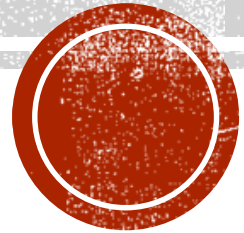
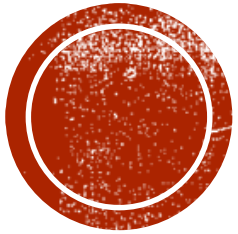
OOP WEEK 4

GVC Ths. Nguyễn Minh Đạo

BM CNPM - Khoa CNTT - ĐHSPKT TP.HCM





OPERATOR OVERLOADING

OVERVIEW

- It is a design goal of C# that user-defined classes have all the functionality of built-in types.
- For example, suppose you have defined a type to represent fractions.
- Ensuring that this class has all the functionality of the built-in types means that you must be able to perform arithmetic on instances of your fractions (e.g., add two fractions, multiply, etc.) and convert fractions to and from built-in types such as integer (int).

OVERVIEW (CONT.)

- You could, of course, implement methods for each of these operations and invoke them by writing statements such as:
  - Fraction theSum = firstFraction.Add(secondFraction);
- Although this will work, it is ugly and not how the built-in types are used. It would be much better to write:
  - Fraction theSum = firstFraction + secondFraction;
- Statements like this are intuitive and consistent with how built-in types, such as int, are added.

OVERVIEW (CONT.)

- In this chapter you will learn techniques for adding standard operators to your user-defined types.
- You will also learn how to add conversion operators so that your user-defined types can be implicitly and explicitly converted to other types.

USING THE OPERATOR KEYWORD

- In C#, operators are static methods whose return values represent the result of an operation and whose parameters are the operands.
- When you create an operator for a class you say you have "overloaded" that operator, much as you might overload any member method.
- Thus, to overload the addition operator (+) you would write:
  - public static Fraction operator+(Fraction lhs, Fraction rhs)

USING THE OPERATOR KEYWORD (CONT.)

- The C# syntax for overloading an operator is to write the word operator followed by the operator to overload. The operator keyword is a method modifier.
- Thus, to overload the addition operator (+), write **operator+**.
- When you write:
  - Fraction theSum = firstFraction + secondFraction;
- the overloaded **+** operator is invoked, with the first Fraction passed as the first argument, and the second Fraction passed as the second argument.

USING THE OPERATOR KEYWORD (CONT.)

- When the compiler sees the expression:
  - firstFraction + secondFraction
- it translates that expression into:
  - Fraction.operator+(firstFraction, secondFraction)
- The result is that a new Fraction is returned, which in this case is assigned to the Fraction object named theSum.

CREATING USEFUL OPERATORS

- Operator overloading can make your code more intuitive and enable it to act more like the built-in types.
- It can also make your code unmanageable, complex, and obtuse if you break the common idiom for the use of operators.
- Resist the temptation to use operators in new and idiosyncratic ways.

CREATING USEFUL OPERATORS (CONT.)

- For example, although it might be tempting to overload the increment operator (++) on an employee class to invoke a method incrementing the employee's pay level, this can create tremendous confusion for clients of your class.
- It is best to use operator overloading sparingly, and only when its meaning is clear and consistent with how the built-in classes operate.

LOGICAL PAIRS

- It is quite common to overload the equals operator (==) to test whether two objects are equal (however equality might be defined for your object).
- C# insists that if you overload the equals operator, you must also overload the not-equals operator (!=).
- Similarly, the less-than (<) and greater-than (>) operators must be paired, as must the less-than or equals (<=) and greater-than or equals (>=) operators.

THE EQUALS OPERATOR

- If you overload the equals operator (==), it is recommended that you also override the virtual **Equals**( ) method provided by object and route its functionality back to the equals operator.
- This allows your class to be polymorphic and provides compatibility with other .NET languages that do not overload operators (but do support method overloading).

THE EQUALS OPERATOR (CONT.)

- The object class implements the **Equals**( ) method with this signature:
  - public override bool Equals(object o)
- By overriding this method, you allow your Fraction class to act polymorphic with all other objects.
- Inside the body of **Equals**( ), you will need to ensure that you are comparing with another Fraction, and if so you can pass the implementation along to the equals operator definition that you've written.

THE EQUALS OPERATOR (CONT.)

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

- The is operator is used to check whether the runtime type of an object is compatible with the operand (in this case, Fraction).
- Thus o is Fraction will evaluate true if o is in fact a type compatible with Fraction.

CONVERSION OPERATORS

- C# converts int to long **implicitly**, and allows you to convert long to int **explicitly**.
- The conversion from int to long is **implicit** because you know that any int will fit into the memory representation of a long.
- The reverse operation, from long to int, must be **explicit** (using a cast) because it is possible to lose information in the conversion:

```
int myInt = 5;
long myLong;
myLong = myInt;          // implicit
myInt = (int) myLong;    // explicit
```

EXAMPLE

- Example illustrates how you might implement implicit and explicit conversions, and some of the operators of the Fraction class
- You can place a breakpoint on each of the test statements, and then step into the code, watching the invocation of the constructors as they occur.
- When you compile this example, it will generate some warnings because GetHashCode( ) is not implemented.

```
using System;

public class Fraction
{
    public Fraction(int numerator, int denominator)
    {
        Console.WriteLine("In Fraction Constructor(int, int)");
        this.numerator=numerator;
        this.denominator=denominator;
    }

    public Fraction(int wholeNumber)
    {
        Console.WriteLine("In Fraction Constructor(int)");
        numerator = wholeNumber;
        denominator = 1;
    }
```

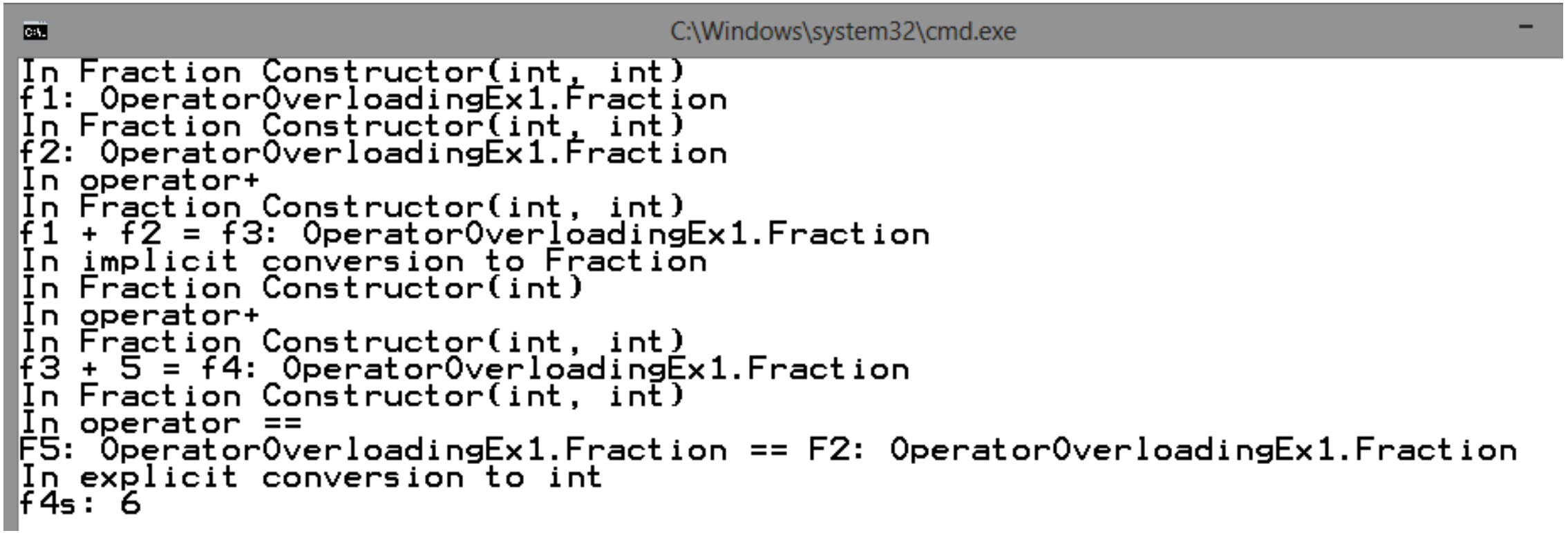
```
public static implicit operator Fraction(int theInt)
{
    System.Console.WriteLine("In implicit conversion to Fraction");
    return new Fraction(theInt);
}

public static explicit operator int(Fraction theFraction)
{
    System.Console.WriteLine("In explicit conversion to int");
    return theFraction.numerator /
        theFraction.denominator;
}
```

```
public static bool operator==(Fraction lhs, Fraction rhs)
{
    Console.WriteLine("In operator ==");
    if (lhs.denominator == rhs.denominator &&
        lhs.numerator == rhs.numerator)
    {
        return true;
    }
    // code here to handle unlike fractions
   return false;
}

public static bool operator !=(Fraction lhs, Fraction rhs)
{
    Console.WriteLine("In operator !=");

    return !(lhs==rhs);
}
```

```
public override bool Equals(object o)
{
    Console.WriteLine("In method Equals");
    if (! (o is Fraction) )
    {
        return false;
    }
    return this == (Fraction) o;
}

public override string ToString( )
{
    String s = numerator.ToString( ) + "/" +
        denominator.ToString( );
    return s;
}
```

```
public static Fraction operator+(Fraction lhs, Fraction rhs)
{
    Console.WriteLine("In operator+");
    if (lhs.denominator == rhs.denominator)
    {
        return new Fraction(lhs.numerator+rhs.numerator,
            lhs.denominator);
    }

    // simplistic solution for unlike fractions
    // 1/2 + 3/4 == (1*4) + (3*2) / (2*4) == 10/8
    int firstProduct = lhs.numerator * rhs.denominator;
    int secondProduct = rhs.numerator * lhs.denominator;
    return new Fraction(
        firstProduct + secondProduct,
        lhs.denominator * rhs.denominator
    );

}
```

```
public class Tester
{
    static void Main( )
    {
        Fraction f1 = new Fraction(3,4);
        Console.WriteLine("f1: {0}", f1.ToString( ));

        Fraction f2 = new Fraction(2,4);
        Console.WriteLine("f2: {0}", f2.ToString( ));

        Fraction f3 = f1 + f2;
        Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString( ));

        Fraction f4 = f3 + 5;
        Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString( ));


         Fraction f5 = new Fraction(2,4);
         if (f5 == f2)
         {
             Console.WriteLine("F5: {0} == F2: {1}",
                 f5.ToString( ),
                 f2.ToString( ));
         }

    }
}
```


C:\Windows\system32\cmd.exe
In Fraction Constructor(int, int)
f1: OperatorOverloadingEx1.Fraction
In Fraction Constructor(int, int)
f2: OperatorOverloadingEx1.Fraction
In operator+
In Fraction Constructor(int, int)
f1 + f2 = f3: OperatorOverloadingEx1.Fraction
In implicit conversion to Fraction
In Fraction Constructor(int)
In operator+
In Fraction Constructor(int, int)
f3 + 5 = f4: OperatorOverloadingEx1.Fraction
In Fraction Constructor(int, int)
In operator ==
F5: OperatorOverloadingEx1.Fraction == F2: OperatorOverloadingEx1.Fraction
In explicit conversion to int
f4s: 6


LIVE
DEMO


Q
A


Thank
You!