**Specifying Multiple Sort Orders with IComparer**

In this version of the Car type, you used the car’s ID as the base for the sort order. Another design might

have used the pet name of the car as the basis for the sorting algorithm (to list cars alphabetically). Now,

what if you wanted to build a Car that could be sorted by ID *as well as* by pet name? If this is the type of

behavior you are interested in, you need to make friends with another standard interface named

IComparer, defined within the System.Collections namespace as follows:

// A general way to compare two objects.

interface IComparer

{

int Compare(object o1, object o2);

}

The generic version of this interface (IComparer<T>) provides a more type-safe manner to handle

comparisons between objects

Unlike the IComparable interface, IComparer is typically *not* implemented on the type you are trying

to sort (i.e., the Car). Rather, you implement this interface on any number of helper classes, one for each

sort order (pet name, car ID, etc.). Currently, the Car type already knows how to compare itself against

other cars based on the internal car ID. Therefore, allowing the object user to sort an array of Car objects

by pet name will require an additional helper class that implements IComparer. Here’s the code (be sure

to import the System.Collections namespace in the code file):

// This helper class is used to sort an array of Cars by pet name.

public class PetNameComparer : IComparer

{

// Test the pet name of each object.

int IComparer.Compare(object o1, object o2)

{

Car t1 = o1 as Car;

Car t2 = o2 as Car;

if(t1 != null && t2 != null)

return String.Compare(t1.PetName, t2.PetName);

else

throw new ArgumentException("Parameter is not a Car!");

}

}

The object user code is able to make use of this helper class. System.Array has a number of

overloaded Sort() methods, one that just happens to take an object implementing IComparer.

static void Main(string[] args)

{

...

// Now sort by pet name.

Array.Sort(myAutos, new PetNameComparer());

// Dump sorted array.

Console.WriteLine("Ordering by pet name:");

foreach(Car c in myAutos)

Console.WriteLine("{0} {1}", c.CarID, c.PetName);

...

}

Custom Properties and Custom Sort Types

It is worth pointing out that you can make use of a custom static property in order to help the object user

along when sorting your Car types by a specific data point. Assume the Car class has added a static readonly

property named SortByPetName that returns an instance of an object implementing the IComparer

interface (PetNameComparer, in this case):

// We now support a custom property to return

// the correct IComparer interface.

public class Car : IComparable

{

...

// Property to return the PetNameComparer.

public static IComparer SortByPetName

{ get { return (IComparer)new PetNameComparer(); } }

}

The object user code can now sort by pet name using a strongly associated property, rather than just

“having to know” to use the stand-alone PetNameComparer class type:

// Sorting by pet name made a bit cleaner.

Array.Sort(myAutos, Car.SortByPetName);

Hopefully, at this point you not only understand how to define and implement your own interfaces,

but also understand their usefulness. To be sure, interfaces are found within every major .NET

namespace, and you will continue working with various standard interfaces over the remainder of this

text.