Class Example: Math_Vector Class

Let's look at a useful class that manages a math vector object. It's helpful, but not necessary to understand these notes if you've seen vectors in mathematics or physics class before. Vectors are great tools for solving some geometric problems. A vector represents a direction of movement. This example uses two dimensional vectors, so its instance variables are:

```
private double x_val;
private double y_val;
```

indicating movement in the x and y directions respectively.

We can create a vector from two points in the Cartesian plane: a starting point and ending point. The vector between both points is the difference in their x and y coordinates, Here is the corresponding constructor:

```
public Math_Vector(double xStart, double yStart, double xEnd, double yEnd) {
        x_val = xEnd - xStart;
        y_val = yEnd - yStart;
}
```

One important property of a vector is its magnitude, which is defined as its total length. We can calculate this via the Pythagorean formula (or distance formula depending on how you view it):

```
public double getMagnitude() {
        return Math.sqrt(x_val*x_val+y_val*y_val);
}
```

There are many things that can be done with math vectors, but for this example we'll just explore one key task: calculating the angle between vectors via the dot product. The dot product of two vectors v_1 and v_2 is defined as follows: $v_1 \circ v_2 = |v_1||v_2|cos\theta$, where θ is the angle between the two vectors. (Note: the vertical bars indicate the magnitude of the vector.) It turns out that there is an alternate way to calculate the dot product between two vectors, which just involves multiplying the two corresponding x and y components. Here is the method in the Math_Vector class that returns the dot product of this vector and other:

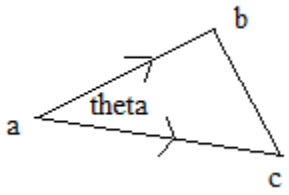
```
public double dotProduct(Math_Vector other) {
        return x_val*other.x_val + y_val*other.y_val;
}
```

Once we have this method, using the formula above, we can solve for the cosine of the angle between two vectors and use that to find the angle itself, via the cosine inverse function. Here is the method that returns the angle (in degrees) between the vectors this and other:

```
public double angleBetween(Math_Vector other) {
        double dot = dotProduct(other);
        double cosAngle = dot/getMagnitude()/other.getMagnitude();
        double angleRadians = Math.acos(cosAngle);
        return Math.toDegrees(angleRadians);
}
```

Now that we've seen most of this class, we can use it to solve a problem:

Consider finding the three angles in a triangle. To find one angle, we can just find the two vectors that form that angle:



If the points are a, b and c of the triangle, we create a vector from point a to point b, and then another from point a to point c, and just call the angleBetween method. We can repeat this two more times to get the other two angles.

Imagine reading in the three points of the triangle into (x1, y1), (x2, y2), and (x3, y3). Then finding one of the angles is as easy as:

```
Math_Vector v1 = new Math_Vector(x1, y1, x2, y2);
Math_Vector v2 = new Math_Vector(x1, y1, x3, y3);
Double angle = v1.angleBetween(v2);
```

Notice how easy this is to do for the person ***using*** the Math_Vector class. They don't really need to understand any of the math, just how to call the constructor and the angleBetween method!

The key to writing a good class is to provide the functionality to the user that is flexible and easy to use so they can call the methods how they would like to, to help solve problems they have. A Math_Vector class fits the bill very well because vectors can be used in many different ways to solve a wide variety of problems.

Class Example: Fraction Class

The fraction class is a larger class that provides detailed functionality with fractions. This class has examples of a couple features of classes we haven't seen before: method overloading and static methods. In addition, it also uses many of the features we've already discussed. It's a good idea to try to draw out pictures to visualize how the fraction class accomplishes its various tasks.

Method Overloading

Method overloading is the practice of creating two methods in the same class with the same name. In order to be allowed to do this, the two methods must have parameter lists that are different. This means that at least one corresponding type for parameters must be different. Obviously, any two methods with a different number of parameters can automatically be distinguished from one another.

The compiler decides which method of multiple methods of the same name gets called by looking at the types of the actual parameters. These types must match the corresponding formal parameters.

Consider the overloading of the add method in the fraction class illustrated below:

```
public fraction add(fraction f) {

  int num = this.num*f.den + this.den*f.num;
  int den = this.den*f.den;

  fraction temp;
  temp = new fraction(num, den);
  return temp;
}

public fraction add(int n) {

  int num = this.num + this.den*f.num;

  fraction temp;
  temp = new fraction(num, this.den);
  return temp;
}
```

We can tell which of these two methods to call based upon whether the user passes us a fraction or an integer as the only parameter. It makes sense to name both of these add because they both add things. The first adds a fraction to this fraction while the other adds an integer to this fraction.

Static Methods

The term static means, "belongs to the class." Thus, a method that is static does NOT pertain to an object. All the methods in the Math class are static because they do NOT operate on, or pertain to, a Math object.

In the Time class, each method does pertain to a Time object. But, there are other situations where a method might be included in a class, even though it doesn't pertain to that object. For example, consider creating a fraction class. It would have to instance variables, one to store the numerator and another to store the denominator. Usually, when dealing with fractions, it's good to keep the fraction in lowest terms. In order to internally do this, we would have to "reduce" fractions after calculations. This reduction involves finding the greatest common divisor between the numerator and denominator. Typically speaking, the task of finding a greatest common divisor between two integers is one that isn't tied to any object. Thus, it would make sense in the fraction class to have a method with the following prototype:

```
public static int gcd(int a, int b);
```