08.08 Virtual Lecture Notes (Part 1)

There are a myriad of decisions that have to be made as programs grow in complexity. One involves whether or not to perform calculations in return statements. Your decision will affect other design issues such as the use of instance variables and the need for additional methods.

Code Anaylsis

First consider the following segments of code for a method to calculate the area of a triangle, and its corresponding invoking statement.

```
public double calcTriArea(int s1, int s2)
{
          return s1 * s2 * .5;
}
...

double triArea = shapes.calcTriArea(side1, side2);
```

The values of variables **`side1`** and **`side2`** are passed to the method parameters **`s1`** and **`s2`**. An arithmetic expression appears in the **`return`** statement, so as soon as the triangle area is calculated, the value is returned and assigned to the **`triArea`** variable in the invoking statement. Even without seeing the rest of the program, some assumptions can be made about this design.

1. Values are passed directly to the method's parameters, which suggest that the **`Shapes`** class uses a default constructor.
2. If a default constructor was used, there probably are not any private instance variables declared for the sides of the triangle, which required the use of the local parameter variables in the calculation.
3. Since the calculation is carried out in the **`return`** statement, the value for the area of the triangle is not retained by the **`shapes`** object.

Now consider an alternative approach shown by the following segments of code:

```
public void calcTriArea()
{
   myArea = mySide1 * mySide2 * .5;
}
...

shapes.calcTriArea();
```

Some very different assumptions can be made based on this design:

1. The method does not take any parameters, so the values for the sides of the triangle were mostly likely passed to the parameter list of a constructor when the object was initialized.
2. The variables in the calculation appear to be private instance variables (**`mySide1`** and **`mySide2`**), which would have been initialized by the constructor.
3. The result of the calculation is assigned to another private instance variable (**`myArea`**), so the area of the triangle is retained by the shapes object.
4. **`The calcTriArea()`** method is invoked by a message statement, not an assignment statement, consequently the return type is **`void`**. So, if nothing is returned, but the value of the triangle's area is needed in the **`main()`** method, how does it get back there?

As you can see, the **`calcTriArea()`** method is overloaded; both versions accomplish the same task, but in very different ways. Program design has many implications that need to be evaluated *before* coding is begun.

Accessors, Mutators, Getters, and Setters

Methods can be categorized based on the tasks they perform and their return type.

The first **`caclTriArea()`** method listed above is an **accessor method**. When invoked, or accessed, it simply performs its duty and returns a value, nothing is modified. The second version of the **`calcTriArea()`** method is a **mutator method**. When a mutator method is invoked, the value of a variable or object is changed. Notice in these two examples, the area of a triangle can be calculated with either an accessor or a mutator method; however, the choice affects the type of constructor used and how returned values are handled.

Mutator methods allow an object to retain values in private instance variables, but they usually (although not always) have a **`void`** return type. To retrieve an object's private instance variable(s), **getter methods** can be used. As the name suggests, a getter method simply returns a value as illustrated in the following code segments.

```
public double getMyArea()
{
          return myArea;
}

...

double triArea = shapes.getMyArea();
```

There is not much to a getter method, it just gets something! Or, as you have seen in other examples, a getter method can be included as an argument in another method.

```
System.out.println("Area = " + shapes.getMyArea());
```

Getter methods have a companion called a **setter method**. Can you predict the purpose of a setter method? A setter method's sole purpose is to set the value of a private instance variable, as indicated by the following example:

```
shapes.setMySide1(15);

...
public void setMySide1(int s1)
{
     mySide1 = s1;
}
```

Setter methods are also very simple. Their job is simply to set something.

In case you are wondering, mutator methods can be implemented to return values in several ways, one of which is shown below:

```
public double calcTriArea()
{
     myArea = mySide1 * mySide2 * .5;
     return myArea;
}

...

shapes.calcTriArea();
```

Before attempting to write the assignment for this lesson, be sure you understand the design implications and consequences of using accessor, mutator, getter, and setter methods. Your instructor will be pleased to further discuss the differences with you.