Lesson 26: More on Classes and Objects

In this lesson we will explore some additional features of classes and objects.

***private* methods and state variables:**

Consider the following class:

```
public class Recipe
{
        public Recipe(String theName )
        { …some code… }

        public void setServings(int x)
        { …some code… }

        public double getRetailCost( )
        {
                …
                int x = 13;
                double tempCost = pricePerCalorie(x) * calories + cost;
                …
        }
        private double pricePerCalorie(int z)
        { …some code… }

        public int calories;
        public int carbs;
        public int fat;
        private double cost;
        public String name;
}
```

We notice that there is a ***private*** method called *pricePerCalorie* and a ***private*** instance field called *cost* (See Nug12-1 for more detail on ***public*** and ***private***).

From within some other class, instantiate an object from this *Recipe* class.

```
Recipe yummyStuff = new Recipe("Watermelon Salad");
```

The following code would be illegal from within the other class:

```
double ff = yummyStuff.cost; //illegal! Cost is private
double dj = yummyStuff. pricePerCalorie(3); //illegal! Method is private
//Both would be legal if private is replaced with public.
```

Notice that from within the *getRetailCost* method that we can **legally** access the ***private*** data member and the ***private*** method. Thus we learn that

***private* things can only be accessed from within the class itself.**

Declaring and instantiating an object:

Normally when we instantiate an object, we do it in one line of code:

```
Circle cir1 = new Circle(3.0);
```

However, it can be done in two lines:

```
Circle cir1;                 //Here, cir1 is merely declared to be of type Circle
cir1 = new Circle(3.0);      //Here, it is finally instantiated.
```

Anonymous objects:

It is possible to **instantiate an object without a name**. Suppose that in the *Ozzy* class (having an object named *osborne*) there is a method that we wish to call that has the following signature:

```
public void melloJello(Circle cirA)
```

Notice that the parameter is of type *Circle* so in our calling code below, we dutifully pass a *Circle* object to the *melloJello* method:

```
osborne.melloJello(new Circle(5) );
```

The code, *new Circle(5)*, instantiates the object; however, in the region of the calling code it doesn't have a name. In the code of the *melloJello* method it **does** have a name, *cirA*. In that code we can do such things as *cirA.area( )* to find the area of the circle, etc.

Setting two objects equal:

Recall the *Circle* class from the previous lesson. Suppose we have instantiated a *Circle* object called *cir1*…..

```
Circle cir1 = new Circle(5.3);          //cir1 has a radius of 5.3
```

We will now demonstrate how to declare a *cir2* object, but not to instantiate it. Then in another line of code, set it equal to *cir1*:

```
Circle cir2; //cir2 has only been declared to be of type Circle
cir2 = cir1; //cir2 and cir1 now refer to the same object. There is only one object.
             //It simply has two references to it.
```

Thus, *cir2.area( )* returns exactly the same as *cir1.area( )*….and *cir1.radius* is exactly the same as *cir2.radius*,…etc.

Determining if two objects are equal:

Look just above at *cir1* and *cir2*. We have said these are equal objects (actually the same object). Since they are equal, the following should print a *true*:

```
System.out.println(cir1 = = cir2); //true
```

However, if we recreate *cir1* and *cir2* in the following way and then compare them, they will **not** be equal.

```
Circle cir1 = new Circle(11);
```

```
Circle cir2 = new Circle(11);
System.out.println(cir1 = = cir2); //false, in spite of the fact they both have a
                                   //radius of 11
```

We see that various objects of the same class must refer to the **same** object in order to be judged equal using = =. (Of course, we could also test with !=.)
Now suppose we change the code as follows:

```
Circle cir1 = new Circle(11);
Circle cir2 = new Circle(11);

System.out.println( cir1.equals(cir2) );
```

What would be printed? This would behave **exactly** as the previous code, printing a *false*. In other words, *(cir1.equals(cir2))* is equivalent to *(cir1 = = cir2)*. In a later lesson on inheritance we would say that the *Circle* class inherits the *equals* method from the cosmic superclass *Object* and simply compares to see if we are referring to the **same** object. There is, however, an exception. If the programmer who created the *Circle* class created an *equals* method for it, then that overrides the inherited method and compares the **contents** of the two objects (likely the radii). In this case, the *println* above would print a *true* since the contents of the two objects are the same (they both have a radius of 11).

With regard to the = = operator, *String* objects behave in **exactly** the same way as other objects; however, they can sometimes **appear** to not follow the rule. Consider the following:

```
String s1 = "Hello";
String s2 = "Hello"; //s1 and s2 are String constants
System.out.println(s1 = = s2); // prints true
```

The *String* constant pool:

Why did this print a *true* when *s1* and *s2* appear to be two separate objects? The reason is that all *String* literals are stored as *String* constants in a separate memory area called the *String constant pool* (as are all *String* literals at compile time). When object *s1* is created, "Hello" is placed in the *String constant pool* with the reference *s1* pointing to it. Then, for efficiency, when the reference (variable) *s2* is created, Java checks the pool to see if the *String* constant being specified for *s2* is already there. Since it is in this case, *s2* also points to "Hello" stored in the *String constant pool*. Physically, *s1* and *s2* are two separate *String* object references, but logically they are pointing to the **same** object in the *String constant pool*. So, in *(s1 = = s2)* from the code above we see that both *s1* and *s2* are referencing the same object, and a *true* is returned.

Now consider *String*s built in the following way and their reaction to the = = operator:

```
String s1 = new String("Felix");
String s2 = new String("Felix"); // s1 and s2 are not String constants
System.out.println(s1 = = s2); // prints false
```

This code behaves exactly as expected since the two *String* objects, s1 and s2, really are two separate objects referenced in an area of memory apart from the *String constant pool* (as dictated by *new*).

While we are on the subject of *String* storage let's see what happens with the following:

```
String s = new String("my string");
```

This actually results in the creation of *two String* objects. The reference *s* points to the newly created *String* object in "regular" memory. The *String* literal "my string" is encountered at compile time and is placed as a *String* constant in the *String constant pool*.

The moral of all this confusion is that if you want to compare the contents of *String*s, use either the *equals* or the *compareTo* method, not the = = operator.

Reassignment of an object:

The name of an object is simply a reference to the actual object and can be easily made to point to a different object.

```
Plant species = new Plant("ragweed");
System.out.println( species.status( ) );
species = new Plant("redwood"); //species is set equal to the new Plant object
species.endangered = false;
```

The reassignment above is exactly analogous to the following in which the integer *x* is assigned a new value.

```
int x = 3;
. . .
x = 5;
```

Default rule of *public*/*private*:

Suppose in a class we have the following method and data member:

```
public double method1( )
{ … some code … }

public int var1;
```

What would these mean if the word *public* was left off of each? By **default** they would be *Package* (see Nug12-1) which for most student applications will behave like *public*.

Initializing state variables at the time of declaration:

Look back at the *Recipe* class on the first page of this lesson. There, you will find the following declaration for the state variable *cost*.

private double cost; //numeric **state variables** are automatically **initialized to 0**.

Notice that *cost* is only declared, not initialized. Typically, initialization is done in the constructor; however, it can be done at the time of declaration as follows:

```
private double cost = 3;
```

Notice that a numeric state variable can be declared, but not initialized as follows:

```
public int idNum;
```

In this case *idNum* is **automatically** initialized to 0.

The rules are different for initialization of a numeric variable in the **body of a method**. Assume that *amount* in the code below is in the body of a method. It is **not** automatically initialized to 0. In fact, trying to use it without initializing will result in a compile error.

```
double amount;
```

A final word about constructors:

When calling a constructor, for example, with

ClassA obj = new ClassA("Yes", 3); ,

the parameters (a *String* and *int* type for this example) must match exactly with one of the constructors in the class. An exception to this is when calling the default constructor

ClassA obj = new ClassA( ); ,

it is permissible to have no constructors in the class. However, if the other constructors are present, the default constructor must be present if called.