Operator Overloading

Consider the Fraction Class

On object in the class Fraction encapsulates two integer variables designated by numer and denom. These two variables together symbolize and contain the rational number that we typically think of as numer/denom.

For Example:

5/2, 3/5, 2/9, 100/3, etc.

Encapsulated Features in Fraction

In designing the constructors and the other methods in the class, we ensured that the following always hold:

- A zero never occurs in the denom;
- A Fraction object is always in reduced terms;
- The denom is always positive. In other words, a negative Fraction is encapsulated in the numer only.

Fraction Arithmetic

We wrote a method to add two Fractions, and if C and B are Fractions, their sum is computed by the expression C.Add(B);

Add is a non-static method. C (or always the object to the left of the dot operator) is the object that calls the Add method, and B is the argument. Their sum is computed and returned. C and B are not changed in the call.

Review the definition of Add. Why is it not static?

A more natural way to add.

Since we are accustomed to doing Fraction arithmetic with +, -, *, and /, it would be more natural to write C+B instead of C.Add(B).

This is easily accomplished by “overloading” the + operator for the Fraction class.

The next slide shows the overloaded method.

Overloaded +

```
public static Fraction operator+(
                          Fraction X,
                          Fraction Y)
{
  return X.Add(Y);
}
```

We are making use of the Add method already written. If it didn't exist we could include the code here using X.numer, X.denom, Y.numer and Y.denom. But we might as well use what's already done to make life easier.

What happens?

If C and B are Fraction objects, then the expression C + B invokes a call to the overloaded operator +, passing C to X and B to Y in the parameter list of the method operator+. The overloaded operator+ then invokes the method Add passing along references to the appropriate objects. Again, if Add didn't exist, we could do the work right in operator+ and return the value.

Adding int's and Fractions

Could we use an expression N + C, where N is an integer and C is a Fraction?

Not yet. But we can add another overloaded method that will do it.

Another Overloaded +

```
public static Fraction operator+
                (int N, Fraction C)
{
  return (new Fraction(N))+C;
}
```

Observation!

In the previous overloaded method, we merely use one of the Fraction class constructors to create a Fraction from N and then use the already defined overloaded+ for two Fraction objects.

Will this work for C + N? No. But it's easy to add a third operator+ that takes a Fraction on the left and an integer on the right.

Other Operators?

Many operators can be overloaded. For the Fraction class, we could overload the other arithmetic operators -, *, and /. In doing so, we could, just as with +, use Sub, Mult, and Div that have already been written.

Overloading Relational Operators

Suppose we want to know if one Fraction object is less than another in the usual meaning.

How could we write, for example C < B?

The overloaded operator < is shown on the next slide.

operator <

```
public static bool operator<
            (Fraction X, Fraction Y)
{
  return X.numer*Y.denom <
         Y.numer*X.denom);
}
```

Here we use the fact that a/b < c/d if and only if ad<bc, as long as d and b are positive. (Multiplying both sides of an inequality by a positive number preserves the inequality.)

A rule to follow

When overloading relational operators, they must be overloaded in pairs. For example if you overload <, you must overload >. Similarly for <= and >=; and == and !=;

Comment

If you overload operator == and operator !=, you will receive two warnings when the program is compiled.

FractionClass.cs(3,7): warning CS0660: 'Fraction' defines operator == or operator != but does not override Object.Equals(object o)

FractionClass.cs(3,7): warning CS0661: 'Fraction' defines operator == or operator != but does not override Object.GetHashCode()

These will be discussed briefly; however, they are beyond what we need to worry about at this point. Hashing is a subject for higher-level computer science course.