[1220] Sorting Collections

C#'s List collection class has several ways to Sort the items in the list. Over the years, more and more options have been added to give programmers different ways of sorting Lists. We'll review several different approaches and then discuss the pros and cons of each approach.

List.Sort()

The List collection has a very straightforward method called Sort() that works well as long as the list contains objects that support the IComparable interface and you want the items sorted from smallest to largest. (Note: You can sort the list and then use List.Reverse() to generate a list sorted from largest to smallest.) A List that contains any of the standard C# data types can easily be sorted in this way. Example:

```
List<int> scores = new List<int> { 4, 5, 12, -5, 8 };
scores.Sort();
scores.Reverse();
foreach (int i in scores)
    Console.WriteLine(i);
```

Output:

```
12
8
5
4
-5
```

List.Sort(IComparable)

If you are sorting a List of objects from a class that you wrote, you can have that class implement the IComparable interface and then you can go back to just using List.Sort() without any parameters. Here is a (simple) example:

```
    class DataPoint : IComparable
    {
        private readonly int myValue;

        public DataPoint(int i)
        {
            myValue = i;
        }

        public int CompareTo(Object other)
```

```
        {
            DataPoint otherDP = (DataPoint)other;
            if (myValue < otherDP.myValue) return -1;
            if (myValue > otherDP.myValue) return +1;
            return 0;
        }

        public override string ToString()
        {
            return "" + myValue;
        }
    }

    class Program
    {
        public static void Main()
        {
            Console.WriteLine("List.Sort() Approach with IComparable object");
            List scores = new List { new DataPoint(4), new DataPoint(5), new DataPoint(12), new DataPoint
(-5), new DataPoint(8) };
            scores.Sort();
            scores.Reverse();
            foreach (DataPoint dp in scores)
                Console.WriteLine(dp);
        }
    }
```

Output:

```
12
8
5
4
-5
```

List.Sort(Comparer)

But what if you are sorting things that don't support IComparable? In that case, you can use a special method called a "Comparer method" to help do the sorting. A Comparer method takes two items and "compares" them using whatever method you want and then returns +1, -1, or 0. It returns +1 if the first item is greater than the second item. It returns -1 if the first item is less than that second item. And it returns 0 if the two items are equal. Again, the comparer method can use whatever algorithm you need to use in order tp determine that ordering.

To sort the list, you create the comparer method and then use the name of that method as a parameter for the Sort() method. Here's an (contrived) example:

```
private static int ReverseIntegerComparer(int first, int second)
{
    if (first < second) return +1;
    if (first > second) return -1;
    return 0;
```

```
}

public static void Main()
{
    List<int> scores = new List<int> { 4, 5, 12, -5, 8 };
    scores.Sort(ReverseIntegerComparer);
    foreach (int i in scores)
        Console.WriteLine(i);
}
```

Output:

```
12
8
5
4
-5
```

List.Sort(Comparer) using CompareTo()

The above code can be simplified by using CompareTo() on each item. Example:

```
private static int ReverseIntegerCompareToComparer(int first, int second)
{
    return second.CompareTo(first);
}

public static void Main()
{
    List<int> scores = new List<int> { 4, 5, 12, -5, 8 };
    scores.Sort(ReverseIntegerCompareToComparer);
    foreach (int i in scores)
        Console.WriteLine(i);
}
```

List.Sort(Lambda)

For simple sorting - such as what we are doing in this example - a short Lambda expression is perfect. Example:

```
public static void Main()
{
    List<int> scores = new List<int> { 4, 5, 12, -5, 8 };
    scores.Sort((first,second) => second.CompareTo(first));
    foreach (int i in scores)
        Console.WriteLine(i);
}
```

Note the correlation between the Lambda expression and the ReverseIntegerCompareToComparer() method in the previous example.