Icomparable and IComparer Interfaces

If you have an array of types (such as **string** or **integer**) that already support **IComparer**, you can sort that array without providing any explicit reference to **IComparer**. In that case, the elements of the array are cast to the default implementation of **IComparer** (**Comparer.Default**) for you. However, if you want to provide sorting or comparison capability for your custom objects, you must implement either or both of these interfaces.

The following .NET Framework Class Library namespace is referenced in this article:
System.Collections

IComparable

The role of **IComparable** is to provide a method of comparing two objects of a particular type. This is necessary if you want to provide any ordering capability for your object. Think of **IComparable** as providing a default sort order for your objects. For example, if you have an array of objects of your type, and you call the **Sort** method on that array, **IComparable** provides the comparison of objects during the sort. When you implement the **IComparable** interface, you must implement the **CompareTo** method, as follows:

```
// Implement IComparable CompareTo method - provide default sort order.
int IComparable.CompareTo(object obj)
{
   car c=(car)obj;
   return String.Compare(this.make,c.make);

}
```

The comparison in the method is different depending on the data type of the value that is being compared. **String.Compare** is used in this example because the property that is chosen for the comparison is a string.

IComparer
The role of **IComparer** is to provide additional comparison mechanisms. For example, you may want to provide ordering of your class on several fields or properties, ascending and descending order on the same field, or both.

Using **IComparer** is a two-step process. First, declare a class that implements **IComparer**, and then implement the **Compare** method:

```
private class sortYearAscendingHelper : IComparer
{
   int IComparer.Compare(object a, object b)
   {
      car c1=(car)a;
      car c2=(car)b;
      if (c1.year > c2.year)
         return 1;
      if (c1.year < c2.year)
         return -1;
      else
         return 0;
   }
```

```
}
```

Note that the **IComparer.Compare** method requires a tertiary comparison. 1, 0, or -1 is returned depending on whether one value is greater than, equal to, or less than the other. The sort order (ascending or descending) can be changed by switching the logical operators in this method.

The second step is to declare a method that returns an instance of your **IComparer** object:

```
public static IComparer sortYearAscending()
{
   return (IComparer) new sortYearAscendingHelper();
}
```

In this example, the object is used as the second argument when you call the overloaded **Array.Sort** method that accepts **IComparer**. The use of **IComparer** is not limited to arrays. It is accepted as an argument in a number of different collection and control classes.

Step-by-Step Example

The following example demonstrates the use of these interfaces. To demonstrate **IComparer** and **IComparable**, a class named **car** is created. The **car** object has the **make** and **year** properties. An ascending sort for the **make** field is enabled through the **IComparable** interface, and a descending sort on the **make** field is enabled through the **IComparer** interface. Both ascending and descending sorts are provided for the **year** property through the use of **IComparer**.

1. In Visual C#, create a new Console Application project. Name the application ConsoleEnum.
2. Rename Program.cs as Host.cs, and then replace the code with the following code.

.

```
using System;

namespace ConsoleEnum
{
   class host
   {
      [STAThread]
      static void Main(string[] args)
      {
         // Create an arary of car objects.
         car[] arrayOfCars= new car[6]
         {
            new car("Ford",1992),
            new car("Fiat",1988),
            new car("Buick",1932),
            new car("Ford",1932),
            new car("Dodge",1999),
            new car("Honda",1977)
         };

         // Write out a header for the output.
         Console.WriteLine("Array - Unsorted\n");

         foreach(car c in arrayOfCars)
```

```
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo IComparable by sorting array with "default" sort order.
         Array.Sort(arrayOfCars);
         Console.WriteLine("\nArray - Sorted by Make (Ascending -
IComparable)\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo ascending sort of numeric value with IComparer.
         Array.Sort(arrayOfCars,car.sortYearAscending());
         Console.WriteLine("\nArray - Sorted by Year (Ascending -
IComparer)\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo descending sort of string value with IComparer.
         Array.Sort(arrayOfCars,car.sortMakeDescending());
         Console.WriteLine("\nArray - Sorted by Make (Descending -
IComparer)\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         // Demo descending sort of numeric value using IComparer.
         Array.Sort(arrayOfCars,car.sortYearDescending());
         Console.WriteLine("\nArray - Sorted by Year (Descending -
IComparer)\n");

         foreach(car c in arrayOfCars)
            Console.WriteLine(c.Make + "\t\t" + c.Year);

         Console.ReadLine();
      }
   }
}
```

3. Add a class to the project. Name the class **car**.
4. Replace the code in Car.cs with the following:

```
using System;
using System.Collections;
namespace ConsoleEnum
{
   public class car : IComparable
   {
      // Beginning of nested classes.

      // Nested class to do ascending sort on year property.
      private class sortYearAscendingHelper: IComparer
      {
         int IComparer.Compare(object a, object b)
         {
```

```
            car c1=(car)a;
            car c2=(car)b;

            if (c1.year > c2.year)
               return 1;

            if (c1.year < c2.year)
               return -1;

            else
               return 0;
         }
      }

      // Nested class to do descending sort on year property.
      private class sortYearDescendingHelper: IComparer
      {
         int IComparer.Compare(object a, object b)
         {
            car c1=(car)a;
            car c2=(car)b;

            if (c1.year < c2.year)
               return 1;

            if (c1.year > c2.year)
               return -1;

            else
               return 0;
         }
      }

      // Nested class to do descending sort on make property.
      private class sortMakeDescendingHelper: IComparer
      {
         int IComparer.Compare(object a, object b)
         {
            car c1=(car)a;
            car c2=(car)b;
             return String.Compare(c2.make,c1.make);
         }
      }

      // End of nested classes.

      private int year;
      private string make;

      public car(string Make,int Year)
      {
         make=Make;
         year=Year;
      }
```

```
      public int Year
      {
         get  {return year;}
         set {year=value;}
      }

      public string Make
      {
         get {return make;}
         set {make=value;}
      }

      // Implement IComparable CompareTo to provide default sort order.
      int IComparable.CompareTo(object obj)
      {
         car c=(car)obj;
         return String.Compare(this.make,c.make);
      }

      // Method to return IComparer object for sort helper.
      public static IComparer sortYearAscending()
      {
         return (IComparer) new sortYearAscendingHelper();
      }

      // Method to return IComparer object for sort helper.
      public static IComparer sortYearDescending()
      {
         return (IComparer) new sortYearDescendingHelper();
      }

      // Method to return IComparer object for sort helper.
      public static IComparer sortMakeDescending()
      {
        return (IComparer) new sortMakeDescendingHelper();
      }

   }
}
```

5. Run the project. The following output appears in the Console window:

```
Array - Unsorted

Ford            1992
Fiat            1988
Buick           1932
Ford            1932
Dodge           1999
Honda           1977

Array - Sorted by Make (Ascending - IComparable)

Buick           1932
```

```
Dodge           1999
Fiat            1988
Ford            1932
Ford            1992
Honda           1977

Array - Sorted by Year (Ascending - IComparer)

Ford            1932
Buick           1932
Honda           1977
Fiat            1988
Ford            1992
Dodge           1999

Array - Sorted by Make (Descending - IComparer)

Honda           1977
Ford            1932
Ford            1992
Fiat            1988
Dodge           1999
Buick           1932

Array - Sorted by Year (Descending - IComparer)

Dodge           1999
Ford            1992
Fiat            1988
Honda           1977
Buick           1932
Ford            1932
```

IEquatable

An IEquatable class provides an Equals method that determines whether an object is equal to another object.

```
namespace IEquatablePerson
{
    class Person : IEquatable<Person>
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }



        public bool Equals(Person other)
        {
            return ((FirstName == other.FirstName) &&
                (LastName == other.LastName));
```

```
    }
  }
}
```

```
        Person person = new Person()
        {
            FirstName = firstNameTextBox.Text,
            LastName = lastNameTextBox.Text
        };

        if (People.Contains(person))
        {
            MessageBox.Show("The list already contains this person.");
        }
        else
        {
            People.Add(person);
            firstNameTextBox.Clear();
            lastNameTextBox.Clear();
            firstNameTextBox.Focus();
        }
```

ICloneable

An ICloneable class provides a Clone method that returns a copy of an object.
Deep clone A copy of an object where reference fields refer to new instances of objects, not to the same objects referred to by the original object's fields.
Shallow clone A copy of an object where reference fields refer to the same objects as the original object's fields.

```
namespace ICloneablePerson
{
    class Person : ICloneable
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Person Manager { get; set; }

        // Return a clone of this person.
        public object Clone()
        {
            Person person = new Person();
            person.FirstName = FirstName;
            person.LastName = LastName;
            person.Manager = Manager;
            // Uncomment the following for deep clones.
            //if (Manager != null)
            //    person.Manager = (Person)Manager.Clone();
            return person;
        }

        // Return a textual representation of the Person.
```

```
        public override string ToString()
        {
            string text = FirstName + " " + LastName;
            if (Manager != null)
                text += " (Manager: " + Manager.ToString() + ")";
            return text;
        }
    }
}
```

The following code shown how to clone an object.

```
            Person ann = new Person()
            {
                FirstName = "Ann",
                LastName = "Archer",
                Manager = null
            };
            Person bob = new Person()
            {
                FirstName = "Bob",
                LastName = "Baker",
                Manager = ann
            };
            Person bob2 = (Person)bob.Clone();
            Person cindy = new Person()
            {
                FirstName = "Cindy",
                LastName = "Cane",
                Manager = bob
            };

            // Change Bob's manager's name.
            bob.Manager.FirstName = "Dan";
            bob.Manager.LastName = "Dent";
```

Destructors

- Destructors can be defined in classes only, not structures.
- A class can have at most one destructor.
- Destructors cannot be inherited or overloaded.
- Destructors cannot be called directly.
- Destructors cannot have modifiers or parameters.

The destructor is converted into an override version of the Finalize method. You cannot override Finalize or call it directly.

Using statement

The using statement lets a program automatically call an object's Dispose method, so you can't forget to do it. If you declare and initialize the object in the using statement, this also limits the object's scope to the using block.

The following code shown on how to use using statement.

```
using (DisposableClass obj = new DisposableClass())
{
   obj.Name = "CreateAndDispose " + ObjectNumber.ToString();
   ObjectNumber++;
}
```

Polymorphism

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance.

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used whereever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        System.Collections.Generic.List<Shape> shapes = new
        System.Collections.Generic.List<Shape>();
        shapes.Add(new Rectangle());
        shapes.Add(new Triangle());
        shapes.Add(new Circle());

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (Shape s in shapes)
        {
            s.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

Upcasting and downcasting

Upcasting converts an object of a specialized type to a more general type
Downcasting converts an object from a general type to a more specialized type

A specialization hierarchy of bank accounts

BankAccount ba1, ba2 = new BankAccount("John", 250.0M, 0.01);

LotteryAccount la1, la2 = new LotteryAccount("Bent", 100.0M);

ba1 = la2; // upcasting - OK

// la1 = ba2; // downcasting - Illegal - discovered at compile time

// la1 = (LotteryAccount)ba2; // downcasting – Illegal - discovered at run time

la1 = (LotteryAccount)ba1; // downcasting - OK - ba1 already refers to a LotteryAccount

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an InvalidCastException to be thrown.

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;
      //solution
      // if (a is Reptile)
      //    Reptile r = (Reptile)a;

    }

}
```

C# provides the is and as operators to enable you to test for compatibility before actually performing a cast.

```
                Base b = d as Base;
                if (b != null)
                {
                    Console.WriteLine(b.ToString());
                }

            }
        }
    }
```

Oveloading operators

This example shows how you can use operator overloading to create a complex number class Complex that defines complex addition. The program displays the imaginary and the real parts of the numbers and the addition result using an override of the ToString method.

Example:

```
public struct Complex
{
    public int real;
    public int imaginary;

    // Constructor.
    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Specify which operator to overload (+),
    // the types that can be added (two Complex objects),
    // and the return type (Complex).
    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
    }

    // Override the ToString() method to display a complex number
    // in the traditional format:
    public override string ToString()
    {
        return (System.String.Format("{0} + {1}i", real, imaginary));
    }
}

class TestComplex
{
    static void Main()
    {
        Complex num1 = new Complex(2, 3);
        Complex num2 = new Complex(3, 4);

        // Add two Complex objects by using the overloaded + operator.
```

```
        Complex sum = num1 + num2;

        // Print the numbers and the sum by using the overridden
        // ToString method.
        System.Console.WriteLine("First complex number:  {0}", num1);
        System.Console.WriteLine("Second complex number: {0}", num2);
        System.Console.WriteLine("The sum of the two numbers: {0}", sum);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    First complex number:  2 + 3i
    Second complex number: 3 + 4i
    The sum of the two numbers: 5 + 7i
*/
```

Class versus struct

Syntactical Comparison:

Now, semantics aside, there are a lot of things that are similar between **struct** and **class** in C#, but there are also a fair number of surprising differences. Let's look at a table that sums them up:

Feature	*Struct*	*Class*	*Notes*
Is a reference type?	No	Yes*	
Is a value type?	Yes	No	
Can have nested Types (enum, class, struct)?	Yes	Yes	
Can have constants?	Yes	Yes	
Can have fields?	Yes*	Yes	*Struct instance fields cannot be initialized, will automatically initialize to default value.*
Can have properties?	Yes	Yes	
Can have indexers	Yes	Yes	
Can have methods?	Yes	Yes	
Can have events?	Yes*	Yes	*Structs, like classes, can have events, but care must be taken that you don't subscribe to a* **copy**

			of a struct instead of the struct you intended.
Can have static members (constructors, fields, methods, properties, etc.)?	Yes	Yes	
Can inherit?	No*	Yes*	*Classes can inherit from other classes (or object by default). Structs always inherit from System.ValueType and are sealed implicitly*
Can implement interfaces?	Yes	Yes	
Can overload constructor?	Yes*	Yes	*Struct overload of constructor does not hide default constructor.*
Can define default constructor?	No	Yes	*The struct default constructor initializes all instance fields to default values and cannot be changed.*
Can overload operators?	Yes	Yes	
Can be generic?	Yes	Yes	
Can be partial?	Yes	Yes	
Can be sealed?	Always*	Yes	*Structs are always sealed and can never be inherited from.*
Can be referenced in instance members using *this* keyword?	Yes*	Yes	*In structs, this is a value variable, in classes, it is a readonly reference.*
Needs *new* operator to create instance?	No*	Yes	*C# classes must be instantiated using new. However, structs do not require this. While new can be used on a struct to call a constructor, you can elect not to use*

			new and init the fields yourself, but you must init all fields and the fields must be public!

Many of the items in this table should be fairly self explanatory. As you can see there the majority of behavior is identical for both **class** and **struct** types, however, there are several differences which can become potential pitfalls if not respected and understood!

```
public class PointClass
{
    public int X { get; set; }
    public int Y { get; set; }

}

public struct PointStruct
{
    public int X { get; set; }
    public int Y { get; set; }

}

public static class Program
{
    public static void Main()
    { // assignment from one reference to another simply copies reference and increments
      //reference count.

      // In this case both references refer to one single object

      var pointClass = new PointClass { X = 5, Y = 10 };

      var copyOfPointClass = pointClass;

      // modifying one reference modifies all references referring to the same object

      copyOfPointClass.X = 0;

      // this will output [0, 10] even though it's our original reference because they are both
      //referring to the same object

      Console.WriteLine("Original pointClass is [{0},{1}]", pointClass.X, pointClass.Y);

      // ...
      // assignment from one struct to another makes a complete copy, so there are two separate
      //PointStruct each with their own X and Y


      var pointStruct = new PointStruct { X = 5, Y = 10 };

      var copyOfPointStruct = pointStruct;
```

```
        // modifying one reference modifies all references referring to the same object
        copyOfPointStruct.X = 0;
        // the output will be [5,10] because the original pointStruct is unaffected by changes to the
        //copyOfPOintStruct
        Console.WriteLine("Original pointStruct is [{0},{1}]", pointStruct.X, pointStruct.Y);
    }
}
```

Namespace

Namespaces are C# program elements designed to help you organize your programs. They also provide assistance in avoiding name clashes between two sets of code. Implementing Namespaces in your own code is a good habit because it is likely to save you from problems later when you want to reuse some of your code. For example, if you created a class named Console, you would need to put it in your own namespace to ensure that there wasn't any confusion about when the System.Console class should be used or when your class should be used. Generally, it would be a bad idea to create a class named Console, but in many cases your classes will be named the same as classes in either the .NET Framework Class Library or a third party library and namespaces help you avoid the problems that identical class names would cause.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface SampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace SampleNamespace.Nested
    {
        class SampleClass2 { }
    }
}
```

The following example shows how to call a static method in a nested namespace.

```
namespace SomeNameSpace
{
    public class MyClass
    {
```

```
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

Using Directive

```
// Namespace Declaration
using System;
using csharp_station.tutorial;

// Program start class
class UsingDirective
{
  // Main begins program execution.
  public static void Main()
  {
    // Call namespace member
    myExample.myPrint();
  }
}

// C# Station Tutorial Namespace
namespace csharp_station.tutorial
{
  class myExample
  {
    public static void myPrint()
    {
      Console.WriteLine("Example  of using a using directive.");
    }
  }
}
```