Introduction to Objects

In an Object Oriented Programming language (often referred to simply as OOP) such as JAVA there exists a flexible system of combining data with the code that is responsible for managing the data into one unit. Such an entity is called an OBJECT. The collection of instructions for creating and maintaining the object are referred to as the CLASS of the Object.

You have been working with one special Object for quite some time now by using the `String` class. While the `String` class does have a few special characteristics that differentiate it from standard object classes, the basic functionality it demonstrates does illustrate how an object works.

Once you have created a `String`, you can access methods that return results based upon the value of the `String`. The AP subset includes the `length()`, `substring()`, `equals()`, `indexOf()`, and `compareTo()` methods.

So a `String` represents the combination of data, the contents of the `String`, and methods that return information about (`length()` and `equals()`), or even a subset of (`substring()`) the string.

In the past all the methods that have been written have had the `static` keyword in the method names. The `static` keyword indicates that a method is entirely self-contained and does not use any data from the class that it is a part of. When writing a class we will see that this keyword will usually no longer be used.

Let's look at the basic anatomy of a class:

CLASS DECLARATION: The class declaration primarily contains the name of the class that will be used to "instantiate" an object of that class. By convention, the name of the class should be capitalized.

```
01 public class SimpleObject {
02   private String name;
03   private int age;
04   public SimpleObject() {
05     name="";
06     age=0;
07   }
08   public SimpleObject(String setName) {
09     name=setName;
10     age=0;
11   }
12   public void setName(String setName) {
13     name=setName;
14   }
15   public void setAge(int setAge) {
16     age=setAge;
17   }
18   public String getName() {
19     return name;
20   }
21   public int getAge() {
22     return age;
23   }
24   public String toString() {
25     return name;
26   }
27 }
```

- **`Line 01`**: First comes the class declaration with the keywords **`public`** and **`class`** followed by the name of the class: `SimpleObject`

INSTANCE VARIABLE: An instance variable (sometimes also called non-Static Fields in JAVA, or "data members" in many languages) stores information that the class uses. An instance variable is declared by giving its type and variable name. The variable's first letter should be lowercase.

Making an instance variable **`private`** means that it cannot be "seen" by any code outside this class. This is called "data hiding" or "protecting" and is an important feature of object-oriented design.

- **`Lines 02-03`**: Generally the next items are the "Instance Variables." The `SimpleObject` class has two instance variables, name and age that are declared but not given values.

Note: Instance variables use the keyword qualifier **`private`**. While it **is** possible to make an instance variable **`public`**, it is *highly* discouraged and **all** instance variables are to be private for AP Testing purposes!

CONSTRUCTOR: A constructor is a special method that has no return type and has the method name of the class itself. There may be more than one constructor for a class, but there **must** be as least one.

Constructors **must** have the keyword qualifier `public`.

There should be a "default constructor" which has no parameters.

The job of a constructor is to initialize all the instance variables of the class to their initial, or default values.

- **`Lines 04-07`**: The default constructor for `SimpleObject` sets the instance variable `name` to an empty String and the `age` to `0`.
- **`Lines 08-11`**: This overloaded constructor for `SimpleObject` allows the name to be set during instantiation by passing the `String setName` as a parameter to the method. (See OVERLOADING below!)

SETTER: A setter method (also referred to as a "mutator") allows the program that has instantiated this object to alter the value of one or more of its instance variables.

Using a setter method allows the code within the method to "check" or "validate" the value or change that is being made. In this way, the class maintains control over the possible values that its instance variables can have.

GETTER: A getter method (also referred to as an "accessor") allows the program that has instantiated the object to get information from the object.

A getter method often simply returns the current value of an objects instance variable.

In other cases, the method may perform some operations, combining and evaluating data to return useful information.

OVERRIDDEN INHERITED GETTER: A method that is "inherited" from another class and then defined in the current class is an "overridden" method.

In JAVA, all objects are said to be extended from the `Object` class, and therefore "inherit" certain methods. One of these methods is `toString()` which returns a `String`. This method has the special property that it is used whenever an object is "cast" as a `String` variable.

This most commonly occurs when an object is passed to the `System.out.println()` method.

```
01 public class SimpleObject {
02   private String name;
03   private int age;
04   public SimpleObject() {
05     name="";
06     age=0;
07   }
08   public SimpleObject(String setName) {
09     name=setName;
10     age=0;
11   }
12   public void setName(String setName) {
13     name=setName;
14   }
15   public void setAge(int setAge) {
16     age=setAge;
17   }
18   public String getName() {
19     return name;
20   }
21   public int getAge() {
22     return age;
23   }
24   public String toString() {
25     return name;
26   }
27 }
```

- **`Lines 12-14`**: This setter method allows the program that has instantiated the object to set the value for the instance variable `name`. The parameter `setName` is passed to the method and then assigned to the instance variable `name`.
- **`Lines 15-17`**: Another setter method that allows the value of `age` to be mutated, or set, to the value passed as the parameter `setAge` by the program that has instantiated the object.
- **`Lines 18-20`**: This getter method simply returns the current value of the `name` instance variable.
- **`Lines 21-23`**: Getter method simply returns the current value of `age`.
- **`Lines 24-26`**: This special getter method gets called automatically any time the object is used as a String. The code `System.out.println(thing)` would cause this method to be called because `thing` is expected to be a `String` in that usage of the `println()` method.
- **`Lines 27`**: The final closing brace ends the definition of the class.

It is important to understand that the code in a class by itself doesn't "do" anything. Think of a class definition as a "blueprint" for a building. A blueprint by itself doesn't do anything until the blueprint is used to construct the building depicted by the blueprint.

The process of creating an object from a class definition is called INSTANTIATION and is analogous to "constructing" a building from the blueprint. The analogy further holds in that a single blueprint can be used to construct many copies of a basic building design. Once the building has been created, it is a unique instance of that particular building. Rooms may be occupied (instance variables set or mutated) and information about the building can be accessed using its getter methods. The `SimpleObjectDriver` program below will demonstrate how a `SimpleObject` is first instantiated, and then used:

INSTANTIATION: The process of instantiation generally occurs when a variable assignment is followed by the keyword `new` for any variable that is an object.

This most often occurs right after a variable has been declared so it takes the form: `Type name=new Type();`

Notice that after the keyword `new`, the type is repeated with parenthesis after it. This is because the constructor method is getting called, **not** because the type is getting repeated!

OBJECT METHOD CALL: After an Object has been instantiated its methods can be used to access information or send information in the form of method parameters.

Calling a method is often referred to as "sending a message." If there are no parameters in the method than the method call itself **is** the message because the object is generally being asked to perform some action!

If the method has a return value, it represents a return message.

```
01 public class SimpleObjectDriver {
02   public static void main(String[] args) {
03     SimpleObject thing1=new SimpleObject();
04     thing1.setName("Thing 1");
05     thing1.setAge(21);
06     System.out.println("thing1.getName()="+thing1.getName());
07     System.out.println("thing1.getAge()="+thing1.getAge());
08     System.out.println("I am " + thing1);
09     }
10   }
11 }
```

- **`Line 03`**: To create the variable `thing1`, first the type is given as `SimpleObject` followed by the variable name `thing1` and it is instantiated by the keyword `new` and the constructor method `SimpleObject()`. Because the constructor has no parameters, this instance of a `SimpleObject` will be created using the "default constructor."
- **`Line 04`**: Calls the `setName()` method of `thing1`. This method has one parameter, the new name, "Thing 1" is given to this instance of a `SimpleObject` called `thing1`. What actually occurs? The following lines of code from `SimpleObject` get executed:

  - **`Line 12`**: Simply accepts the parameter variable `setName`, giving it the value "Thing 1."
  - **`Line 13`**: Assigns the value of `setName` to the instance variable `name`.
  - **`Line 14`**: Returns control back to the main program.

```
12   public void setName(String setName) {
13     name=setName;
14   }
```

  - **`Line 18`**: `getName()` has no parameters so this line just serves as the entry point to the method.
  - **`Line 19`**: Returns the value of the instance variable `name`, which currently has the value "Thing 1" and then returns control back to the main program.

```
18   public String getName() {
19     return name;
20   }
```

- **`Line 05`**: Calls the `setAge()` method of `thing1`. This method works just like the `setName()` method except that an integer value is passed as the parameter that gets assigned to the instance variable `age` of `thing1`.
- **`Line 06`**: Outputs the text "`thing1.getName()=`" to the console, but then the code from `thing1` is triggered when the `getName()` method of `thing1` is reached resulting in the console output:

```
thing1.getName()=Thing 1
```

```
01 public class SimpleObjectDriver {
02   public static void main(String[] args) {
03     SimpleObject thing1=new SimpleObject();
04     thing1.setName("Thing 1");
05     thing1.setAge(21);
06     System.out.println("thing1.getName()="+thing1.getName());
07     System.out.println("thing1.getAge()="+thing1.getAge());
08     System.out.println("I am " + thing1);
09     }
10   }
11 }
```

- **Line 07**: This line works like Line 06 except that an integer value is returned from the method. Note that the System.*out*.println() method starts with the String "thing1.getAge()=" so the integer value returned by the getAge() method is converted into the String "21" and then concatenated with the text "thing1.getAge()=" resulting in the console output:

```
thing1.getAge()=21
```

- **Line 08**: This line is very similar to Line 06 but with one **very** significant difference. The line begins by executing a System.*out*.println() method. The parameter that gets passed to this method must first be evaluated. First comes the String "I am " followed by thing1. Notice that there is no method given after thing1 so thing1 must be "cast" into a String so that it can be concatenated with the String "I am ". Recall that the special method toString() is called when an Object is cast into a String. So the thing1 object is implicitly cast into a String, triggering its toString() method at **Line 24**. The return value "Thing 1" is then concatenated with the String "I am " resulting in the console output:

```
I am Thing 1
```

- **Line 24**: toString() has no parameters so this line just serves as the entry point to the method.
- **Line 25**: Returns the value of the instance variable name, which currently has the value "Thing 1" and then returns control back to the main program.

```
24   public String toString() {
25     return name;
26   }
```

OVERLOADING EXAMPLES: The pattern of variable types used when a method is called will determine which overloaded method actually gets triggered.

The code below illustrates a single method name with five different combinations of parameters. Not that the variables names are inconsequential, only the **order** of variable types matters!

Each of the methods below returns a different String. Each of the method calls starting on line **Line 08** will result in a different version of doIt() getting called:

- **Line 08**: The call to doIt() with no parameters executes the method on **Line 02** resulting "A" getting output to the console.
- **Line 09**: This call to doIt() has one integer parameter so the doIt(**int** x) method on **Line 03** gets called so that "B" is output to the console.
- **Line 10**: This call to doIt() includes the integer parameter 5 followed by the double parameter 5.0 so the doIt(**int** x, **double** y) method on **Line 04** gets called and "C" is output to the console.
- **Line 11**: The call to doIt() on this line requires that the expression 2/.5 be evaluated first, and since .5 is a double, the whole expression will be a double so the first parameter is a double. The second parameter is the integer 1 so the doIt(**double** y, **int** x) method on **Line 05** gets called and "D" is output to the console.
- **Line 12**: This call to doIt() has the double value 0.0 for the first parameter and the second parameter is the integer value 5 cast to be a double so the doIt(**double** x, **double** y) method on **Line 06** gets called and "E" is output to the console.

```
01 public class Overloading {
02   public static String doIt() {return "A";}
03   public static String doIt(int x) {return "B";}
04   public static String doIt(int x, double y) {return "C";}
05   public static String doIt(double y, int x) {return "D";}
06   public static String doIt(double x, double y) {return "E";}
07   public static void main(String[] args) {
08     System.out.println(doIt());
09     System.out.println(doIt(5));
10     System.out.println(doIt(5,5.0));
11     System.out.println(doIt(2/.5,1));
12     System.out.println(doIt(0.0,(double)5));
13   }
14 }
```