# working with C# interfaces

**by Glen McCluskey**

Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

*glenm@glenmccl.com*

Suppose that you're doing some C programming and have a list of numbers to sort in descending order. Instead of writing your own sort routine, you decide it would be better to use the library function qsort. Here's some sample code:

```
#include <stdio.h>
#include <stdlib.h>

#define N 10

int cmp(const void* ap, const void* bp) {
    int a = *(int*)ap;
    int b = *(int*)bp;

    return (a < b ? 1 : a == b ? 0 : -1);
}

int main() {
    int i;
    int list[N];

    for (i = 0; i < N; i++)
        list[i] = i;

    qsort(list, N, sizeof(int), cmp);

    for (i = 0; i < N; i++)
        printf("%d ", list[i]);
    printf("\n");
}
```

It is possible to make general use of the library sort function because its interface has been standardized, and the element comparison function has been factored out and is supplied by the user.

Suppose that you'd like to write some equivalent C# code. What might it look like? Here's one way of doing it:

```
using System;
using System.Collections;

public class MyComparer : IComparer {
    public int Compare(object aobj, object bobj) {
        int a = (int)aobj;
        int b = (int)bobj;

        return (a < b ? 1 : a == b ? 0 : -1);
    }
}

public class SortDemo {
    public static void Main() {
        const int N = 10;
        ArrayList list = new ArrayList();

        for (int i = 0; i < N; i++)
            list.Add(i);

        list.Sort();

        for (int i = 0; i < N; i++)
            Console.Write(list[i] + " ");
        Console.WriteLine();

        list.Sort(new MyComparer());

        for (int i = 0; i < N; i++)
            Console.Write(list[i] + " ");
        Console.WriteLine();
    }
}
```

When this code is run, the result is:

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

This approach uses an instance of the ArrayList class, a list of objects represented using an internal array. ArrayList has a Sort method, which sorts the objects in natural (ascending) order. There's also a Sort method to which you specify a comparator. Since C# has no global functions, the idea here is that an object of a class MyComparer is created and passed to the Sort method. MyComparer is a class whose instances serve as wrappers for a comparison method, the equivalent of the C comparison function.

Because the Sort method is part of a standard library class that will call a user-supplied comparator method, there has to be some way of uniformly specifying what such methods look like. C# uses what are called interfaces for this purpose. In the example above, the standard interface IComparer would be declared like this:

```
public interface IComparer {
    int Compare(object a, object b);
}
```

A class such as MyComparer then implements the interface by defining a method Compare with the appropriate signature. The Compare method has similar semantics to what is found in

C, returning -1, for example, if the first element is "less than" the second and 1 if the first element is "greater."

The Sort method in ArrayList is declared like this:

```
void Sort();

void Sort(IComparer);
```

The first of these declarations represents the default, and the second has a single parameter of type IComparer, meaning that an object of any class that implements the IComparer interface can be passed to the Sort method.

A C# interface specifies that an implementing class will define particular methods with specific signatures, but says nothing about what those methods will actually do. If, for example, I write a comparator method to be used in sorting, and that method returns a random value (-1, 0, 1) each time it is called, then the sorting process isn't going to turn out very well. An interface is a contract that specifies *what*, not *how*.

## Writing Your Own Interface

When might you wish to use your own interfaces? Consider an application where you have some objects of classes for which it is meaningful to calculate the distance between objects. For example, the objects might represent X,Y points on a plane or calendar dates, and your application needs to know the distance between any two objects.

Here's some code that shows how an interface can be defined and then used:

```
using System;

public interface IDistance {
    double GetDistance(object obj);
}

public class Point : IDistance {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int GetX() {
        return x;
    }

    public int GetY() {
        return y;
    }

    public double GetDistance(object obj) {
        Point pobj = obj as Point;
        if (pobj == null)
            throw new NullReferenceException();

        double sum = 0;
        sum += (x - pobj.x) * (x - pobj.x);
        sum += (y - pobj.y) * (y - pobj.y);

        return Math.Sqrt(sum);
    }
}

public class DistDemo {
    public static void Main() {
        IDistance p1 = new Point(10, 10);
        IDistance p2 = new Point(20, 20);

        Console.WriteLine("distance = " + p1.GetDistance(p2));
    }
}
```

The interface IDistance specifies a single method GetDistance. The idea is that you have an object, and GetDistance is called to compute the distance between that object and another:

```
double distance = obj1.GetDistance(obj2);
```

The interface doesn't specify how the distance is computed. In our example, we calculate the Euclidean distance between two X,Y points.

Note that the GetDistance implementation uses the "as" operator. The IDistance interface is specified generically, to work on any type of objects. However, when the interface is implemented in the Point class, it's only meaningful to compute the distance between one Point and another. The "as" operator checks whether an arbitrary object is of type Point, and, if so, returns a Point reference. Otherwise it returns null.

## Programming in Terms of Interfaces

In the previous example, you might have noticed lines of code such as the following:

```
IDistance p1 = new Point(10, 10);
```

A class type like Point is compatible with the type of interface that it implements, such as IDistance.

In some situations, this compatibility can serve as the basis for a whole style of programming. Suppose, for example, that you're using the standard class ArrayList in your application. You can specify method parameter types and so forth using the ArrayList type, but it's also possible to use IList, a system interface that ArrayList implements. IList describes a collection that supports indexable access to individual members.

Here's an example of this idea:

```
using System;
using System.Collections;

public class IntDemo {
    static void method1(IList list) {
        list.Add(10);
        list.Add(20);
```

```
        list.Add(30);
    }

    static void method2(IList list) {
        for (int i = 0; i < list.Count; i++)
            Console.WriteLine(list[i]);
    }

    public static void Main() {
        IList list = new ArrayList();

        method1(list);
        method2(list);
    }
}
```

method1 and method2 are implemented in terms of IList instead of ArrayList.

Why does this matter? Suppose that at some later time you want to use a class LinkedList in place of ArrayList. Arrays and linked lists have some performance tradeoffs. For example, random access is much faster in an array than in a linked list, but inserting in the middle of a linked list is much faster than in an array.

If you program in terms of interfaces, as this example illustrates, then it's possible to change the underlying implementation of a data structure without having to touch most of your code. In the example, method1 and method2 are not programmed in terms of particular data structures such as ArrayList, but in terms of an interface that specifies methods like Add. Programming in this way is an example of what is called polymorphism, or programming using a particular interface without regard to the underlying implementation details.

## Extending Interfaces

It's possible to extend interfaces, just as with classes. For example, in this code:

```
public interface Interface1 {
    void f1();
}

public interface Interface2 : Interface1 {
    void f2();
}

public class ClassA : Interface2 {
    public void f1() {}
    public void f2() {}
}
```

Interface2 extends Interface1, and ClassA must define both f1 and f2 in order to actually implement the interfaces.

Our example from the previous section uses the standard interface IList. This interface extends the more general interface ICollection, which defines the property Count (a count of the number of elements in a collection) used in our example.

You can also specify that a class implement more than one interface – for example, the IList, IComparer, and IDistance interfaces discussed above. Implementing interfaces defines the "implements" relationship between the class and the interface (the term "mix in" is sometimes used to describe adding capabilities to a class by implementing additional interfaces).

## Testing Interface Types

If you have an object reference of interface type, it's possible to distinguish the underlying class type, using the "is" operator, as in the following:

```
using System;

public interface IDummy {}

public class ClassA : IDummy {}

public class ClassB : IDummy {}

public class TestDemo {
    static void f(IDummy obj) {
        if (obj is ClassA)
            Console.WriteLine("found a ClassA object");
    }

    public static void Main() {
        IDummy obj1 = new ClassA();
        f(obj1);

        IDummy obj2 = new ClassB();
        f(obj2);
    }
}
```

This technique is useful for performance reasons – for example, if you need to find out whether an IList reference actually refers to an ArrayList, a LinkedList, or something else.

It's also useful at times to define marker interfaces, empty interfaces that serve only to distinguish a particular class that implements them. Here's an illustration:

```
using System;

public interface IDummy {}

public class ClassA : IDummy {}

public class ClassB {}

public class MarkerDemo {
    static void f(object obj) {
        if (obj is IDummy)
            Console.WriteLine("found an IDummy object");
    }

    public static void Main() {
        ClassA obj1 = new ClassA();
        f(obj1);

        ClassB obj2 = new ClassB();
```

```
        f(obj2);
    }
}
```

You can use this technique to give a group of classes a particular property that can be distinguished at runtime.

Interfaces are quite useful in specifying required behavior, and they enable you to program in terms of high-level types without having to get into implementation details.

# practical perl

**by Adam Turoff**

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

*ziggy@panix.com*

## Integration with Inline

Perl is a great language, but there are some things that are best left to a compiled language like C. This month, we take a look at the Inline module, which eases the process of integrating compiled C code into Perl programs.

Perl exists to make easy things easy and hard things possible. If you are writing programs using nothing but Perl, thorny issues like memory management just go away. You can structure your program into a series of reusable Perl modules and reuse some of the many packages available from CPAN. Things start to break down if you need to use a C library that does not yet have a Perl interface. Things get hard when you need to optimize a Perl sub by converting it to compiled C code.

Languages like Perl, Java, and C# focus on helping you program *within* a managed runtime environment. Reusing C libraries cannot be done entirely within these environments. Each of these platforms offers an "escape hatch" for those rare occasions when compiled code is necessary. In Java, the Java Native Interface (JNI) serves this purpose. In C#/.Net, programs can be linked to "unmanaged code," compiled libraries that live outside the .Net environment. In Perl, integration with external libraries is performed using XSubs and the esoteric XS mini-language.

Linking compiled code into Perl, Java, or .Net is necessary for a minority of projects. It is one of those "hard things that should be possible." Compared to Perl Java and .Net, Perl's XS interface is the oldest and admittedly the least easy to use. XS is a mix of C, C macros, and Perl API functions that are preprocessed to generate a C program. The resulting C source is then compiled to produce a shared object file that is dynamically linked into Perl on demand, where it provides some Perl-to-C interface glue and access to other compiled code, such as a library to manipulate images, an XML parser, or a relational database client library.

Creating an XS program is a little tricky. The mini-language itself is documented in the perlxs manual page and the perlxstut tutorial that come with Perl. XS programs may need to call Perl API functions, which are documented in the perlguts and perlapi manual pages. You can find more information in books like *Programming Perl*, *Writing Perl Modules for CPAN*, and *Extending and Embedding Perl*.

To create simple XS wrappers around compiled libraries, start by preprocessing a C header file with the h2xs tool and write additional XS wrapper functions as necessary. Another common approach uses swig to create the necessary wrapper code without using XS explicitly. If you are knowledgeable about Perl internals, you might avoid both approaches and write XS interface code from scratch.

Writing XS wrappers is tricky, and the skill is difficult to learn. Many long-time Perl programmers avoid XS because of its complexity. The state of XS is one of the factors that led the Perl development team to start the Perl 6 project. One of the goals behind Perl 6 is the creation of a new runtime engine, Parrot, that provides a substantially simplified interface for integrating with compiled code.

### Enter Inline::C

One day at the Perl Conference in 2000 (shortly after the Perl 6 project was announced), Brian Ingerson had an epiphany. Linking Perl to compiled C programs is one of those "hard things