Item 31: Implement Ordering Relations with IComparable<T> and IComparer<T>

Your types need ordering relationships to describe how collections should be sorted and searched. The .NET Framework defines two interfaces that describe ordering relationships in your types: IComparable<T> and IComparer<T>. IComparable defines the natural order for your types. A type implements IComparer to describe alternative orderings. You can define your own implementations of the relational operators (<, >, <=, >=) to provide type-specific comparisons, to avoid some runtime inefficiencies in the interface implementations. This item discusses how to implement ordering relations so that the core .NET Framework orders your types through the defined interfaces and so that other users get the best performance from these operations.

The IComparable interface contains one method: CompareTo(). This method follows the long-standing tradition started with the C library function strcmp: Its return value is less than 0 if the current object is less than the comparison object, 0 if they are equal, and greater than 0 if the current object is greater than the comparison object. IComparable<T> will be used by most newer APIs in the .NET landscape. However, some older APIs will use the classic IComparable interface. Therefore, when you implement IComparable<T>, you should also implement IComparable. IComparable takes parameters of type System.Object. You need to perform runtime type checking on the argument to this function. Every time comparisons are performed, you must reinterpret the type of the argument:

```
public struct Customer : IComparable<Customer>, IComparable
{
    private readonly string name;
    public Customer(string name)
    {
        this.name = name;
    }

    #region IComparable<Customer> Members
    public int CompareTo(Customer other)
    {
        return name.CompareTo(other.name);
    }
    #endregion

    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
            throw new ArgumentException(
                "Argument is not a Customer", "obj");
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }
    #endregion
}
```

Notice that IComparable is explicitly implemented in this structure. That ensures that the only code that will call the object-typed version of CompareTo() is code that was written for the previous interface. There's just too much to dislike about the classic version of IComparable. You've got to check the runtime type of the argument. Incorrect code could legally call this method with anything as the argument to the CompareTo method. More so, proper arguments must be boxed and unboxed to provide the actual comparison. That's an extra runtime expense for each compare. Sorting a collection will make, on average n lg(n) comparisons of your object using the IComparable.Compare method. Each of those will cause three boxing and unboxing operations. For an array with 1,000 points, that will be more than 20,000 boxing and unboxing operations, on average: n lg(n) is almost 7,000, and there are 3 box and unbox operations per comparison.

You may be wondering why you should implement the nongeneric IComparable interface at all. There are two reasons. First, there's simple backward compatibility. Your types will interact with code created before .NET 2.0. That means supporting the pregeneric interface. Second, even more modern code will avoid generics when it's based on reflection. Reflection using generics is possible, but it's much more difficult than reflection using nongeneric type definitions. Supporting the nongeneric version of the IComparable interface makes your type easier to use for algorithms making use of reflection.

Because the classic IComparable.CompareTo() is now an explicit interface implementation, it can be called only through an IComparable reference. Users of your customer `struct` will get the type-safe comparison, and the unsafe comparison is inaccessible. The following innocent mistake no longer compiles:

```
Customer c1;
Employee e1;
if (c1.CompareTo(e1) > 0)
    Console.WriteLine("Customer one is greater");
```

It does not compile because the arguments are wrong for the public Customer.CompareTo(Customer right) method. The IComparable.CompareTo(object right) method is not accessible. You can access the IComparable method only by explicitly casting the reference:

```
Customer c1 = new Customer();
Employee e1 = new Employee();
if ((c1 as IComparable).CompareTo(e1) > 0)
    Console.WriteLine("Customer one is greater");
```

When you implement IComparable, use explicit interface implementation and provide a strongly typed public overload. The strongly typed overload improves performance and decreases the likelihood that someone will misuse the CompareTo method. You won't see all the benefits in the Sort function that the .NET Framework uses because it will still access CompareTo() through the interface pointer (see Item 22), but code that knows the type of both objects being compared will get better performance.

We'll make one last small change to the Customer `struct`. The C# language lets you overload the standard relational operators. Those should make use of the type-safe CompareTo() method:

```
public struct Customer : IComparable<Customer>, IComparable
{
    private readonly string name;
    public Customer(string name)
    {
        this.name = name;
    }

    #region IComparable<Customer> Members
    public int CompareTo(Customer other)
    {
        return name.CompareTo(other.name);
    }
    #endregion

    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
            throw new ArgumentException(
                "Argument is not a Customer", "obj");
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }
    #endregion

    // Relational Operators.
    public static bool operator <(Customer left,
      Customer right)
    {
        return left.CompareTo(right) < 0;
    }
    public static bool operator <=(Customer left,
      Customer right)
    {
        return left.CompareTo(right) <= 0;
    }
    public static bool operator >(Customer left,
      Customer right)
    {
        return left.CompareTo(right) > 0;
    }
    public static bool operator >=(Customer left,
      Customer right)
    {
        return left.CompareTo(right) >= 0;
    }
}
```

That's all for the standard order of customers: by name. Later, you must create a report sorting all customers by revenue. You still need the normal comparison functionality defined by the Customer struct, sorting them by name. Most APIs developed after generics became part of the .NET Framework will ask for a Comparison<T> delegate to perform some other sort. It's simple to create static properties in the Customer type that provide other comparison orders. For example, this delegate compares the revenue generated by two customers:

```
public static Comparison<Customer> CompareByReview
{
    get
    {
        return (left,right) =>
            left.revenue.CompareTo(right.revenue);
    }
}
```

Older libraries will ask for this kind of functionality using the IComparer interface. IComparer provides the standard way to provide alternative orders for a type without using generics. Any of the methods delivered in the 1.x .NET FCL that work on IComparable types provide overloads that order objects through IComparer. Because you authored the Customer struct, you can create this new class (RevenueComparer) as a private nested class inside the Customer struct. It gets exposed through a static property in the Customer struct:

```
public struct Customer : IComparable<Customer>, IComparable
{
    private readonly string name;
    private double revenue;

    public Customer(string name, double revenue)
    {
        this.name = name;
        this.revenue = revenue;
    }

    #region IComparable<Customer> Members
    public int CompareTo(Customer other)
    {
        return name.CompareTo(other.name);
    }
    #endregion

    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
            throw new ArgumentException(
                "Argument is not a Customer", "obj");
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }
    #endregion

    // Relational Operators.
    public static bool operator <(Customer left,
      Customer right)
    {
        return left.CompareTo(right) < 0;
    }
    public static bool operator <=(Customer left,
      Customer right)
    {
        return left.CompareTo(right) <= 0;
    }
    public static bool operator >(Customer left,
      Customer right)
    {
        return left.CompareTo(right) > 0;
    }
    public static bool operator >=(Customer left,
      Customer right)
    {
        return left.CompareTo(right) >= 0;
    }

    private static RevenueComparer revComp = null;

    // return an object that implements IComparer
    // use lazy evaluation to create just one.
    public static IComparer<Customer> RevenueCompare
    {
        get
        {
            if (revComp == null)
                revComp = new RevenueComparer();
            return revComp;
        }
    }

    public static Comparison<Customer> CompareByReview
    {
        get
        {
            return (left,right) =>
                left.revenue.CompareTo(right.revenue);
        }
    }

    // Class to compare customers by revenue.
    // This is always used via the interface pointer,
    // so only provide the interface override.
    private class RevenueComparer : IComparer<Customer>
    {
        #region IComparer<Customer> Members
        int IComparer<Customer>.Compare(Customer left,
            Customer right)
        {
            return left.revenue.CompareTo(
              right.revenue);
        }
        #endregion
    }
    }
```

The last version of the Customer `struct`, with the embedded RevenueComparer, lets you order a collection of customers by name, the natural order for customers, and provides an alternative order by exposing a class that implements the IComparer interface to order customers by revenue. If you don't have access to the source for the Customer class, you can still provide an IComparer that orders customers using any of its public properties. You should use that idiom only when you do not have access to the source for the class, as when you need a different ordering for one of the classes in the .NET Framework.

Nowhere in this item did I mention Equals() or the == operator (see Item 6). Ordering relations and equality are distinct operations. You do not need to implement an equality comparison to have an ordering relation. In fact, reference types commonly implement ordering based on the object contents, yet implement equality based on object identity. CompareTo() returns 0, even though Equals() returns false. That's perfectly legal. Equality and ordering relations are not necessarily the same.

IComparable and IComparer are the standard mechanisms for providing ordering relations for your types. IComparable should be used for the most natural ordering. When you implement IComparable, you should overload the comparison operators (<, >, <=, >=) consistently with our IComparable ordering. IComparable.CompareTo() uses System.Object parameters, so you should also provide a type-specific overload of the CompareTo() method. IComparer can be used to provide alternative orderings or can be used when you need to provide ordering for a type that does not provide it for you.