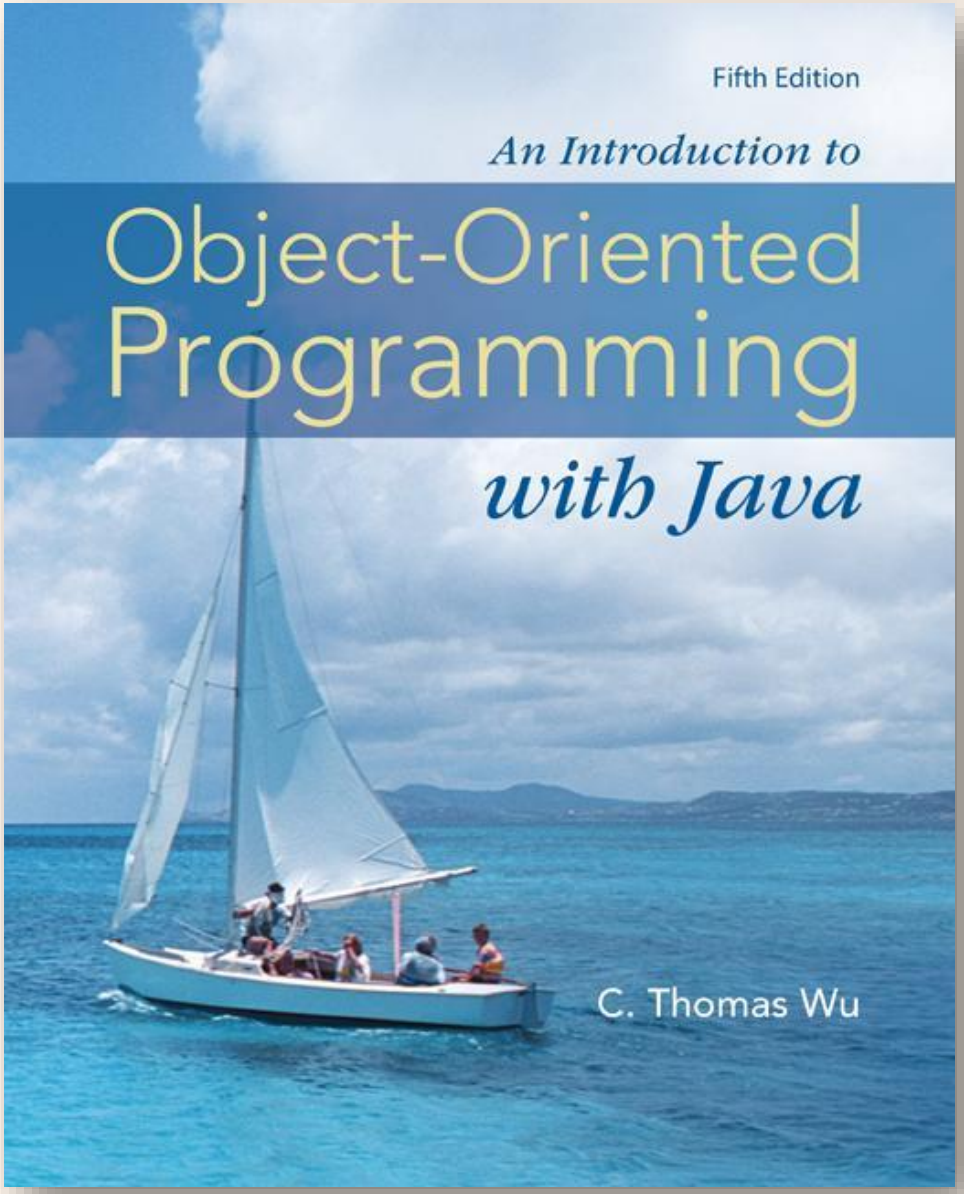



# Chapter 7

## Defining Your Own Classes

## Part 2

Animated Version

# Objectives

- After you have read and studied this chapter, you should be able to
  - Describe how objects are returned from methods
  - Describe how the reserved word this is used
  - Define overloaded methods and constructors
  - Define class methods and variables
  - Describe how the arguments are passed to the parameters using the pass-by-value scheme
  - Document classes with javadoc comments
  - Organize classes into a package

# Returning an Object from a Method

- As we can return a primitive data value from a method, we can return an object from a method also.
- We return an object from a method, we are actually returning a reference (or an address) of an object.
  - This means we are not returning a copy of an object, but only the reference of this object

# Sample Object-Returning Method

- Here's a sample method that returns an object:

```
public Fraction simplify( ) {

    Fraction simp;

    int num   = getNumberator();
    int denom = getDenominator();
    int gcd   = gcd(num, denom);

    simp = new Fraction(num/gcd, denom/gcd);

    return simp;
}
```

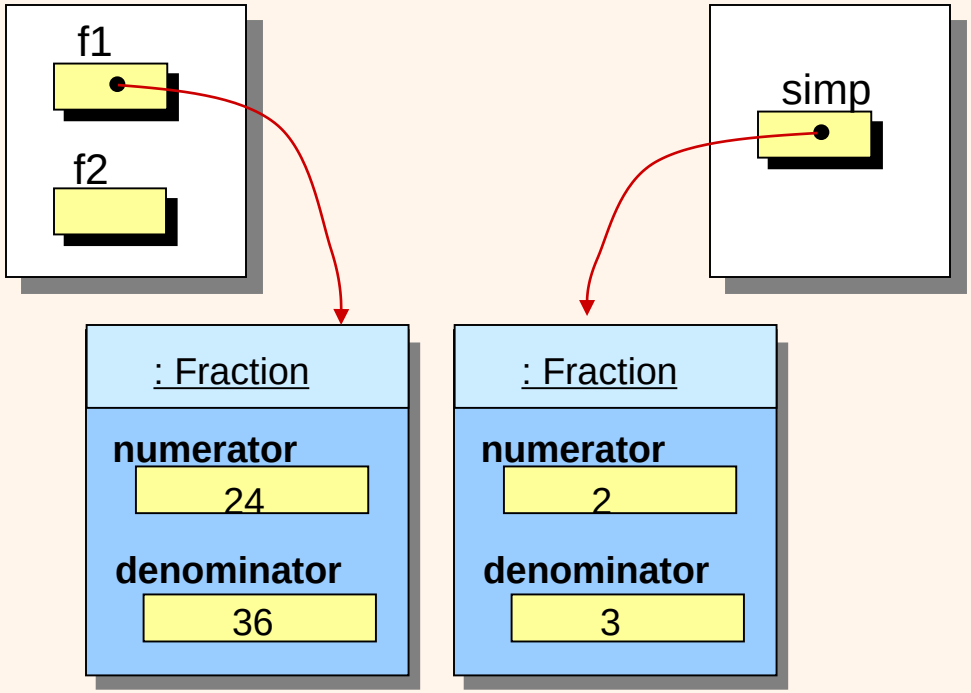
Return type indicates the class of an object we're returning from the method.

Return an instance of the Fraction class

# A Sample Call to simplify

```
f1 = new Fraction(24, 26);

f2 = f1.simplify();
```

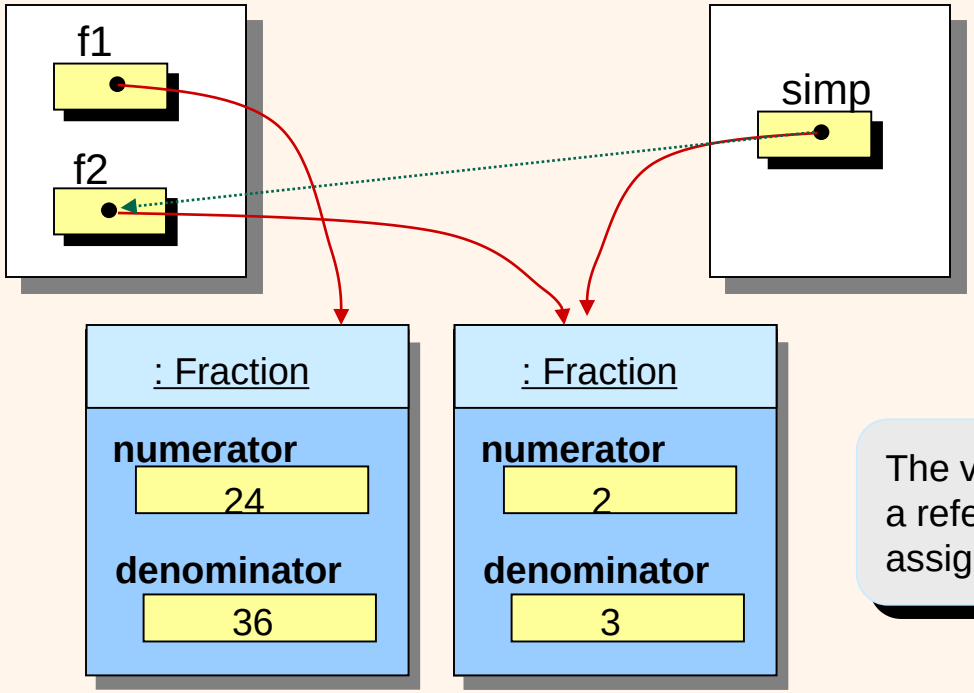
```
public Fraction simplify( ) {

  int num   = getNumerator();
  int denom = getDenominator();
  int gcd   = gcd(num, denom);

  Fraction simp = new
       Fraction(num/gcd, denom/gcd);

  return simp;
}
```



# A Sample Call to simplify (cont'd)

```
f1 = new Fraction(24, 26);

f2 = f1.simplify();
```

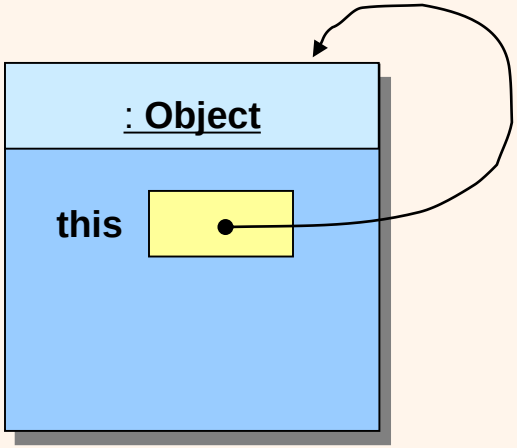
```
public Fraction simplify( ) {

  int num   = getNumerator();
  int denom = getDenominator();
  int gcd   = gcd(num, denom);

  Fraction simp = new
       Fraction(num/gcd, denom/gcd);

  return simp;
}
```




The value of simp, which is a reference, is returned and assigned to f2.

# Reserved Word this

- The reserved word this is called a *self-referencing pointer* because it refers to an object from the object's method.

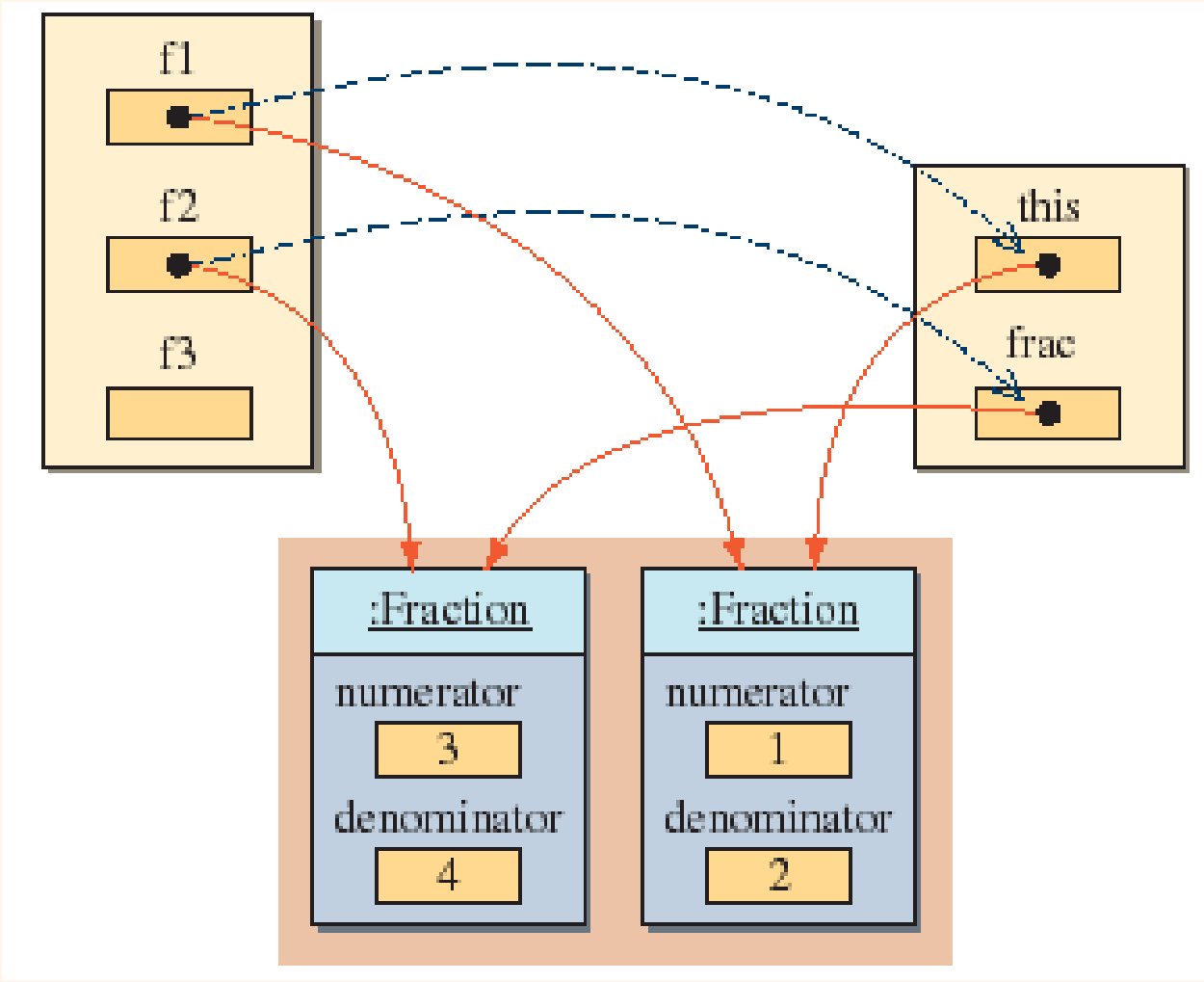



- The reserved word this can be used in three different ways. We will see all three uses in this chapter.

# The Use of this in the add Method

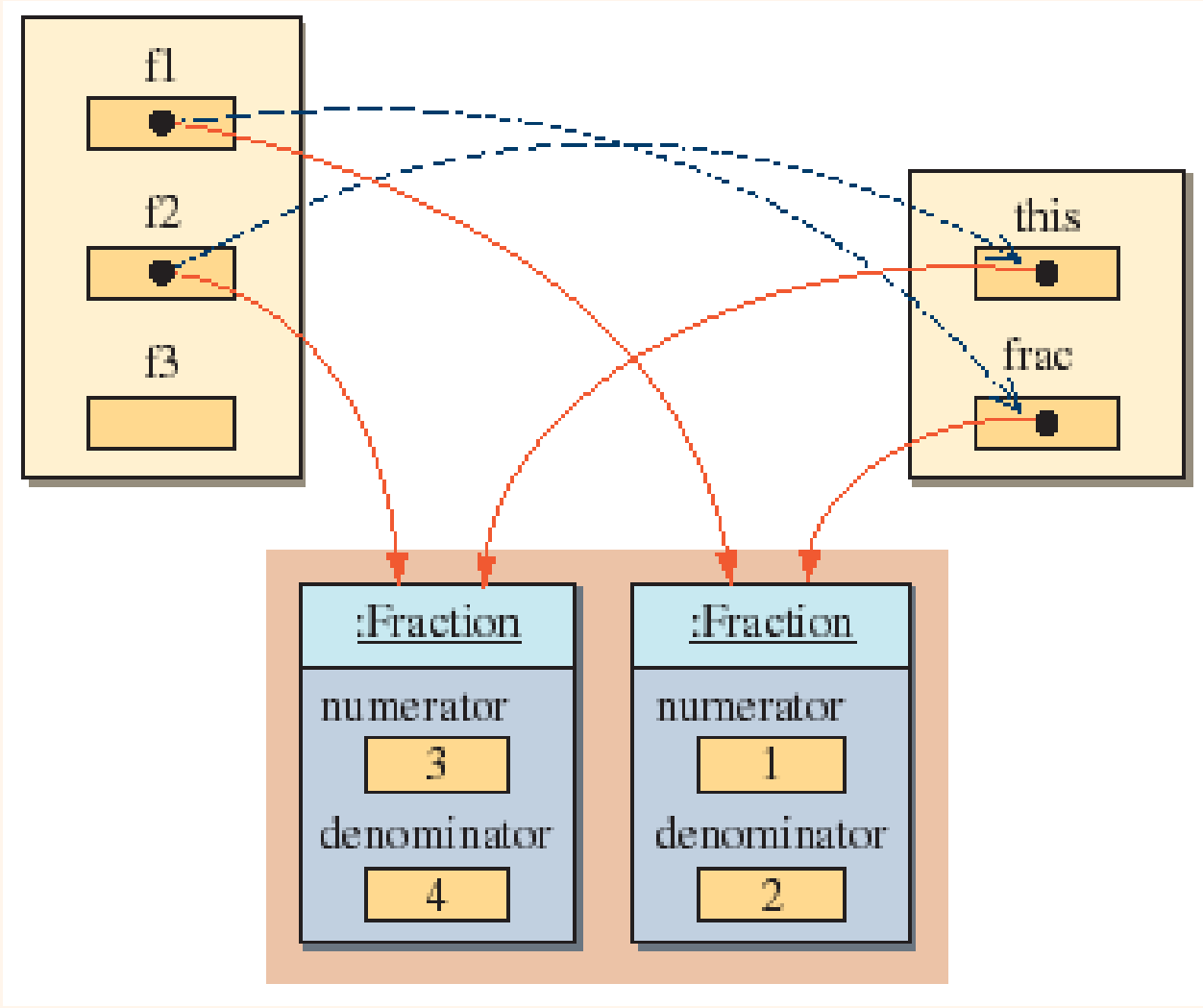
```
public Fraction add(Fraction frac) {

   int      a, b, c, d;
   Fraction sum;

   a = this.getNumerator();   //get the receiving
   b = this.getDenominator(); //object's num and denom

   c = frac.getNumerator();   //get frac's num
   d = frac.getDenominator(); //and denom

   sum = new Fraction(a*d + b*c, b*d);

   return sum;
}
```

# f3 = f1.add(f2)




Because f1 is the receiving object (we're calling f1's method), so the reserved word this is referring to f1.

# f3 = f2.add(f1)




This time, we're calling f2's method, so the reserved word this is referring to f2.

# Using this to Refer to Data Members

- In the previous example, we showed the use of this to call a method of a receiving object.
- It can be used to refer to a data member as well.

```
class Person {

   int  age;

    public void setAge(int val) {
       this.age = val;
   }
   . . .
}
```

# Overloaded Methods

- Methods can share the same name as long as
  - they have a different number of parameters (Rule 1) or
  - their parameters are of different data types when the number of parameters is the same (Rule 2)

```
public void myMethod(int x, int y) { ... }
public void myMethod(int x) { ... }
```

✓ Rule 1

```
public void myMethod(double x) { ... }
public void myMethod(int x) { ... }
```

✓ Rule 2

# Overloaded Constructor

- The same rules apply for overloaded constructors
  - this is how we can define more than one constructor to a class

```
public Person( ) { ... }
public Person(int age) { ... }
```

✓ Rule 1

```
public Pet(int age) { ... }
public Pet(String name) { ... }
```

✓ Rule 2

# Constructors and this

- To call a constructor from another constructor of the same class, we use the reserved word this.

```
public Fraction( ) {
    //creates 0/1
    this(0, 1);
}

public Fraction(int number) {
    //creates number/1
    this(number, 1);
}

public Fraction(Fraction frac) {
    //copy constructor
    this(frac.getNumerator(),
         frac.getDenominator());
}

public Fraction(int num, int denom) {
    setNumerator(num);
    setDenominator(denom);
}
```

# Class Methods

- We use the reserved word static to define a class method.

```
public static int gcd(int m, int n) {

   //the code implementing the Euclidean algorithm
}


public static Fraction min(Fraction f1, Fraction f2) {

   //convert to decimals and then compare

}
```

# Call-by-Value Parameter Passing

- When a method is called,
  - the value of the argument is passed to the matching parameter, and
  - separate memory space is allocated to store this value.
- This way of passing the value of arguments is called a *pass-by-value* or *call-by-value scheme*.
- Since separate memory space is allocated for each parameter during the execution of the method,
  - the parameter is local to the method, and therefore
  - changes made to the parameter will not affect the value of the corresponding argument.

# Call-by-Value Example

```
class Tester {
    public void myMethod(int one, double two ) {
        one = 25;
        two = 35.4;
    }
}
```

```
Tester tester;
int x, y;
tester = new Tester();
x = 10;
y = 20;
tester.myMethod(x, y);
System.out.println(x + " " + y);
```

produces

```
10 20
```

# Memory Allocation for Parameters

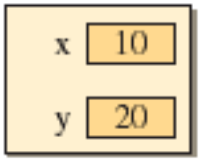
1

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

execution flow

```
public void myMethod( int one, double two ) {

   one = 25;
   two = 35.4;
}
```

at (1) before calling **myMethod**




state of memory

**Local variables do not exist before the method execution.**

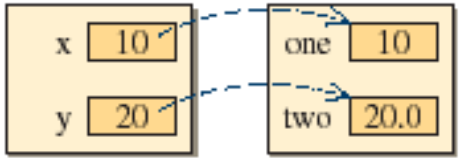
2

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

```
public void myMethod( int one, double two ) {
   one = 25;
   two = 35.4;
}
```

(2)

values are copied at (2)




**Memory space for myMethod is allocated, and the values of arguments are copied to the parameters.**

# Memory Allocation for Parameters (cont'd)

3

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

at (3) before return

| x | 10 |
|---|---|
| y | 20 |

| one | 25 |
|---|---|
| two | 35.4 |

```
public void myMethod( int one, double two ) {

   one = 25;
   two = 35.4;
}
```

(3)

**The values of parameters are changed.**

4

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

(4)

at (4) after **myMethod**

| x | 10 |
|---|---|
| y | 20 |

```
public void myMethod( int one, double two ) {

   one = 25;
   two = 35.4;
}
```

**Memory space for myMethod is deallocated, and parameters are erased. Arguments are unchanged.**

# Parameter Passing: Key Points

1. *Arguments are passed to a method by using the pass-by- value scheme.*
2. *Arguments are matched to the parameters from left to right.The data type of an argument must be assignment-compatible with the data type of the matching parameter.*
3. *The number of arguments in the method call must match the number of parameters in the method definition.*
4. *Parameters and arguments do not have to have the same name.*
5. *Local copies, which are distinct from arguments,are created even if the parameters and arguments share the same name.*
6. *Parameters are input to a method, and they are local to the method.Changes made to the parameters will not affect the value of corresponding arguments.*

# Main Point

- Java method calls are in every case *call by value* (and never *call by reference*). Even though an object reference can be passed, only a *copy* of such a variable is ever passed to a method (in other words, call by value). Call by value is resembles the incorruptible quality of pure consciousness – "fire cannot burn it, nor water wet it".

# Organizing Classes into a Package

- For a class A to use class B, their bytecode files must be located in the same directory.
  - This is not practical if we want to reuse programmer-defined classes in many different programs
- The correct way to reuse programmer-defined classes from many different programs is to place reusable classes in a package.
- A *package* is a Java class library.

# Creating a Package

- The following steps illustrate the process of creating a package name myutil that includes the Fraction class.

**1.** Include the statement

```
package myutil;
```

as the first statement of the source file for the Fraction class.

**2.** The class declaration must include the visibility modifier public as

```
public class Fraction {
   ...
}
```

**3.** Create a folder named myutil, the same name as the package name. In Java, the package must have a one-to-one correspondence with the folder.

**4.** Place the modified Fraction class into the myutil folder and compile it.

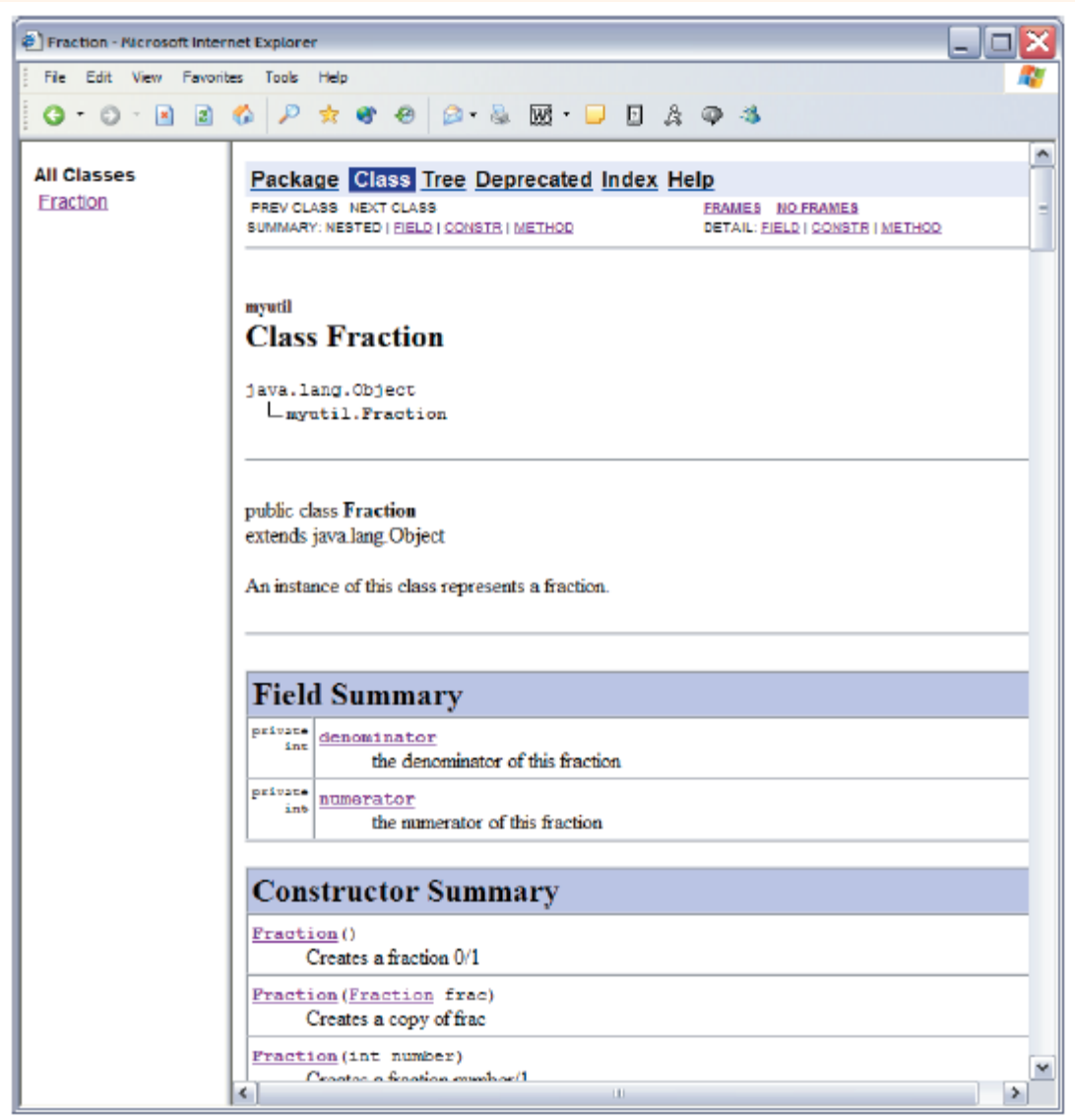
**5.** Modify the CLASSPATH environment variable to include the folder that contains the myutil folder.

# Using Javadoc Comments

- Many of the programmer-defined classes we design are intended to be used by other programmers.
  - It is, therefore, very important to provide meaningful documentation to the client programmers so they can understand how to use our classes correctly.
- By adding javadoc comments to the classes we design, we can provide a consistent style of documenting the classes.
- Once the javadoc comments are added to a class, we can generate HTML files for documentation by using the javadoc command.

# javadoc for Fraction

- This is a portion of the HTML documentation for the Fraction class shown in a browser.
- This HTML file is produced by processing the javadoc comments in the source file of the Fraction class.



# javadoc Tags

- The javadoc comments begins with /** and ends with */
- Special information such as the authors, parameters, return values, and others are indicated by the @ marker

  @param

  @author

  @return

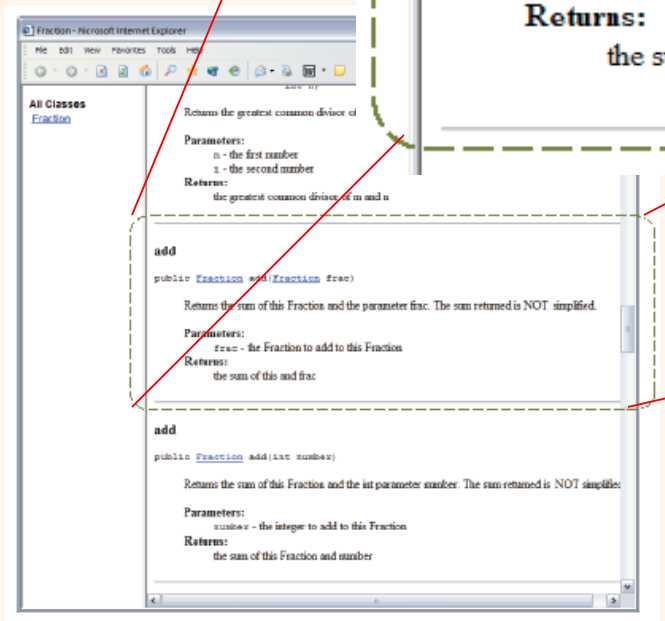
  etc

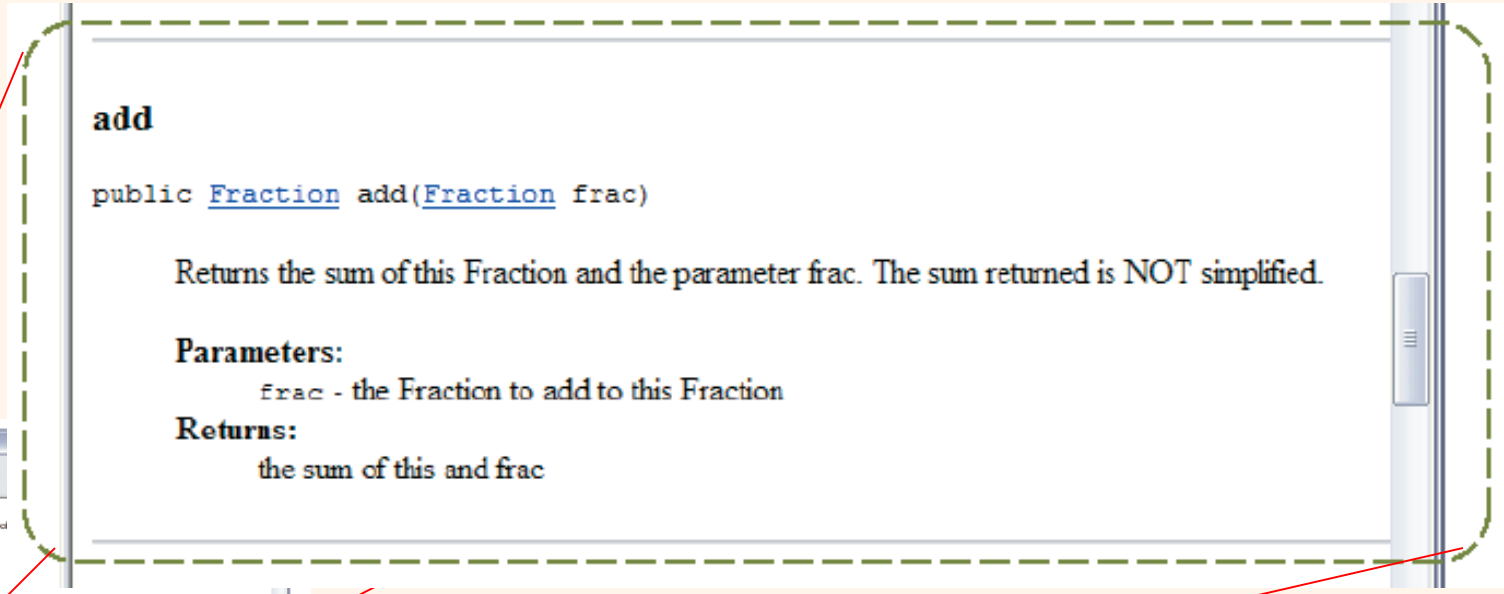
# Example: javadoc Source

```
. . .

/**
* Returns the sum of this Fraction
* and the parameter frac. The sum
* returned is NOT simplified.
*
* @param frac the Fraction to add to this
*             Fraction
*
* @return the sum of this and frac
*/
public Fraction add(Fraction frac) {
    ...
}
. . .
```

this javadoc will produce

# Example: javadoc Output



# javadoc Resources

- General information on javadoc is located at

  http://java.sun.com/j2se/javadoc

- Detailed reference on how to use javadoc on Windows is located at

  http://java.sun.com/j2se/1.5/docs/tooldocs/windows/javadoc.html