C# overloaded operators

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

In our examination of the C# programming language thus far, we've seen that classes are a basic design and structuring tool. For example, you might have an application that uses a lot of X,Y points, and you could implement a Point class using C# language features. Instances (objects) of this class would represent specific points like 123,456.

Classes bring together both data (such as a pair of integers to represent points) and operations on that data (e.g., comparing one point to another). The operations are called methods, and in this column we'll look at how methods can be specified using operator names.

The idea is that a method's name can be something like == instead of Equals, or + instead of add, and using such names leads to a natural way of expressing operations on objects.

An Example

Let's look at an example, using a Point class to illustrate the idea of overloading:

```
using System;

public class Point {
    public readonly int x;
    public readonly int y;

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

    public static bool operator==(Point p1, Point p2) {
        return p1.x == p2.x && p1.y == p2.y;
    }

    public static bool operator!=(Point p1, Point p2) {
        return !(p1 == p2);
    }
}
```

```
public class Driver {
    public static void Main(string[] args) {
        Point point1 = new Point(10, 20);
        Point point2 = new Point(20, 30);
        Point point3 = new Point(10, 20);

        if (point1 == point2)
            Console.WriteLine("point1 == point2");

        if (point1 == point3)
            Console.WriteLine("point1 == point3");

        if (point1 != point2)
            Console.WriteLine("point1 != point2");
    }
}
```

This class defines a couple of public readonly fields, used to represent particular X,Y values. Readonly means that the fields are initialized in the constructor, and can then be read but not written by users of the class's objects. C# properties can also be used to achieve a similar end; they are a hybrid of data fields and methods.

The class also defines two operator methods, operator== and operator!=. These methods take two objects of Point type, and thus it is possible to say things like:

```
if (point1 == point2)
    ...

if (point3 != point4)
    ...
```

and provide a customized definition of what == and != mean for a given class. In the Point example, two points are equal if their X,Y values are the same, and inequality is defined simply as not being equal.

Defining == in this way might seem pretty obvious. But in real-world situations, equality can be defined in many ways. For example, if the X,Y points are represented as double values instead of integers, it might make sense to consider two points equal if the values are close to each other, say within 0.001%, rather than exactly the same.

We can define operator== with any semantics we like. For example, we can swap the bodies of the operator== and operator!= methods, and thereby invert the semantics. But doing so violates a fundamental rule of operator overloading – using such operators in a confusing way can make programs impossible to read and comprehend. It's possible to create your own reality using overloaded operators, a reality incomprehensible to others.

C# requires that the == and != operators be overloaded as a unit. If one is overloaded, the other must be as well.

A C# compiler may give a warning for the code above, saying that operator== is overloaded, but there is no Equals method specified. What does this mean? Equals is a method in the root class (System.Object), and typically you want to override it to provide class-specific behavior. Since C# is designed to interoperate with other languages, and those languages may not have operator overloading but may wish to call a C# Equals method, it's a good idea also to define Equals if operator== is defined.

This can be done by adding some additional lines of code:

```
public override bool Equals(object obj) {
    if (!(obj is Point))
        return false;
    return this == (Point)obj;
}

public override int GetHashCode() {
    return x ^ y;
}

public override string ToString() {
    return String.Format("({0},{1})", x, y);
}
```

We have defined Equals in terms of the == operator already specified above. GetHashCode and ToString are two other System.Object methods that are typically overridden as well, and we have also provided implementations of them.

Conversion Operators

You can also specify conversion operators in the C# classes you design. Such operators are used when converting from one data type to another.

For example, suppose that we define another Point class, one that represents X,Y values using unsigned 32-bit integers. An alternate representation of such points might be a single unsigned 64-bit integer, with the two 32-bit values stored in the two halves of the larger integer. Here's some code that shows how this idea can be implemented:

```
using System;

public class Point {
    public readonly uint x;
    public readonly uint y;

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

    public Point(ulong val) {
        x = (uint)(val >> 32);
        y = (uint)(val & 0xffffffffUL);
    }

    public static implicit operator ulong(Point p) {
        return ((ulong)p.x << 32) | p.y;
    }
}

public class Driver {
    public static void Main(string[] args) {
        Point p1 = new Point(123456, 234567);

        ulong pt = p1;

        Point p2 = new Point(pt);

        Console.WriteLine(p2.x + " " + p2.y);
    }
}
```

This class defines the usual constructor that takes an X,Y pair of values, along with a constructor that takes a single 64-bit value. The class also defines a method:

```
implicit operator ulong(Point p)
```

Such a method supports operations such as:

```
Point point1 = new Point(123, 456);

ulong p = point1;
```

that is, automatic conversion from a Point object to an unsigned long value.

The implicit specifier is used in declaring the method. If we'd used the explicit specifier instead, we would then need to say:

```
ulong p = (ulong)point1;
```

This situation is analogous to converting a long primitive value to a short value; some languages require an explicit cast, because the conversion may not be possible without data loss.

Indexers

A final example of C# operator overloading illustrates what is called an indexer. The idea is that you might have a class whose objects represent databases or tables or something, and it would be natural to overload the [] operator to represent lookup in the database or table.

Here's an example of how indexers are used:

```
using System;
using System.Collections;

public class Index {
    private string[,] list = new string[,] {
        {"jane jones",  "123-4567"},
        {"tom garcia",  "234-5678"},
        {"mildred smither", "345-6789"}
    };
```

```
    public string this[string index] {
        get {
            int listlen = list.GetUpperBound(0);

            for (int i = 0; i <= listlen; i++) {
                if (list[i,0] == index)
                    return list[i,1];
            }

            return null;
        }
    }
}

public class Driver {
    public static void Main(string[] args) {
        Index phonelist = new Index();

        string num = phonelist["tom garcia"];

        Console.WriteLine(num);
    }
}
```

This demo implements a simple phone list lookup scheme.

The key line of code in this example is:

```
public string this[string index] { ... }
```

This says that when [] is applied to objects of the Index class, the get and set code should be executed to actually do the indexing. In this example, we specify a string argument to [] that is used as the key for lookup, but any type of argument is allowed, and you can even use multiple arguments – for example, to implement virtual two-dimensional arrays.

The syntax is very similar to what is used for C# properties. The get code is invoked when obj[index] is used in an rvalue context, and the set logic (which we do not define) is executed when obj[index] is used as an lvalue.

Operator overloading is a powerful technique that you might want to use in your C# programs.