

COMPUTER PROGRAMMING IN JAVA

COLUMBIA UNIVERSITY HIGH SCHOOL SCIENCE HONORS PROGRAM

Object Oriented Programming, Part 1

Lecture

2008 Mar 29

Object Oriented (OO) Programming: Why?

We've spent our entire time so far programming using methods; why change?

Answer: as our programs get larger and more complex, procedural programming (essentially programming using only methods as the organizational unit) starts to become unwieldy. Also, there are many problems for which it is very natural to formulate the problem in terms of objects, each of which have their own properties and state.

The Old Picture: Variables and Methods

The universe consisted of a bunch of variables, each in their own scope, and methods, which can call each other. These variables had no properties of their own; we could only call operations on them. If we had a particular, well-defined task that we needed to do, we would collect that code into a method, allowing any other method to call this method to help it do its own task.

The New Picture: Classes and Objects

The main thing OO brings to the table is *polymorphism*, but we'll save that topic for next lesson. Today we'll focus more on the basic mechanics of OO programming and a few tangential benefits it brings to the table.

Firstly, in a programming language which supports the OO paradigm, everything is a *class*. A class is like a blueprint or a Platonic ideal (for those of you philosophically inclined): it describes how each *instance* of this class looks like and how it behaves. The Java keyword **new** is how we *instantiate* instances of a class:

```
Triangle aTriangle = new Triangle();  
Square aSquare = new Square();
```

Think about the kinds of variables we've been working with: ints, booleans, and chars:

```
int i = 1;
boolean b = false;
char c = 'm';
```

In the above, “i”, “b”, and “c” are all names of variables *themselves*. However, with objects, like in the example before, “aTriangle” and “aSquare” are *references* to *objects* (a synonym for instance). That is, there is the object itself, and then there is a reference to it (think of it as an arrow pictorially). So if we did:

```
Triangle triangleReference = aTriangle;
```

We're creating a new reference called “triangleReference” which points to “aTriangle”. The compiler knows that “aTriangle” itself points to an object of type Triangle, so therefore “triangleReference” points to the same instance of Triangle.

This is what the Triangle class looks like right now:

```
public class Triangle
{
    public int numSides;
    public int side1;
    public int side2;
    public int side3;
    public String color;
}
```

Methods

A class contains two kinds of things: variables (in OO terminology, *data members*) and methods.

Remembering what we know about scopes, data members are at the class scope; that is, all methods can access their class's data members. Methods we are already familiar with, but note the lack of a “static” in the declaration. We'll talk about exactly what “static” truly means later.

Now, let's add a method to the Triangle class that calculates the perimeter:

```
public int calcPerimeter()
{
    return side1 + side2 + side3;
}
```

Getters and Setters

Getters and setters are methods that access data members. Why would we want this? Getters and setters allow us to control access to our private variables. For example, we can prevent illegal values from getting set, like a negative length for a side of a triangle.

Getter methods always start with “get”, followed by the name of the data member. They are always public, return the same type as the variable, and take no arguments. For example, these are getters for color and side1, respectively:

```
public String getColor()
{
    return color;
}

public int getSide1()
{
    return side1;
}
```

Setter methods always start with “set”, followed by the name of the data member. They are also always public, but return nothing (void), and take 1 argument of the same type as the data member. Here are setters for color and side1:

```
public void setColor(String newColor)
{
    color = newColor;
}
```

```
public void setSide1(int s)
{
    s1 = s;
}
```

So, at this point, our triangle class looks like this:

```
public class Triangle
{
    public int numSides;
    private int side1;
    private int side2;
    private int side3;
    private String color;

    public void setColor(String newColor)
    {
        color = newColor;
    }

    public void setSide1(int s)
    {
        s1 = s;
    }
    public String getColor()
    {
        return color;
    }
    public int getSide1()
    {
        return side1;
    }
}
```

Constructors

Constructors are a special type of method. A constructor has the following properties:

- 1) The name of the method is the same as the name of the class,
- 2) The method has no return type; instead, it returns a reference to a new instance of the class.

For example, here is a constructor for our good ol' Triangle class that allows us to instantiate all the sides:

```
public Triangle(int s1, int s2, int s3)
{
    side1 = s1;
    side2 = s2;
    side3 = s3;
}
```

The Triangle class as we have built it up to this point is:

```
public class Triangle
{
    public int numSides;
    private int side1;
    private int side2;
    private int side3;
    private String color;

    public Triangle(int s1, int s2, int s3)
    {
        side1 = s1;
        side2 = s2;
        side3 = s3;
    }

    public void setColor(String newColor)
    {
        color = newColor;
    }
    public void setSide1(int s)
    {
        s1 = s;
    }
    public String getColor()
    {
        return color;
    }
    public int getSide1()
    {
        return side1;
    }
}
```

Here's another example, this time using pies.

```
public class Pie
{
    protected String typeOfPie = null;

    /**
     * This is an EMPTY constructor
     */
    public Pie()
    {
        this.typeOfPie = "generic";
    }

    /**
     * This is a constructor that allows you to specify
     * the type of pie at initialization time.
     */
    public Pie(String newType)
    {
        this.typeOfPie = newType;
    }

    public String toString()
    {
        return this.typeOfPie;
    }
}

Pie applePie = new Pie("apple");
Pie genericPie = new Pie();

System.out.println(applePie.toString());
System.out.println(genericPie.toString());
```