that subclass to Sort.

In short, using a nested subclass that inherits from IComparer and implements the Compare method is the most general approach to object sorting. I presented to you a useful pattern for multilevel comparisons where you first check for equality (returning 0), then use n checks for greater than (returning +1) and then fall to a return of -1. Happy object sorting!

**IComparable** and **IComparer** both interfaces are used to sort the list of items. Before analysing these two interfaces first we try to understand the need for these interfaces.  
There is one method named **Sort()** which is used to sort the collection of items. This method is worked very well with predefined data types.  
Suppose we have a list of integers and we try to sort this list by using the Sort method.

class Program

{

static void Main(string[] args)

{

List<int> numbers = new List<int>() { 5, 9, 1, 4, 8, 2, 6 };

numbers.Sort();

Console.WriteLine("List of numbers after sorting:");

foreach(int item in numbers)

{

Console.WriteLine(item);

}

Console.Read();

}

}

**Output**

List of numbers after sorting:

1

2

4

5

6

8

9

So our Sort method working fine with predefined data types. Now we check whether the Sort method will work with user-defined data types or not.  
Suppose we have a list of employees and we try to sort this list by using the Sort method.

public class Employee

{

public string Emp\_Name { get; set; }

public string Emp\_Department { get; set; }

public int Emp\_Salary { get; set; }

}

class Program

{

static void Main(string[] args)

{

List<Employee> employees = GetEmployeeList();

employees.Sort();

Console.WriteLine("Employee list after sorting:");

foreach (Employee employee in employees)

{

Console.WriteLine($"Employee Name: {employee.Emp\_Name}, Employee Department: {employee.Emp\_Department}, " +

$"Employee Salary: {employee.Emp\_Salary}");

}

Console.Read();

}

/// <summary>

/// Method to get employee list

/// </summary>

/// <returns></returns>

private static List<Employee> GetEmployeeList()

{

List<Employee> employees = new List<Employee>();

employees.Add(new Employee() { Emp\_Name = "Rahul", Emp\_Salary = 70000, Emp\_Department = "IT" });

employees.Add(new Employee() { Emp\_Name = "Amit", Emp\_Salary = 20000, Emp\_Department = "HR" });

employees.Add(new Employee() { Emp\_Name = "Suresh", Emp\_Salary = 40000, Emp\_Department = "IT" });

employees.Add(new Employee() { Emp\_Name = "Nitish", Emp\_Salary = 80000, Emp\_Department = "Account" });

return employees;

}

}

**Output**

System.InvalidOperationException: 'Failed to compare two elements in the array.'

Inner Exception

ArgumentException: At least one object must implement IComparable.

So Sort method is not worked with user-defined data types. To make the Sort method worked with user-defined data types, we need to override the Sort method for the Employee class.   
For this purpose, IComparable and IComparer interfaces are used. Now we try to understand both interfaces one by one.

**IComparable**

IComparable interface is used to compare two objects. This interface contains the CompareTo method, which we need to implement in our class. We have to implement custom sorting for Employee object within this method.

public class Employee : IComparable

{

public string Emp\_Name { get; set; }

public string Emp\_Department { get; set; }

public int Emp\_Salary { get; set; }

//This method return int value based on comparision

//If current value is greater than next value, it return 1 (swap both object)

//If current value is smaller than next value, it return -1 (do not swap)

//If current value and next value are same, it return 0 (do nothing)

public int CompareTo(object obj)

{

if(obj == null)

{

return 1;

}

else

{

Employee nextEmployeeObj = obj as Employee;

if(nextEmployeeObj != null)

{

return Emp\_Name.CompareTo(nextEmployeeObj.Emp\_Name); //Here we done sorting based on employee name

}

else

{

throw new ArgumentException("Object is not proper.");

}

}

}

}

class Program

{

static void Main(string[] args)

{

List<Employee> employees = GetEmployeeList();

employees.Sort();

Console.WriteLine("Employee list after sorting:");

foreach (Employee employee in employees)

{

Console.WriteLine($"Employee Name: {employee.Emp\_Name}, Employee Department: {employee.Emp\_Department}, " +

$"Employee Salary: {employee.Emp\_Salary}");

}

Console.Read();

}

/// <summary>

/// Method to get employee list

/// </summary>

/// <returns></returns>

private static List<Employee> GetEmployeeList()

{

List<Employee> employees = new List<Employee>();

employees.Add(new Employee() { Emp\_Name = "Rahul", Emp\_Salary = 70000, Emp\_Department = "IT" });

employees.Add(new Employee() { Emp\_Name = "Amit", Emp\_Salary = 20000, Emp\_Department = "HR" });

employees.Add(new Employee() { Emp\_Name = "Suresh", Emp\_Salary = 40000, Emp\_Department = "IT" });

employees.Add(new Employee() { Emp\_Name = "Nitish", Emp\_Salary = 80000, Emp\_Department = "Account" });

return employees;

}

}

**Output**

Employee list after sorting:

Employee Name: Amit, Employee Department: HR, Employee Salary: 20000

Employee Name: Nitish, Employee Department: Account, Employee Salary: 80000

Employee Name: Rahul, Employee Department: IT, Employee Salary: 70000

Employee Name: Suresh, Employee Department: IT, Employee Salary: 40000

**IComparer**

### IComparer Interface:

* This interface is used to sort elements
* Method in IComparable interface is System.IComparer.Compare(System.Object,System.Object)
* This method compare two objects and returns a value indicating whether one is less than, equal to or greater than other
* Returns zero if both are same
* Less than zero if first object is less than zero
* Greater than zero if first object is greater than zero

Let’s look at the below example:

class Shape

    {

        public int id;

    }

    class Rectangle : Shape

    {

    }

    class ShapeComparer : IComparer<Shape>

    {

        public int Compare(Shape x, Shape y)

        {

            return x.id.CompareTo(y.id);

        }

    }

In your code, the following code snippet should intuitively work because anymethod that accepts a Shape should also accept a Rectangle:

ShapeComparer shapeComparer = new ShapeComparer();

IComparer<Rectangle> irc = shapeComparer;

Sorting on user-defined data types are done very easily with the IComparable interface but there is one problem in the IComparable interface. IComparable interface needs to update user-defined classes for implementing sorting functionality. Think that, if we want to sort the list of user-defined classes and this class is a third class library and we can not update. Meaning that we can not change the implementation of that class. In this case, the IComparable interface can not be used, here the IComparer interface will be used.

public class Employee

{

public string Emp\_Name { get; set; }

public string Emp\_Department { get; set; }

public int Emp\_Salary { get; set; }

}

//This class is used to sort employee by name

public class SortEmployeeByName : IComparer<Employee>

{

//This method return int value based on comparision

//If current value is greater than next value, it return 1 (swap both object)

//If current value is smaller than next value, it return -1 (do not swap)

//If current value and next value are same, it return 0 (do nothing)

public int Compare(Employee x, Employee y)

{

return x.Emp\_Name.CompareTo(y.Emp\_Name);

}

}

//This class is used to sort employee by salary

public class SortEmployeeBySalary : IComparer<Employee>

{

//This method return int value based on comparision

//If current value is greater than next value, it return 1 (swap both object)

//If current value is smaller than next value, it return -1 (do not swap)

//If current value and next value are same, it return 0 (do nothing)

public int Compare(Employee x, Employee y)

{

return x.Emp\_Salary.CompareTo(y.Emp\_Salary);

}

}

class Program

{

static void Main(string[] args)

{

List<Employee> employees = GetEmployeeList();

SortEmployeeByName sortEmployeeByName = new SortEmployeeByName();

employees.Sort(sortEmployeeByName);

Console.WriteLine("Employee list after sorting (By Name):");

foreach (Employee employee in employees)

{

Console.WriteLine($"Employee Name: {employee.Emp\_Name}, Employee Department: {employee.Emp\_Department}, " +

$"Employee Salary: {employee.Emp\_Salary}");

}

SortEmployeeBySalary sortEmployeeBySalary = new SortEmployeeBySalary();

employees.Sort(sortEmployeeBySalary);

Console.WriteLine("Employee list after sorting (By Salary):");

foreach (Employee employee in employees)

{

Console.WriteLine($"Employee Name: {employee.Emp\_Name}, Employee Department: {employee.Emp\_Department}, " +

$"Employee Salary: {employee.Emp\_Salary}");

}

Console.Read();

}

/// <summary>

/// Method to get employee list

/// </summary>

/// <returns></returns>

private static List<Employee> GetEmployeeList()

{

List<Employee> employees = new List<Employee>();

employees.Add(new Employee() { Emp\_Name = "Rahul", Emp\_Salary = 70000, Emp\_Department = "IT" });

employees.Add(new Employee() { Emp\_Name = "Amit", Emp\_Salary = 20000, Emp\_Department = "HR" });

employees.Add(new Employee() { Emp\_Name = "Suresh", Emp\_Salary = 40000, Emp\_Department = "IT" });

employees.Add(new Employee() { Emp\_Name = "Nitish", Emp\_Salary = 80000, Emp\_Department = "Account" });

return employees;

}

}

**Output**

Employee list after sorting (By Name):

Employee Name: Amit, Employee Department: HR, Employee Salary: 20000

Employee Name: Nitish, Employee Department: Account, Employee Salary: 80000

Employee Name: Rahul, Employee Department: IT, Employee Salary: 70000

Employee Name: Suresh, Employee Department: IT, Employee Salary: 40000

Employee list after sorting (By Salary):

Employee Name: Amit, Employee Department: HR, Employee Salary: 20000

Employee Name: Suresh, Employee Department: IT, Employee Salary: 40000

Employee Name: Rahul, Employee Department: IT, Employee Salary: 70000

Employee Name: Nitish, Employee Department: Account, Employee Salary: 80000

In .NET 4, IComparer<T> has been changed to IComparer<in T>, which means that objects of type T are used only as input parameters. Therefore, an  
object implementing this interface can be assigned to interfaces of a more derived type. This is called contravariance.

### When to use IComparable and Comparer Interface?

* While searching and sorting elements in collection,comparision of elements is needed
* The comparisions can be between elements of collection or between an element and specified value
* Methods used for comparision are Equal, IComparable.CompareTo and IComparer.Compare