Classes and Objects in Java – Instance Methods

Objects can be more than just collections of data; they can also have functionality, implemented through methods. The methods can be grouped into two categories – *instance methods* and *class methods*.

Creating an Instance Method

Suppose we want to extend our `Fraction` class with a method that returns the size of the fraction. We will define the size as the absolute value (always positive) of the fraction, represented in decimal form. Consider the following implementation of the method:

```
class Fraction
{
  int num;
  int den;

  public double size ()
  {
    return Math.abs((double)num/den);
  }
}
```

There are many items to recall and note here:

1. The method does not have the modifier `static`, which has appeared in all previous methods until now. Any method without the `static` modifier is an *instance method*, while any method with the `static` modifier is a *class method*.

2. Recall the abs method from the Math class, `Math.abs()`, will return the absolute value (positive integer or decimal value) of any numeric value.

3. The `(double)` is a *cast*, which is used to force the quotient of two integers, `num/den`, to be calculated as a `double` (i.e., a decimal value will be generated).

4. In this method, we use `num` and `den` directly. We do not refer to `f.num` and `f.den`, or any other object. The association is *implicit* here that the `num` and `den` are those of the object currently associated with the method. In our first call, `f.size()`, the `size` method is called from the object `f`. In the second call, `g.size()`, the `size` method is called from the object `g`.

5. As an alternative, it is possible to refer to the *implicit object* in an *instance method* using the Java reserved word `this`. Thus it would have also been correct to program the method as:

   ```
   return Math.abs((double)this.num/this.den);
   ```

Although this particular instance method has no parameters, it is also possible (and common) for instance methods to have one or more parameters.

From our `main()` method, or another method, we might use the following code to test our new instance method:

```
Fraction f = new Fraction();
f.num = 2;
f.den = 3;
System.out.println(f.size());  // should be 0.666666 = 2/3
```

Creating an Instance Method with a Parameter List

Now let us consider comparing the size of two fractions. We will use the `size` method already discussed, a make use of the keyword `this` for clarity.

```
// compare two Fraction objects,
// return TRUE if calling fraction is larger, FALSE otherwise
public boolean isLargerThan (Fraction other)
{
     if (this.size() >= other.size())
          return true;
     else
          return false;
}
```

Assume we have previously declared and initialized `f` and `g`, both of type `Fraction`. We could test the `isLargerThan()` method by writing something like

```
f.num = 2;
f.den = 3;
g.num = 5;
g.den = 6;

// g is clearly larger than f
System.out.println(f.isLargerThan(g));  // prints false
System.out.println(g.isLargerThan(f));  // prints true
```

Classes and Objects in Java – Instance Methods

We will use the `size` method again to create a similar method, but the difference is the return value. This time, we return a *reference* to a `Fraction` object (the larger of the two).

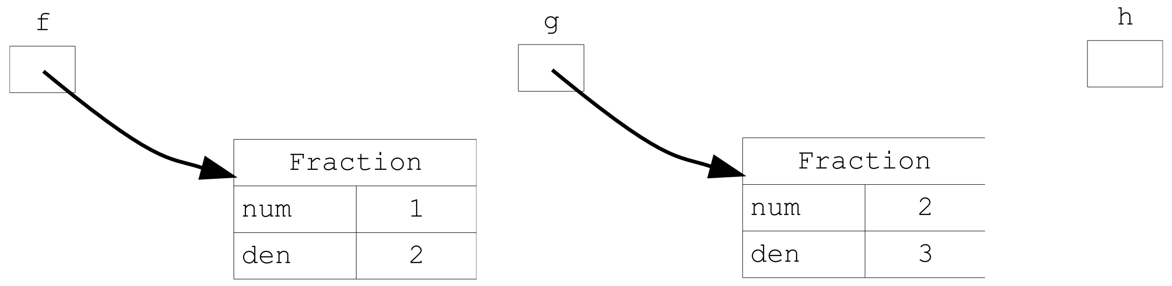
```
// compare two Fraction objects, return reference to larger object
public Fraction larger (Fraction other)
{
     if (this.size() >= other.size())
          return this;
     else
          return other;
}
```

Assume we have previously declared and initialized f, g, and h, all of type Fraction. We could use the larger method by writing a statement such as

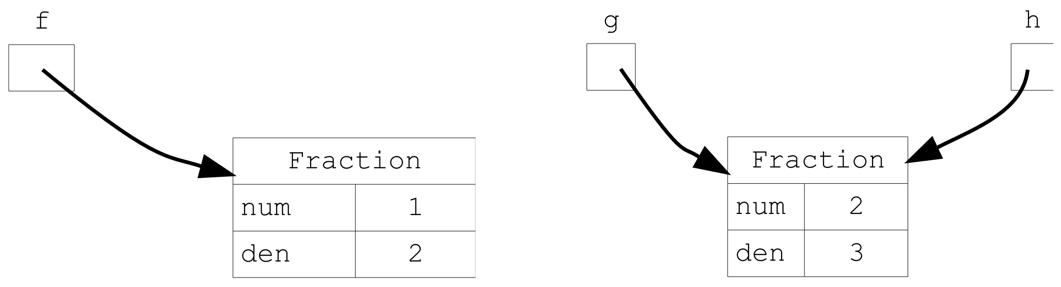
```
h = f.larger(g);
```

This statement would compare `f` to `g`, and whichever was larger would be assigned to `h`.

Suppose we had the following initial setup:



Clearly the size of `g` (0.667) is greater than the size of `f` (0.5), so h would refer to the same location in memory as `g`.



Creating an Instance Method with No Return Value

As with any other methods, instance methods can take the form of commands that do not return values.

Suppose we want to duplicate the *= Java operator for our Fraction class. Using this operator, the following two statements are equivalent:

```
x *= 2;    // multiply x by 2 and store the result in x
```

is equivalent to

```
x = x * 2; // multiply x by 2 and store the result in x
```

For fractions, multiplication involves numerator times numerator, and denominator times denominator. The required method might look like

```
// multiply the current fraction by the given fraction, p
// store the result in the current fraction
public void timesEquals(Fraction p)
{
     this.num *= p.num;
     this.den *= p.den;
}
```

If `f` and `g` are objects of type `Fraction`, then we could use this method by writing

```
f.timesEquals(g);
```

As a result, the fraction `f` is multiplied by the fraction `g`, and the result is stored (overwrites) `f`, while `g` is unchanged.

Passing Arguments by Value vs Passing Arguments by Reference

The previous example also illustrates a very important, yet very subtle, difference between using objects as parameters and using primitive data types (e.g., int, char, double) as parameters.

In our previous discussions of primitive data types, it was made very clear that modifying the value in the method has *no effect* on the original variable. In this case, we have passed only the *value* of the original variable (i.e., a copy is made). Any changes made to the copy do not affect the original.

When passing an object as a parameter, we also make a copy of the *value*. In this case, however, the value is a *reference*, or *address*, of some memory location. Although it is true that we cannot change the *reference value* of the original object, we can change the data stored at that particular address in memory.

Consider the following real-world example: You write your home address on a piece of paper (123 Main Street). Your friend wants to use that information, so you let him make a copy. If your friend accidentally, or intentionally, writes a new number on the paper, your original is still intact.

If, on the other hand, your friend reads the address and paints the house at that address purple, you now have a purple house. While it is true that your friend cannot change your home address, they can affect your home at that address. This is like passing an argument by reference.

Creating an Object within a Method

It is also possible for a method to create a new instance of an object in memory. Consider the following:

```
// multiply two fractions, return reference to the (new) result
public Fraction times (Fraction other)
{
      Fraction result = new Fraction();
      result.num = this.num * other.num;
      result.den = this.den * other.den;
      return result;
}
```

In this example, the `new Fraction()` creates an instance of a `Fraction` object in memory. When the method is complete, this object in memory persists. On the other hand, the variable result only exists inside the method block. A reference to the new object is passed to the calling code by the return statement, which might look like

```
Fraction f = g.times(h);
```

It is important to understand that `f` is a brand new object, completely separate from `g` and `h`. The only relationship to g and h is that the numerator and denominator of `f` was calculated using the numerators and denominators of `g` and `h`.

Questions & Exercises

1. Suppose that `p`, `q`, and `r` are all objects of type `Fraction`. What fraction would `r` represent after the statement
   `r = p.larger(q);`
   is executed, given that larger is the method previously discussed in this lesson.
   a) `p` represents $\frac{1}{3}$ and `q` represents $\frac{4}{5}$
   b) `p` represents $\frac{7}{-5}$ and `q` represents $\frac{-9}{-7}$
   c) `p` represents $\frac{5}{6}$ and `q` represents $\frac{-25}{-30}$
   d) `p` represents $\frac{-9}{-12}$ and `q` represents $\frac{-3}{4}$
   e) `p` represents $\frac{5}{8}$ and `q` represents $\frac{13}{20}$

2. Complete the definitions of the following instance methods for the `Fraction` class.

   a) `public void plusEquals (Fraction other)`
   Add two fractions and save the result in the calling object (this). See the `timesEquals` method previously discussed.

   b) `public Fraction plus (Fraction other)`
   Add two fractions and return the resulting fraction as its own object. See the `times` method previously discussed.

   c) `public void reduce ()`
   This method should reduce its implicit `Fraction` parameter to lowest terms. For example, if `f` represents $\frac{16}{24}$ would be change `f` so that it represents the fraction $\frac{2}{3}$.

Solutions

1. (a) q is larger, r is set to q
   (b) p is larger, r is set to p
   (c) size is the same, so r is set to p (the calling object)
   (d) size is the same, so r is set to p (the calling object)
   (e) q is larger, so r is set to q
2. (a)
```
public void plusEquals(Fraction other)
{
     this.num = this.num * other.den + other.num * this.den;
     this.den = this.den * other.den;
}
```
   (b)
```
public Fraction plus (Fraction other)
{
     Fraction result = new Fraction();
     result.num = this.num * other.den + other.num * this.den;
     result.den = this.den * other.den;
     return result;
}
```
   (c)
```
public void reduce ()
{
  // find the largest number that divides into
  // the numerator and denominator
  int gcf = 1;   // greatest common factor, 1 always works

  for (int i = 2; i <= this.num && i <= this.den; i++)
  {
    if (this.num % i = 0 && this.den % i = 0)
    {
      gcf = i;
    }
  }

  // divide numerator and denominator by GCM
  this.num = this.num / gcf;
  this.den = this.den / gcf;
}
```