Project 11: Encapsulating Data[1]

This semester, we've written lots of classes that consisted of nothing but static methods. In the real world, classes are only occasionally written like this, such as the Math and Arrays classes from the Java API. In contrast, most classes have instance (i.e. non-static) methods and variables.

A class with instance variables is a bit like a custom data type. Objects of the class have their own copies of each instance variable. This allows the objects to represent particular examples of whatever category of things the class represents.

Consider, for example, the String class, which represents an arbitrary sequence of characters. String objects represent particular character sequences, such as "Java" or "CS 1324" or ":-)". To accomplish this, the String class has a character array instance variable.

Each String object has its own copy of the array that it uses to store its unique text.

In this lab, we will write a class named "Triangle" that represents any three-sided polygon.

Triangle objects represent particular triangles, such as a 3-4-5 right triangle. To accomplish this, the class will have three double instance variables. Each Triangle object will have its own copies of these variables that it uses to store its side lengths.

While writing the Triangle class, we will contend with the fact that some sets of double values cannot be the lengths of a triangle (e.g., -1, 0, 3). To ensure that each Triangle object represents an actual triangle, we will use encapsulation to prevent the user of the class from creating objects in invalid states.

Objectives

The UML diagram on the next page is described in detail in this handout. Each method must be implemented to meet the constraints exactly as described.

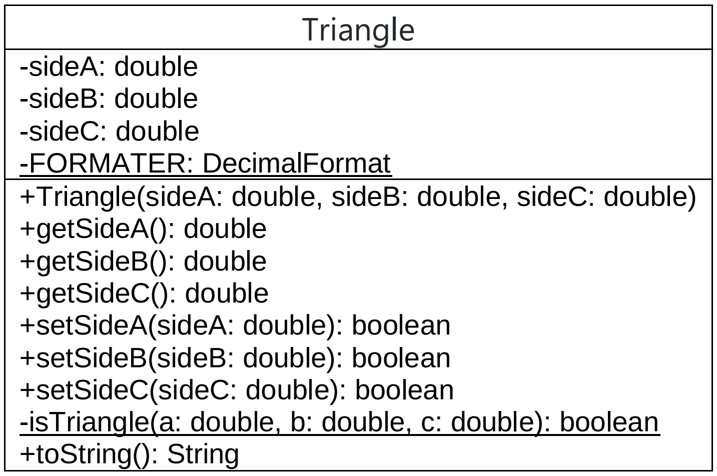
1. Implement instance data for the Triangle class (5 points).
2. Implement the constructor in the Triangle class (5 points).
3. Implement the accessors for the Triangle class (10 points).
4. Implement the mutators for the Triangle class (30 points)
5. Implement class data for the Triangle class (10 points)
6. Implement a class method for the Triangle class (10 points)
7. Test the implementation of your Triangle class in a class called Driver. (20 points)
8. Your code should contain explanatory comments, reasonable white space, sensible indentation, and use meaningful variable names. (10 points)

This project is due on Wednesday, December 9 by 11:59 p.m. If it is submitted by the deadline it will receive 20 points of extra credit. It may be submitted as late as Sunday, December 13 by 11:59 p.m. without penalty.

[1] Thanks to James Dizikes for the design of this project, and preparation of the handout

Triangle Class

The members of the Triangle class are shown in the UML (unified modeling language) diagram below. Notice that there is only one T in the identifier for the FORMATER object.



```
Triangle
-sideA: double
-sideB: double
-sideC: double
-FORMATER: DecimalFormat
+Triangle(sideA: double, sideB: double, sideC: double)
+getSideA(): double
+getSideB(): double
+getSideC(): double
+setSideA(sideA: double): boolean
+setSideB(sideB: double): boolean
+setSideC(sideC: double): boolean
-isTriangle(a: double, b: double, c: double): boolean
+toString(): String
```

Recall that class diagrams consist of three compartments that contain the following information:

Top: class name

Middle: variable names and data types

Bottom: method signatures and return types

The list of variables in the middle compartment shows the name of each variable followed by a colon and then its data type. Similarly, the list of methods in the bottom compartment shows the signature of each method followed by a colon and then its return type.

In addition to this information, special notation indicates which variables are constant, which members are static, and the accessibility of all members:

- Variables with uppercase names are constants and should be declared with the final keyword.
- Underlined members are static.
- Plus and minus symbols represent access modifiers. Plus means that a member is public; minus means that a member is private.

From the diagram, we can see that Triangle has three instance variables: `sideA`, `sideB`, and `sideC`. These variables store the lengths of the sides of a triangle. Because the variables are non-static, each Triangle object has its own copies. This allows each object to represent a different triangle.

Imagine for a moment how the class would be different if the length variables were static. If this were the case, Triangle objects would not have unique copies. All the objects would share the same three length variables. This would mean that the class could only represent a single triangle.

Encapsulating the Side Lengths

The design of the Triangle class follows an object-oriented programming principle known as "encapsulation." This principle dictates that instance variables should be private and public accessor ("getter") and mutator ("setter") methods should be used to access and modify their values.

For the Triangle class, the accessors are `getSideA`, `getSideB`, and `getSideC`, and the mutators are `setSideA`, `setSideB`, and `setSideC`. The accessors return the values of `sideA`, `sideB`, and `sideC`, and the mutators change the values.

At first glance, it appears that accessors and mutators add needless complexity. Why don't we just get rid of these methods and make the variables public?

Later in programming we'll see that encapsulation has multiple benefits, but there's one in particular that matters here: It allows us to constrain the variable values. This is important because the sides of a triangle cannot have arbitrary lengths. For instance, negative side lengths don't make physical sense. However, if the length variables are public, then a user of our class can create Triangle objects with any side lengths they please, including negative lengths. Encapsulation allows us to prevent this.

Side Length Constraints

There are two constraints we need to enforce on the side lengths:

1. Each side must be positive.

2. The sum of any two sides must be greater than the third.

The first constraint applies to any "proper" triangle. As already mentioned, negative sides are unphysical, but the side lengths also can't be zero, since, by definition, a triangle has three sides.

The second constraint is known as the triangle inequality. If you're unfamiliar with it, try this little experiment. Touch the tips of your pointer fingers together to form a straight line. Imagine this line is one side of a triangle. Now try to form the other two sides with your thumbs. Unless your thumbs are unbelievably long, you'll quickly see why this is impossible.

Mathematically, the triangle inequality is actually expressed as the following three inequalities, where a, b, and c are the side lengths: $a < b+c$, $b < a+c$, and $c < a+b$.

If any one of these inequalities is false, then a, b, and c cannot be the side lengths of a triangle. (Note that these inequalities exclude the zero-area case, where the sum of two sides is exactly equal to the third.)

In the next section, we'll see that these constraints must be checked in both the constructor and the setters. We could write identical constraint-checking code in all these methods, but this is bad practice, since duplicate code blocks make programs more difficult to read, debug, and maintain code.

A better way to reuse our constraint-checking code is to write a helper method. This is the purpose of `isTriangle` in the UML diagram. Write this method so that it returns false if either constraint is violated. The method should return true only if *both* constraints are satisfied.

Finally, note that `isTriangle` is a private static method. Its accessibility is set to private because only the constructor and setters need to call it. The user of the class will be unaware that it exists. There are methods like this hidden all over the code that implements the Java API. The method is static because the only data it needs to perform its job are its parameters. It doesn't require direct access to the instance variables.

Enforcing the Constraints

To ensure that `sideA` , `sideB` , and `sideC` are only assigned valid lengths, the constraints listed in the previous section must be enforced anywhere the user of the class can change their values. That is, the constraints must be enforced in any public method that can change the length variables. The Triangle class has four such methods: the constructor (`Triangle`) and the three mutators (`setSideA`, `setSideB`, and `setSideC`).

The constructor initializes the length variables after a Triangle object is created with the new operator. Suppose the constructor is called with an invalid set of arguments like this:

```
Triangle triangle = new Triangle(1, 1, 3); // invalid lengths: 1 + 1 <= 3
```

What can the constructor do to prevent an invalid object from begin created? Nothing, at this point.

The only reasonable option is to assign different values to the length variables. The choice is arbitrary as long as the values are valid, so let's just initialize each side length to 1.

To summarize, here is how the constructor should work:

1. If `isTriangle` returns true, assign the constructor arguments to `sideA` , `sideB` , and `sideC`.

2. If `isTriangle` returns false, assign 1 to each variable instead.

The other methods we need to worry about are the mutators. Fortunately, they're more straightforward to understand than the constructor. This is because a valid Triangle object will already exist when a setter is called. The setter just needs to check whether changing its corresponding variable will put the object in an invalid state. If so, the setter should leave the variable unchanged.

Let's consider an example. Suppose the user runs the following code:

```
Triangle triangle = new Triangle(3, 3, 3);

triangle.setSideC(-1);
```

The first line creates an equilateral Triangle with side lengths equal to 3. This is a valid state for a Triangle object, since the lengths are positive and satisfy the triangle inequality. The second line tries to assign a negative value to `sideC`. If successful, the Triangle would be in an invalid state. To prevent this from happening, the mutator leaves the value as 3.

In addition to checking the constraints and possibly modifying a length variable, each mutator also returns a boolean value. This value indicates whether the Triangle was changed. If the setter updates its length variable, it returns true; otherwise, it returns false.

To summarize, here is how each mutator should work:

1. If `isTriangle` returns true with the new side length, assign the value to the corresponding length variable and return true.

2. If `isTriangle` returns false, leave the variable unchanged and return false.

String Representation

The UML diagram contains one last method that we have not discussed: `toString`.

Implementing this method allows us to define a string representation for our Triangle objects. Once we've done this, printing a Triangle reference variable will output the object's representation, rather than its memory address. This can be very useful for debugging a program.

The string representation of an object should include information that makes it unique from other objects of the same class. For Triangle objects, this information is the side lengths. Let's write `toString` so that it returns a String with the following format:

```
Triangle(<value of sideA>, <value of sideB>, <value of sideC>)
```

Additionally, let's reduce the length of the String by rounding any values with more than 3 digits after the decimal point to the thousandths place.

To clarify the format, consider this example:

```
Triangle triangle = new Triangle(1, 1, 1.0/3.0);

System.out.println(triangle);
```

Running the code should output the following text to the console:

```
Triangle(1, 1, 0.333)
```

Rather than create a new DecimalFormat object every time `toString` is called, assign an object to the constant `FORMATER`. Because this constant is static, this one object can be used by all the Triangle objects.

To create this object:

DecimalFormat FORMATER = new DecimalFormat(“#.###”);

To use it:

double d = 3.27083;

System.out.println(FORMATER.format(d));

The Driver Class

In order to demonstrate that your program is working, you need to write a main program in another class. This class will exercise all of the methods in the program and make sure they are working as described.

The easiest way to do this is to create an ArrayList<Triangle> objects. This will allow you to manipulate the Triangle objects using a for loop.

Here are the tasks that your Driver class should perform.

1. Declare and construct an ArrayList<Triangle>.
2. Put five triangle objects in the ArrayList. Construct these objects with side lengths from (1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6), and (5, 6, 7). Notice that the first set of side lengths is not legal and should be modified by the constructor to (1, 1, 1).
3. Use the toString() method to print your ArrayList of triangles to the console. If you put the reference to the ArrayList in a System.out.println() statement, it will print out the triangles individually in the correct format. The example below will demonstrate the technique.
4. Use the accessor methods to print out the side lengths of your triangles too. Notice the change in formatting below.
5. For all of the triangles, use the mutator to set the side length of side A to 10. Some of these changes will not be legal.
6. Repeat step 3.
7. Repeat step 5, but change the length of side B for all triangles to 10. Again, some of these changes will not be legal.
8. Repeat step 3.
9. Repeat step 5, but change the length of side C for all triangles to 10. As before, some of these changes will not be legal.
10. Repeat step 3.

These is a special kind of for loop that can be used with the ArrayList that works especially nicely in this part of the project. It's informally called a foreach loop (even though that isn't the keyword). An example is shown below.

```
ArrayList<String> list = new ArrayList<String>();
list.add("Raven");
list.add("Jazz");
list.add("Abby");
for(String s: list)  // change this to the Triangle class when you make an ArrayList<Triangle>
{
        System.out.println(s);
}
System.out.println(list); // compare to what happens above!
```

Here is the exact output that should be shown by your program (notice the labels):

```
New objects
[Triangle(1, 1, 1), Triangle(2, 3, 4), Triangle(3, 4, 5),
Triangle(4, 5, 6), Triangle(5, 6, 7)]

Using Accessors
Triangle: 1.0 1.0 1.0
Triangle: 2.0 3.0 4.0
Triangle: 3.0 4.0 5.0
Triangle: 4.0 5.0 6.0
Triangle: 5.0 6.0 7.0
```

```
After Side A changed
[Triangle(1, 1, 1), Triangle(2, 3, 4), Triangle(3, 4, 5),
Triangle(10, 5, 6), Triangle(10, 6, 7)]

After Side B changed
[Triangle(1, 1, 1), Triangle(2, 3, 4), Triangle(3, 4, 5),
Triangle(10, 10, 6), Triangle(10, 10, 7)]

After Side C changed
[Triangle(1, 1, 1), Triangle(2, 3, 4), Triangle(3, 4, 5),
Triangle(10, 10, 10), Triangle(10, 10, 10)]
```

Project Submission

Submit your project on the Zylab for Project 11.