Lesson 18: *static* Methods and State Variables

You should be aware that *static* methods are sometimes called **class methods**. Similarly, *static* instance fields (*static* state variables) are called **class variables**. The reason for the class designation is that when we access either *static* methods or variables, **we are accessing them at the class level rather than at the object level.** (In this course, we will primarily use the word *static* rather than *class* as the designation of such methods and variables.). This is a profound statement that you will likely only come to appreciate as we move through the material below. …There are two primary reasons for using the key word *static*.

The first reason for using *static*:

We are accustomed to calling a method or accessing a data member (state variable) by first creating an object and then using that object to reach the method or variable. To recall how we do this, consider this class:

```
public class Nerd
{
     public Nerd( )
     { . . . }
     public double methodA(int x)
     { . . . }
     public void methodB(String s)
     { . . . }
     public double abc;
     public int xyz;
}
```

If we want to call *methodB* or access *abc* from outside the *Nerd* class, here is how we have had to do it in the past:

```
Nerd geek = new Nerd( );          //we create a Nerd object called geek
geek.methodB("Some words");       //Here we call methodB, but notice we must use
                                  //the object (geek) we created to do it
geek.abc = 32.38; //Similarly we use the object (geek) to access the state variable
```

Now, we are going to show how to do this **without** having to create an object. First we will do a slight rewrite of the *Nerd* class.

```
public class Nerd
{
     public Nerd( ){ . . . }
     public double methodA(int x){ . . . }
     public static void methodB(String s){ . . . }
     public static double abc;
     public int xyz;
}
```

Accessing without an object:

Notice the key word *static* has been inserted into two places. Both the data member *abc* and *methodB* are *static* which makes the following legal from the "outside world":

```
Nerd.methodB("Some words");
Nerd.abc = 32.38;
```

Notice that we did **not** need to create an object this time. Rather we used the name of the *class*. (That's why they're sometimes called *class* variables and methods.)

Well, this is all rather strange, isn't it? We just aren't accustomed to doing this….But wait! Oh, yes we **have** done this before. Remember our usage of *Math.PI* ? *Math* is a class within Java and *PI* is a data member there. Guess what? It's *static*. That's why we can access it without creating an object.

***static* method from the past:**

Is there an example of where we have used a *static* method in the past? Yes, again. Recall using *Math.sqrt(56.23)*? In fact, all of the methods we have studied in the *Math* class are *static*. We just need to precede the name of the variable or method with the name of the class.

So, there you have it, the first reason for having *static* variables and methods …**the ability to access them without having to create an object**. It should be pointed out that we can still access *static* methods and variables by creating objects…

…*obj.methodB("Some words")*, *obj.abc*, etc. if desired.

Finally, while we are on this topic, we are now able to see why *static* is present in the familiar, *public static void main(String args[ ])* signature. It's because we are accessing the *main* method from the "outside world" (the development environment; BlueJ, JCreator, etc.) **without creating an object** and we now know that the key-word *static* is necessary for us to be able to do that.

The second reason for using *static*:

We will now examine a class with *static* state variables and see what happens when we create various instances of this class. (Notice that's the same as saying we create various objects from the class.)

```
public class Dweeb
{
      . . . some methods and state variables . . .
      public static int x;
}
```

We will now instantiate some objects from this class and manipulate the *static* data member *x*. (The following code is assumed to be in the *main* method of some other class.)

```
Dweeb.x = 79;
System.out.println(Dweeb.x);
```

```
//79…object not necessary to access x

Dweeb twerp1 = new Dweeb( );
//Create objects and still we access the

System.out.println(Dweeb.x);
//79 same, shared value of x

System.out.println(twerp1.x);
//79

twerp1.x = 102;
Dweeb twerp2 = new Dweeb( );
System.out.println(Dweeb.x); //102
System.out.println(twerp2.x); //102
System.out.println(twerp1.x); //102
```

So, we see a second great principle of *static* data members. They are **shared by all instances** (all objects) of the class. In fact, the static variables are still present and available even if no objects are ever instantiated.

Accessing methods and data members from within a *static* method:

If from within a *static* method we try to access another method and/or data member of the same class, then that other method and/or state variable **must also be *static***. This is illustrated in the following code:

```
public class Tester
{
     //Since this method is static, all other
     //methods and state variables
     //in its own class that it accesses must also be
     static.
     public static void main(String[] args)
     {
          . . . some code . . .
          double yz = methodF( );
          double ab = yz + sv;
     }

     . . .more methods . . .

     public static double methodF( )
     { . . . some code. . . }

     public static double sv = 99;
}
```

a. *Static* methods can reference only *static* variables and never the “regular”, non*static*

instance variables.

b. Non-*static* methods can reference either.

Sequence doesn't matter:

Within some class, we might set up a class variable as follows:
public static String s;
The key word sequence *public static* **can be reversed**:
static public String s; //Can also be written this way, but usually the other way.
Even *static* methods can be written with the key-word *static* coming before *public*; however, it's rare to see this in actual practice.

Static constants:

Constants can also be *static* as demonstrated in the following example:

```
public static final double PI = 3.14159;
```

Static imports:

With the advent of Java 5.0 the cumbersome use of *static* methods and variables can now be simpler and more readable. For example, *Math.sqrt(x)* and *System.out.println(x);* can now be written as just *sqrt(x)* and *out.println(x)*; however, the appropriate *static* imports must be made:

```
import static java.lang.Math.*;
import static java.lang.System.out;
```