COP 3330 – Object Oriented Programming in Java
Classes – Focus on Method Overloading

fraction Class

Our next example will be the fraction class. A fraction is defined as two integers: a numerator and denominator. Internally, we would like to always store our fraction in lowest terms with a positive denominator. In order to do this, we'll have to use a static method gcd, which computes the greatest common divisor of two integers. The code for this method will be recursive. If you don't understand exactly how it works that is okay. We'll explain recursion later in the course. For now, just understand that this method efficiently returns the largest common divisor of its two integer inputs. (We also assume that both inputs are non-negative.)

Here is the basic set up of the class with the constructor and the gcd method:

```
public class fraction {

    // Stores the reduced numerator and denominator of the fraction object.
    private int numerator;
    private int denominator;

    final private static double EPSILON = 0.0000001;

    public fraction(int n, int d) {
        numerator = n;
        denominator = d;
        if (denominator < 0) {
            numerator = -numerator;
            denominator = - denominator;
        }
        reduce();
      }

    public fraction(int value) {
        numerator = value;
        denominator = 1;
    }

    private void reduce() {
        int common = gcd(Math.abs(numerator), denominator);
        numerator /= common;
        denominator /= common;
    }

    private static int gcd(int a, int b) {
        if (b == 0) return a;
        return gcd(b, a%b);
    }

    // More methods.
}
```

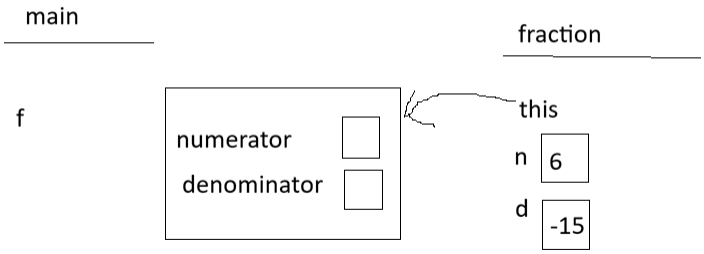
Constructor Overloaded

The constructor is overloaded meaning that there is more than one constructor provided. One constructor takes in two integers, both the numerator and denominator of the fraction object to be created. The other just takes in a single integer, representing the value of the fraction (so it's an integer). In the first case, it's possible the user might give us something like 12 for the numerator and 18 for the denominator, but for our purposes, we'd like to store this as 2 over 3 instead. The two private methods we need to carry out this reduction are gcd and reduce. As previously mentioned, gcd calculates the greatest common divisor between two integers. What reduce does is use gcd. Once the gcd is known, then both the numerator and denominator are divided by this common value.

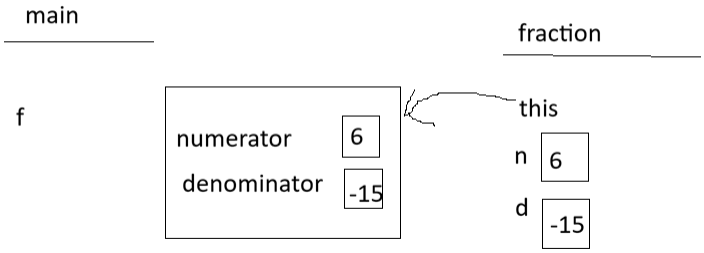
Let's look at a picture of what occurs when we execute the following line of code:

```
fraction f = new fraction(6, -15);
```

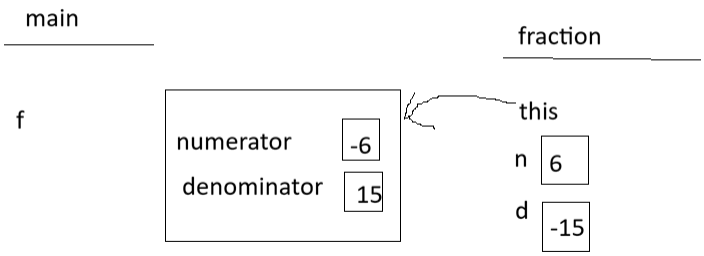
The constructor gets called with this attached to the newly created object and separate formal parameters n and d:



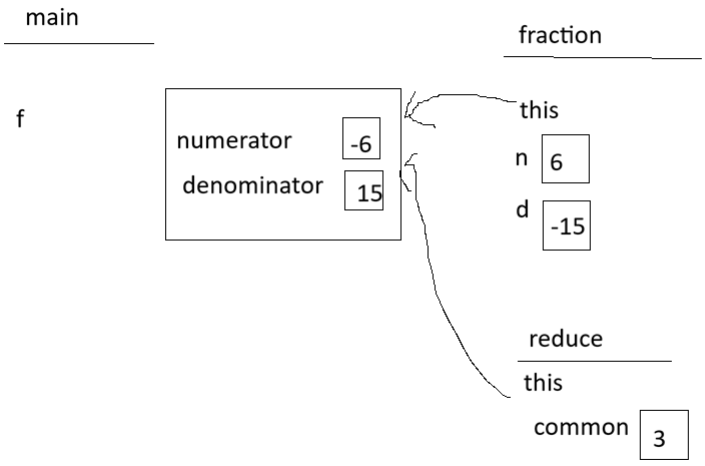
First we copy in 6 and -15 into this.numerator and this.denominator. Java knows that when we write numerator, we mean this.numerator. (When Java sees no local variables with this name, it looks to see if there are instance variables with this name.)



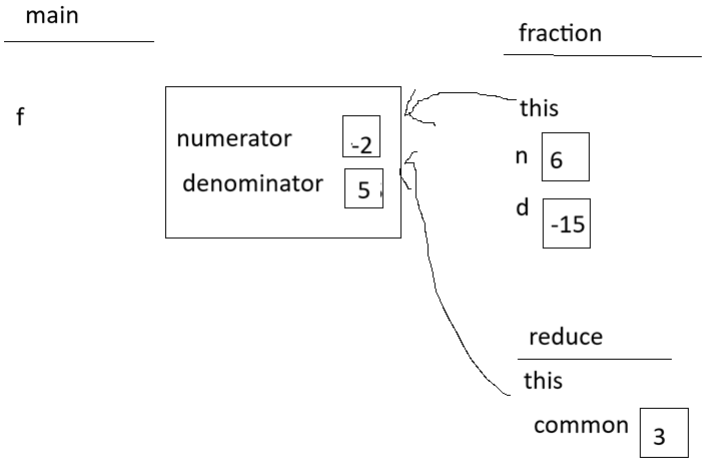
Then we check if the denominator is negative, if so we flip the signs of both the numerator and denominator, so now our picture looks like this:



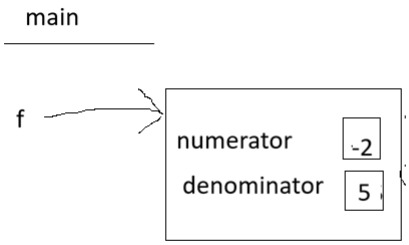
Then we call the reduce method. reduce then calls gcd which calls gcd a few times. When all is said and done, gcd will return 3. Here is the picture right after gcd returns 3:



Thus, notice that each instance method that gets called in a chain has a reference to the same object (of course we called the instance method on something other than this) and makes changes to that object as its code dictates. In this case, the reduce method will divide both the numerator and denominator of this object by 3 before completing:



Then, reduce completes at which point the constructor completes and returns a reference to this. When we get back to main, f gets set to this reference:



From this point on, we won't look at constructors in such detail. Typically, when constructors complete, they return a reference to the completed object. Really understanding the details of what occurs at each step can help speed up debugging when you write incorrect code inadvertently.

It's also possible that one might want to create a fraction object from an integer. That's why a second constructor is provided. This one's much easier to write. The numerator is just set to the formal parameter, the denominator is set to 1 and no reduction needs to occur since this is already in lowest terms.

Now, let's move onto the add method.

add Method

Perhaps the most obvious method we would like to provide users of the fraction class is the add method. This will taken in a second fraction object and return a new fraction object that is the sum of this and the formal parameter, f. Let's look at the code:

```
public fraction add(fraction f) {
    int num = this.numerator*f.denominator + this.denominator*f.numerator;
    int den = this.denominator*f.denominator;
    return new fraction(num, den);
}
```

Now that we have a working constructor, the add method is pretty easy! We symbolically figure out how to add fractions and store the new numerator and denominator. Notice that we don't worry about the reducing the fraction. We just get an obvious denominator that works. Why?

Because our constructor will do that already!!!

So, we just calculate the new numerator and denominator, send these to the constructor and it'll do the heavy lifting and create a new fraction object.

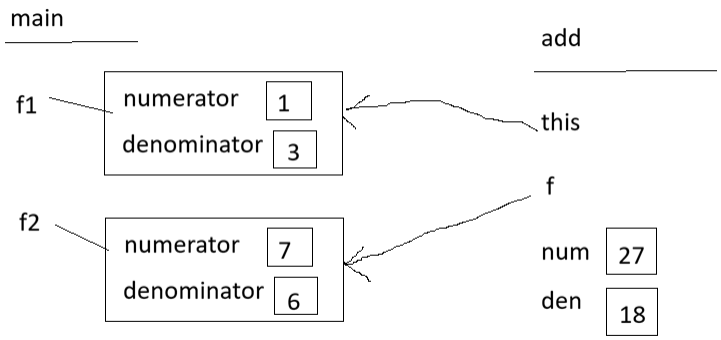
Let's look at two uses of this add method. Let f1 and f2 already be defined as follows:

```
fraction f1 = new fraction(1, 3);
fraction f2 = new fraction(7, 6);
```

It's fairly clear how this looks (I won't waste a picture on this!) Now, consider the line of code:

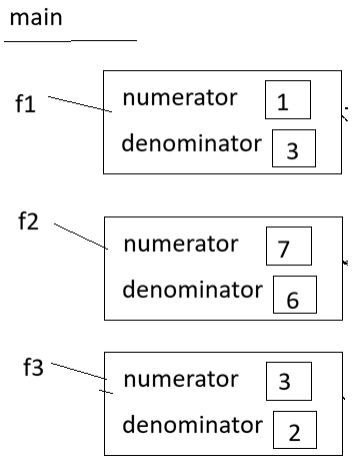
```
fraction f3 = f1.add(f2)
```

After add gets called and we execute the first two lines of code in add, here is our picture:



this is a reference to the object that the add method was called on, which in this case is f1. f is the formal parameter referring to the object the actual parameter, f2 from main was pointing to, num and den are local variables which we've computed based on the values in the instance variable that this and f are pointing to. One thing to notice is that computers solve problems pretty differently than humans. Any human immediately sees that 6 is a multiple of 3 so that we can just keep our denominator as 6 for our initial calculation, but coding this would be fairly annoying and not worth the hassle. The computer spends the same amount of time multiplying 1 and 1 as it does 13497 by 34212. A human does better with smaller numbers. Humans naturally run many different algorithms for many different cases but coding these in a computer is tedious. Rather, coming up with a simple solution that always works (product of denominators) is better in code, especially when our constructor always reduces its input to lowest terms!!!

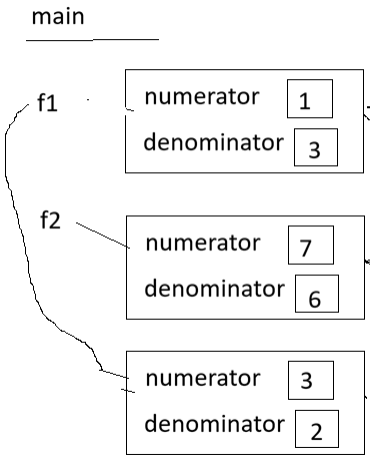
Thus, our next step is to call our constructor with 27 and 18, respectively. We've already gone through all the painstaking steps that occur, so instead of drawing these out, just note that the constructor will return a reference to a fraction equal to 3/2. Finally in main, we have f3 point to this newly returned fraction object:



Now, let's look at what WOULD have happened if we did

```
f1 = f1.add(f2)
```

Everything is the same until the return. So, the ONLY difference is that instead of a new reference, f3, pointing to the newly created object, we'll have the already existing reference f1, point to it:



Notice that NO reference is pointing to the object storing the fraction 1/3, so it's inaccessible to our program. The Java interpreter's garbage collector will eventually notice this and free the memory for this object so that the same memory can be used for other objects. In Java specifically, the programmer doesn't have to worry about this bit of memory management. It's entirely taken care of by the interpreter! (So eventually, the object storing 1/3 will just disappaear from our picture.)

It's important to understand how these two pictures work. Once you do, you should be about to debug 95% of the errors that programmers usually make when writing object oriented code.

Now, let's get to method overloading, which is supposed to be the topic of this lecture!

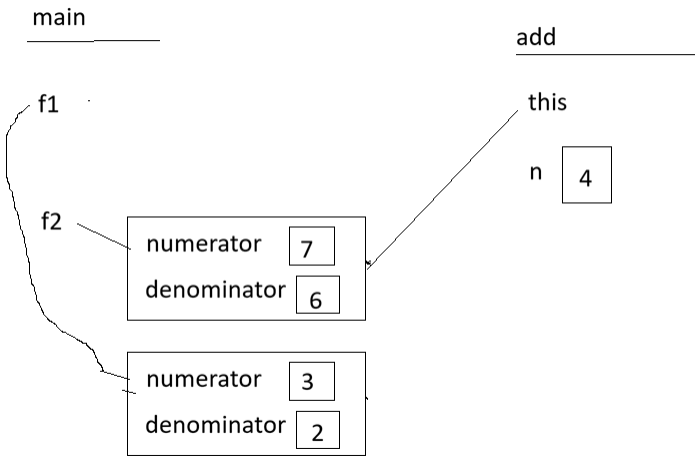
It would be pretty useful to be able to add an integer to a fraction. It would also be annoying if we had to call the method something else because we already used add for adding a fraction to a fraction. Because of method overloading, we don't have to. Here's the add method that takes in an integer:

```
public fraction add(int n) {
    int num = this.numerator+this.denominator*n;
    return new fraction(num, denominator);
}
```

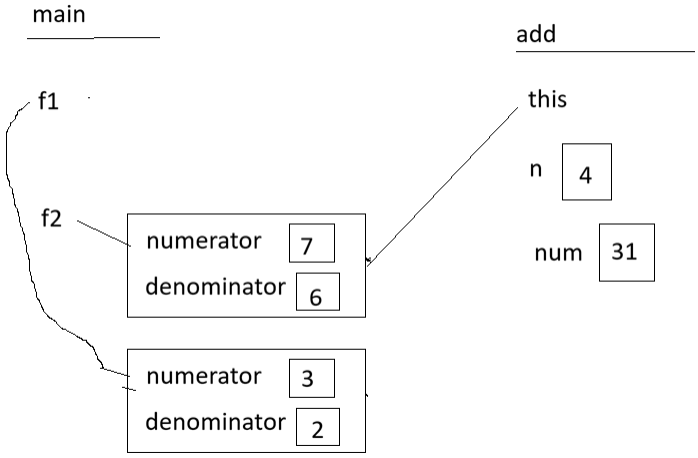
Let's go with our last picture and imagine the following code:

```
fraction f3 = f2.add(4);
```

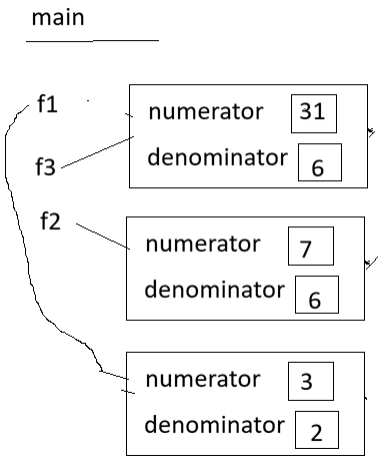
Java looks at the fact that the parameter to add is an integer and not a fraction and realizes that we're calling the add method that takes in an integer. Here's our picture right after the method call:



And here's the picture after the first line of this add (pun intended) is called:



In the second (and last) line of the add method, the constructor is called to create a fraction with the value 31/6. This newly created object is returned to main and the reference f3 from main will point to the object:



this – required use

We'll conclude this lecture with a look at a required use of the keyword this. In this (sorry) example, we'll also see the power of methods building upon each other. First, let's look at the multiply method, which is fairly similar to the add method (just does different math):

```
public fraction multiply(fraction f) {
    int num = this.numerator*f.numerator;
    int den = this.denominator*f.denominator;
    return new fraction(num, den);
}
```

Now that we have this method, we can build on it by writing an exponentiation method (only for positive integer exponents!).

We'll write this the usual way. Regular exponentiation requires creating an accumulator variable and multiplying the base number into it inside of a loop that runs the exponent number of times. Something like this:

```
int ans = 1;
for (int i=0; i<exp; i++)
    ans = ans*base;
```

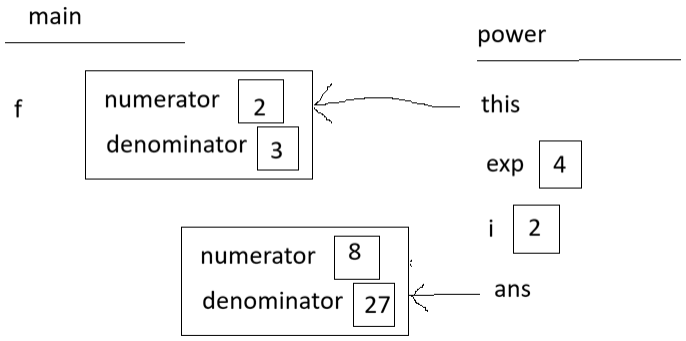
We will literally translate this code above, but for fraction objects. You'll notice that we can't use the * operator (in other languages we can overload operators but not Java). Instead, we'll have to make a multiply call on the accumulator variable. That multiply call will have to take in the base, but keep in mind that the base IS the method the object was called, or this. So this turns out to be a required use of the keyword this:

```
public fraction power(int exp) {

    fraction ans = new fraction(1,1);

    for (int i=0; i<exp; i++)
        ans = ans.multiply(this);

    return ans;
}
```

Hopefully you can see how this is a literal translation of the previous code segment. But because the power method was called on the object we want to exponentiate, and this is the base we need to pass into the multiply method, we HAVE to use the keyword this, as this is the only way to refer to the object that the power method was called on. When this code runs, at any given point in time, ans will be referencing a single object that is some power of this. For example, if we have a fraction object f that is equal to 2/3 and then make the following method call:

```
fraction myans = f.power(4)
```

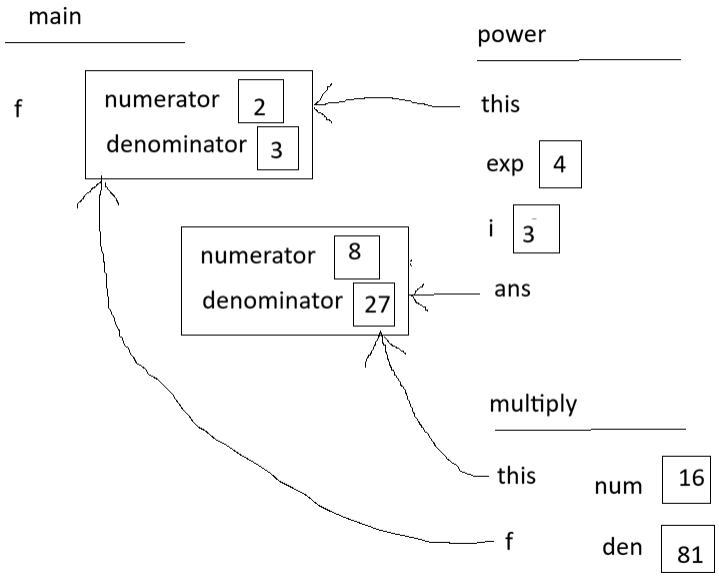
Consider with we complete the loop when i=2, right before the moment in time we increment i to 3. Our picture in this moment would be:



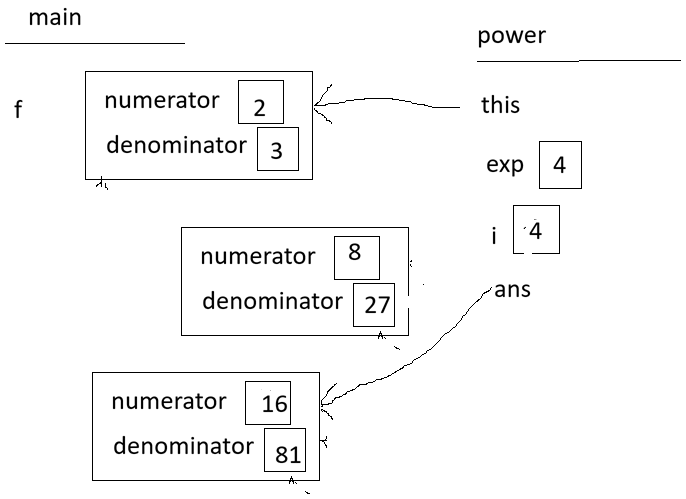
Soon after this picture, i will increment to 3, which is still less than 4 at which point the loop will continue and the line of code

```
ans = ans.multiply(this);
```

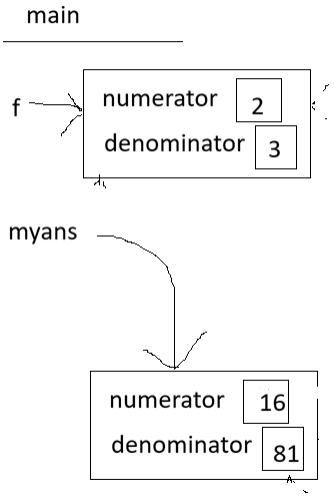
will execute, Here's the picture in the middle of this execution, right after we have assigned both local variables num and den:



Then, after i gets set to 4 and the loop exits, the multiply method returns a new object storing the fraction 16/81 and then ans from power gets reassigned to this. The picture is on the next page:



Soon after the fraction object storing 8/27 will get garbage collected, and then a reference to the object storing 16/81 will be returned to main, leaving this as our final picture after the completion of the power method:



Also included in the fraction class are subtraction, reciprocal, divide, clone and equals. We'll look at these methods later. subtraction, divide and reciprocal are pretty similar to the methods we've already seen. add, multiply and power though exemplify the power (yes pun intended) and flexibility of defining your own class and building upon the methods you've created. These three examples also show all of the necessary mechanics involved in building any class. In lecture we'll look at using the fraction class in one application: FractionGame and one class, FractionTests which just tests the fraction class functionality.