Comparisons and sorts within collections

Article • 09/15/2021 • 7 minutes to read

The System.Collections classes perform comparisons in almost all the processes involved in managing collections, whether searching for the element to remove or returning the value of a key-and-value pair.

Collections typically utilize an equality comparer and/or an ordering comparer. Two constructs are used for comparisons.

Check for equality

Methods such as `Contains`, IndexOf, LastIndexOf, and `Remove` use an equality comparer for the collection elements. If the collection is generic, then items are compared for equality according to the following guidelines:

- If type T implements the IEquatable<T> generic interface, then the equality comparer is the Equals method of that interface.
- If type T does not implement IEquatable<T>, Object.Equals is used.

In addition, some constructor overloads for dictionary collections accept an IEqualityComparer<T> implementation, which is used to compare keys for equality. For an example, see the Dictionary<TKey,TValue> constructor.

Determine sort order

Methods such as `BinarySearch` and `Sort` use an ordering comparer for the collection elements. The comparisons can be between elements of the collection, or between an element and a specified value. For comparing objects, there is the concept of a `default comparer` and an `explicit comparer`.

The default comparer relies on at least one of the objects being compared to implement the **IComparable** interface. It is a good practice to implement **IComparable** on all classes are used as values in a list collection or as keys in a dictionary collection. For a generic collection, equality comparison is determined according to the following:

- If type T implements the System.IComparable<T> generic interface, then the default comparer is the IComparable<T>.CompareTo(T) method of that interface
- If type T implements the non-generic System.IComparable interface, then the default comparer is the IComparable.CompareTo(Object) method of that interface.
- If type T doesn't implement either interface, then there is no default comparer, and a comparer or comparison delegate must be provided explicitly.

To provide explicit comparisons, some methods accept an **IComparer** implementation as a parameter. For example, the List<T>.Sort method accepts an System.Collections.Generic.IComparer<T> implementation.

The current culture setting of the system can affect the comparisons and sorts within a collection. By default, the comparisons and sorts in the **Collections** classes are culture-sensitive. To ignore the culture setting and therefore obtain consistent comparison and sorting results, use the InvariantCulture with member overloads that accept a CultureInfo. For more information, see Perform culture-insensitive string operations in collections and Perform culture-insensitive string operations in arrays.

Equality and sort example

The following code demonstrates an implementation of IEquatable<T> and IComparable<T> on a simple business object. In addition, when the object is stored in a list and sorted, you will see that calling the Sort() method results in the use of the default comparer for the `Part` type, and the Sort(Comparison<T>) method implemented by using an anonymous method.

C#

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

// Simple business object. A PartId is used to identify the
// type of part but the part name can change.
public class Part : IEquatable<Part>, IComparable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString() =>
        $"ID: {PartId}   Name: {PartName}";
```

```
    public override bool Equals(object obj) =>
        (obj is Part part)
                ? Equals(part)
                : false;

    public int SortByNameAscending(string name1, string name2) =>
        name1?.CompareTo(name2) ?? 1;

    // Default comparer for Part type.
    // A null value means that this object is greater.
    public int CompareTo(Part comparePart) =>
        comparePart == null ? 1 : PartId.CompareTo(comparePart.PartId);

    public override int GetHashCode() => PartId;

    public bool Equals(Part other) =>
        other is null ? false : PartId.Equals(other.PartId);

    // Should also override == and != operators.
}

public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        var parts = new List<Part>
        {
            // Add parts to the list.
            new Part { PartName = "regular seat", PartId = 1434 },
            new Part { PartName = "crank arm", PartId = 1234 },
            new Part { PartName = "shift lever", PartId = 1634 },
            // Name intentionally left null.
            new Part { PartId = 1334 },
            new Part { PartName = "banana seat", PartId = 1444 },
            new Part { PartName = "cassette", PartId = 1534 }
        };

        // Write out the parts in the list. This will call the overridden
        // ToString method in the Part class.
        Console.WriteLine("\nBefore sort:");
        parts.ForEach(Console.WriteLine);

        // Call Sort on the list. This will use the
        // default comparer, which is the Compare method
        // implemented on Part.
        parts.Sort();

        Console.WriteLine("\nAfter sort by part number:");
        parts.ForEach(Console.WriteLine);
```

```
        // This shows calling the Sort(Comparison<T> comparison) overload us-
ing
        // a lambda expression as the Comparison<T> delegate.
        // This method treats null as the lesser of two values.
        parts.Sort((Part x, Part y) =>
            x.PartName == null && y.PartName == null
                ? 0
                : x.PartName == null
                    ? -1
                    : y.PartName == null
                        ? 1
                        : x.PartName.CompareTo(y.PartName));

        Console.WriteLine("\nAfter sort by name:");
        parts.ForEach(Console.WriteLine);

        /*

            Before sort:
        ID: 1434   Name: regular seat
        ID: 1234   Name: crank arm
        ID: 1634   Name: shift lever
        ID: 1334   Name:
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette

        After sort by part number:
        ID: 1234   Name: crank arm
        ID: 1334   Name:
        ID: 1434   Name: regular seat
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette
        ID: 1634   Name: shift lever

        After sort by name:
        ID: 1334   Name:
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette
        ID: 1234   Name: crank arm
        ID: 1434   Name: regular seat
        ID: 1634   Name: shift lever

         */
    }
}
```

See also

- IComparer
- IEquatable<T>
- IComparer<T>
- IComparable
- IComparable<T>