STUDENT LESSON

A4 – Object Behavior

STUDENT LESSON

A4 – Object Behavior

INTRODUCTION: It was recognized long ago that programming is best accomplished by working with smaller sections of code that are connected in very specific and formal ways. Programs of any significant size should be broken down into smaller pieces. Classes can be used to create objects that will solve those smaller pieces. We determine what behaviors these objects will perform. These behaviors of the objects are called methods.

The key topics for this lesson are:

A. Writing Methods in Java
B. Parameters and Arguments
C. Returning Values
D. The Signature of a Method
E. Lifetime, Initialization, and Scope of Variables
F. Getters and Setters

VOCABULARY:

ARGUMENT	GETTERS
METHOD	PARAMETER
`return`	SCOPE
SETTERS	SIGNATURE

DISCUSSION:

A. Writing Methods in Java

1. Methods are what an object can actually do, such as in our `DrawingTool` example:

```
myPencil.forward(100);
myPencil.turnLeft();
```

2. Revisiting our example from Student Lesson A2, we can see that we have already been using methods.

```
import gpdraw.*;

public class DrawSquare{

	private DrawingTool myPencil;
	private SketchPad myPaper;

	public DrawSquare(){
		myPaper = new SketchPad(300, 300);
		myPencil= new DrawingTool(myPaper);
	}

	public void draw(){
```

```
                myPencil.forward(100);
                myPencil.turnLeft();
                myPencil.forward(100);
                myPencil.turnLeft();
                myPencil.forward(100);
                myPencil.turnLeft();
                myPencil.forward(100);
        }
}
```

Code Sample 4-1

We wrote our own methods (`DrawSquare, draw`) and we used some methods from the `DrawingTool` class (`forward, turnLeft`).

3. Remember from Lesson A2 that the general syntax of a method is:

```
modifiers return_type method_name ( parameters ){
  method_body
}
```

B. Parameters and Arguments

1. Parameters and arguments are used to pass information to a method. Parameters are used when defining a method, and arguments are used when calling the method.

2. In Code Sample 4-1, we use the `DrawingTool's forward` method to move the `myPencil` object. However, we must tell the `forward` method how far to move, or it would not be a very useful method. We do this by passing an argument to it. In our example, we sent it the `int` value 100. The `turnLeft` method will default to 90 degrees if we don't pass it a value. If we want it to turn a different amount, we can send how far left we want it to turn in degrees.

```
myPencil.turnLeft(60);
```

3. When a method is called with an argument, the parameter receives the value of the argument. If the data type is primitive, we can change the value of the parameter within the method as much as we want and not change the value of the argument passed in the method call. Object variables, however, are references to the object's physical location. When we pass an object's variable name, we get a copy of that reference. Therefore, when we use the passed in reference to access this object within the method, we are in fact working with the same data of the object that was passed as an argument and have the ability to directly change the data inside the object. This can get very messy if the programmer doesn't realize what is going on.

4. When defining a method, the list of parameters can contain multiple parameters. For example:

```
public double doMath(int a, double x){
       ... code ...
       return doubleVal;
}
```

When this method is called, the arguments fed to the `doMath` method must be the correct type and must be in the same order. The first argument must be an int. The second argument can be a `double` or an `int` (since an `int` will automatically be converted by Java).

```
double dbl = doMath(2, 3.5);      //  this is okay
double db2 = doMath(2, 3);        //  this is okay
double db3 = doMath(1.05, 6.37);  //  this will not compile
```

4. Arguments can be either literal values `(2, 3.5)` or variables `(a, x)`.

```
int a = 5;
int x = 10;

double db4 = doMath(5, 10);  // example using literal values
double db5 = doMath(a, x);   // example using variables
//These are equivalent calls
```

C. Returning Values

1. Sometimes we want a method to return a value.

2. In order for a method to return a value, there must be a **`return`** statement somewhere in the body of the method.

3. You must also declare which type of data value will be returned in the method declaration. This can be any primitive data type, a Java class, or any class that you have defined yourself.

```
public int getNumber(){
       int myNumber = 1234;
       return myNumber;
}
```

4. If a method returns no value, the keyword **`void`** should be used. For example:

```
public void printHello(){
       System.out.println("Hello world");
}
```

D. The Signature of a Method

1. In order to call a method legally, you need to know its name, how many parameters it has, the type of each parameter, and the order of those parameters. This information is called the method's *signature*. The signature of the method `doMath` can be expressed as:

```
doMath(int, double)
```

Note that the signature does not include the names of the parameters. If you just want to use the method, you don't need to know what the parameter names are, so the names are not part of the signature.

2. Java allows two different methods in the same class to have the same name, provided that their signatures are different. We say that the name of the method is *overloaded* because it has different meanings. The Java compiler doesn't get the methods mixed up. It can tell which one you want to call by the number and types of the arguments that you provide in the call statement. You have already seen overloading used in the `System.out` object, which is an instance of the `PrintStream` class. This class defines many different methods named `println`. These methods all have different signatures, such as:

```
println(int)        println(double)     println(String)
println(char)       println(boolean)    println()
```

In addition to these, we have been using this concept with the `DrawingTool` class and its `turnLeft` method.

```
turnLeft()          turnLeft(120)
```

3. It is illegal to have two methods in the same class that have the same signature but have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

```
double doMath(int a, double x){}
int doMath(int a, double x){}
```

The Java compiler cannot differentiate return values so it uses the signature to decide which method to call.

E. Lifetime, Initialization, and Scope of Variables

1. Three categories of Java variables have been explained thus far in these lessons.

   - Instance variables
   - Local variables
   - Parameters

2. The lifetime of a variable defines the portion of runtime during which the variable exists. When an object is constructed, all its instance variables are created. As long as any part of the program can access the object, it stays alive. A local variable is created when the program executes the statement that defines it. It stays alive until the block that encloses the variable definition is exited. When a method is called, its parameters are created. They stay alive until the method returns to the caller.

3. The type of the variable determines its initial state. Instance variables are automatically initialized with a default value (0 for numbers, **`false`** for **`boolean`**, **`null`** for objects). Parameters are initialized with copies of the arguments. Local variables are not initialized by default so an initial value must be supplied. The compiler will generate an error if an attempt is made to use a local variable that may not have been initialized.

4. Scope refers to the area of a program in which an identifier is valid and has meaning. Instance variables are usually declared **`private`** and have class scope. Class scope begins at the opening left brace ({) of the class definition and terminates at the closing brace (}). Class scope enables methods to directly access all of its instance variables. The scope of a local variable extends from the point of its definition to the end of the enclosing block. The scope of a parameter is the entire body of its method.

5. An example of the scope of a variable is given in Code Sample 4-2. The class `ScopeTest` is created with three methods:

```
public class ScopeTest{
	private int test = 30;

	public void printLocalTest(){
		int test = 20;
		System.out.println("LocalTest: " + test);
	}

	public void printClassTest(){
		System.out.println("ClassTest: " + test);
	}

	public void printParamTest(int test){
		System.out.println("ParamTest: " + test);
	}
}

//a driver to run the above class
public class ScopeDriver{
	public static void main (String[ ] args){
		int test = 10;

		ScopeTest st = new ScopeTest();
		System.out.println("main: test = " + test);

		st.printLocalTest();
		st.printClassTest();
		st.printParamTest(40);
	}
}
```

Run output:

```
main: test = 10
LocalTest: 20
ClassTest: 30
ParamTest: 40
```

Code Sample 4-2

6. The results show the following about the scope of the variable `test`:

   - Within the scope of `main`, the value of test is 10, the value assigned within the `main` method.
   - Within the scope of `printLocalTest`, the value of test is 20, the value assigned within the `printLocalTest` method

- Within the scope of `printClassTest`, the value of test is 30, the private value assigned within `ScopeTest`, because there is no value given to `test` within the `printClassTest` method
- Within the scope of `printParamTest`, the value of test is 40, the value sent to the `printParamTest` method

F. Getters/Setters

1. When you are first starting to program, some of the most commonly used methods are called Getters and Setters (some like to call them Mutators and Accessors). These methods deal directly with attributes of the object they are associated with.

2. When we are working with an object, sometimes we will need to know certain pieces of information about the object that only the object can tell us reliably. Recall the `DrawSquare` class that we worked with earlier. We might remember the length of a side, but if that length has changed for some reason during the life of the object, we might be in for a surprise if we used that value. Here we would want to store the side length in a private instance variable and provide a Getter method along the lines of `double getSideLength()`. The Getter method's purpose would be to give us current information about that object. We don't want to let other objects access the side length directly, because they might alter that data when we don't want them to. The `getSideLength` method allows other objects to look at the data in our `DrawSquare` object and storing the length in a private variable prevents these objects from directly accessing the length.

3. What happens if we do want to change that side length from outside our class? As it stands right now, that would not be possible. However, we could create a Setter method in the `DrawSquare` class that would do this for us, `void setSideLength(double d)`. This method's basic purpose is to change the value of the sides for us, but it could also do a bit more. For instance, what if someone passed `setSideLength` a value of negative 10? Our square could obviously not exist with a negative side value, so our Setter program should check for validity of the values. Setters can often do calculations for us when necessary, so they are not always simply changing a value and doing nothing else. Often, Setter methods are given a return type of `boolean` which will return true if the value was valid and false if the value was not. This lets clients know if their value was accepted or not. If the client sends an invalid value to a method, it is usually good for them to know that they tried to use the method incorrectly.

SUMMARY/ REVIEW:

As you become more experienced, your programs will grow in size and complexity. When this happens, it becomes more difficult to get your program to do what it is meant to do. Breaking a large task up into smaller tasks by using methods helps make the process easier. Designing methods is complex; the design must integrate a parameter list, the return value, and the goal of the method. Methods should accomplish small tasks within your classes in an easy and organized way. When your programs start to become very large, good

commenting and formatting habits will help you stay organized. Knowing these tools will also help you read code written by other people, which is a great way to learn.

ASSIGNMENT:

Lab Assignment A4.1, *MPG*
Lab Assignment A4.2, *Rectangle*
Worksheet A4.1, *Sphere*