Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-four)

4.6: Static Members

Remember our original circle class:

```
public class Circle
{
   public double x, y, r;
   public Circle()
   {
      x = y = 0.0;
      r = 1.0;
   }
}
```

Separate copies of the variables *x*, *y*, and *r* are created inside each individual Circle object. This makes good sense, because the coordinates and dimensions of a circle are properties of that individual circle. Several circles could not share the same set of variables and still be distinct circles.

But what if we needed to keep track of the number of circles being created? Well, good style dictates that we keep that datum inside the circle class itself, and arrange for the counting to be done automatically:

```
public class Circle
{
    public double x, y, r;
    // we will store the count in 'n'
    private static int n = 0;

    public Circle()
    {
       x = y = 0.0;
       r = 1.0;
       // increment the count for
       //this new Circle
       ++ n;
    }
}
```

This works because the variable n is marked *static*. In Java, a static variable is one that is associated not with each individual object in a class but with the class as a whole; there is never more than one copy of the variable. Whenever a circle method refers to n, it refers to this single copy of the variable, which all Circles share. n increases each time a new circle is created, just as we'd like to see.

We haven't provided a way to read the value of n, though. We could provide an access method:

```
public class Circle
{
    public double x, y, r;
    private static int n = 0;
```

```
    // ...

    public int count() { return n; }
}
```

This works fine, except that it requires us to have a reference to a Circle object before we can call it. If we don't *have* a Circle, but we want to know how many there are, we could create one, then call the method "count( )" on it; but now "count( )" returns the value of n including the bogus Circle we created just to call "count( )"!

The solution to this mess is to make count() static too:

```
public static int count() { return n; }
```

Static methods, like static variables, are attached to the class as a whole, not to a particular object. For this reason, you can call the static methods of a class even if you don't have an object of that type, by using the name of the class:

int howManyCircles = Circle.count();

You can also call static methods using an actual object variable, like so:

```
Circle c = new Circle();
int howManyCircles = c.count();
```

No matter which way you call a static method, it never can refer to the "this" variable, directly or indirectly. A static method has no current instance. It provides functionality associated with the class as a whole, but not with any particular object.

Note that main( ) is by definition a static function. You might have noticed that, while doing the previous homework assignments, code in main( ) couldn't access functions or variables you may have declared outside of but in the same class unless you declared these other member static as well. Now you should understand why this is so: unless these other members are static, they are attached to a specific class instance, and therefore main( ) would need to create an object of the enclosing class through which to access them. In general, utility methods that don't need an enclosing instance should be static. A good example is the function System.arraycopy( ) discussed in a previous module, which is indeed static.

Note, in addition, that I can now explain the precise meaning of System.out.println( ). out is a final static variable in the java.lang.System class. It is an instance of the java.io.PrintStream class. The various forms of print( ) and println( ) are overloaded nonstatic member functions in java.io.PrintStream.