Java Town Objects – Background Material

AP Computer Science with Data Structures

Background: Assignment Statements

Open up JavaTown, and try *executing* the following commands, **one at a time**. Try to predict what each will do, and then verify that JavaTown's behavior matches your prediction. Make sure you understand what each operator does before moving on. Be sure to ask someone if you have any questions.

```
x = 1;
x = 2;
x = 3;
x = 4; //type me too!
x = 3; //x = 4;
x = x + 7;
y = 9 / 4;
y = 2 * (3 + 4);
y = 8 % 3;   (the remainder when 8 is divided by 3)
x = 3 > 1;
x = 3 <= 1;
iLoveComputers = true;
iLOVEComputers = false;
x = !(3 > 2);
x = 1 == 2;
x = 1 != 2;
x = 1==1 && 1==2;
x = 3/0 == 3;
x = 1==1 && 3/0==3;
x = 1==1 || 3/0==3;
x = false || 3/0==3;
x = false && 3/0==3;
x = nutella;
x = null;
x = "Nutella";
```

Let's take a closer look at the following code.

```
count = 3;
count = count + 1;
```

These are called assignment statements, and it should be clear that JavaTown's '=' has little in common with the one you see in math class. The right side of an `=` can be any hairy expression, but the left side *must* be a variable name. To execute an assignment statement, JavaTown first *evaluates* the expression on the right side and then *assigns* that value to the variable name on the left side. The first statement above simply requires evaluating the trivial expression `3` and assigning that value to `count`. The second statement looks a bit trickier, but is executed exactly the same way. JavaTown first evaluates the expression `count + 1` (and since `count` was just assigned the value 3, `count + 1` will evaluate to 4), and then assigns this value to `count`. Together, these two statements have the same effect as the following single statement.

```
count = 4;
```

Background: Conditionals

Try *executing* the following commands, **one at a time**. Make sure you understand what each does before moving on. Be sure to ask someone if you have any questions.

```
if (5 > 3) result = "yay";

if (false) result = "boo";

if ("false") result = "huh?";
```
(No, don't actually report this behavior to the JavaTown development team.)

```
if (5 == 3)
     status = "bad";
else
     status = "good";

if (1 + 2 == 3)
{
     result = "bad";
     result = "i mean good";
}
```

Background: Objects

A much more interesting thing we can do in JavaTown is to define a *class*. We'll start by making a class of *objects* called `Noob`. Execute the following code.

```
public class Noob {}
```

Executing the above does not actually make any `Noob`s. It simply tells JavaTown what a `Noob` is, and having done so, we can now *construct* an *instance* of the `Noob` class as follows. Execute the following code.

```
n1 = new Noob();
```

This is simply another assignment statement. This one evaluates the expression `new Noob()`, which creates a new object of type `Noob`. The value of this expression is the address of the `Noob`'s house, and this is what's assigned to the variable name `n1`.

Now see if you can predict what the following statement will do. Does it create a second `Noob`? Go ahead and execute it.

```
n2 = n1;
```

Here, JavaTown evaluates the expression `n1` to be the address of a house, and assigns that address to `n2`. We say that `n1` and `n2` both *point to* the same object. Thus, an assignment statement alone is not enough to create a new object—that would require the use of the keyword `new`, as we saw earlier. Hence the old saying:

> "No `new`s, no new `Noob`s." --Socrates

Go ahead and execute the following. What is the value of `same`?

```
same = (n1 == n2);
```

What will the following statement do now? This time use the "step" button (repeatedly) to observe JavaTown's behavior one step at a time. (Feel free to press "step" instead of "execute" any time.) What values are stored in `n1` and `n2` now?

```
n2 = new Noob();
```

Now execute the following statement again.

```
same = (n1 == n2);
```

If you haven't already figured it out, the `==` operator tells you whether two variables *contain the same value*. In this case, we're testing if the two variables *contain the same memory address*. In other words, we're testing if the two variables *point to the same object*. Likewise, we can test whether `n1 != n2` to determine if `n1` and `n2` point to different objects. (Since, it is currently *not* the case that `n1` and `n2` point to different objects, the value of this expression would be *false*.)

Execute the following statement, which indicates that `n2` will now point to nothing at all. This means that one of the `Noob` objects is no longer pointed to by any variable. As a result, the stranded `Noob` is automatically *garbage-collected*.

```
n2 = null;
```

Background: Methods

Redefine the `Noob` class as follows.

```
public class Noob
{
     public speak() { return "lol"; }
}
```

Here, we have indicated that the `speak` *message* can now be sent to instances of the `Noob` class. Furthermore, when a `Noob` receives this message, it will execute the code in the corrresponding `speak` *method*. Here, the `speak` method returns the string (text value) "`lol`".

If you're an experienced Java user, you may be surprised that we didn't need to declare the return type of the `speak` method. Although values in JavaTown do have types, these types are never explicitly declared, which makes the syntax a bit simpler. If you do declare a variable or method type, JavaTown will simply ignore it. Thus, JavaTown considers each of the following to be equivalent:

```
f = 31;
int f = 31;
boolean f = 31;
mr f = 31;
```

Now use the *step* button to send the `speak` message to your `Noob` as follows. What value do you see now in the variable `speech`?

```
speech = n1.speak();
```

What happens if you ask `n2` to `speak`?

Let's redefine `Noob` to include a second method.

```
public class Noob
{
     public speak() { return "lol"; }

     public askFriendToSpeak(friend)
     {
          spoken = friend.speak();
          return "friend said " + spoken;
     }
}
```

The `askFriendToSpeak` message requires that you pass it an *argument*. The argument value that you pass to `askFriendToSpeak` becomes associated with the *formal parameter* name `friend`. (We can define define a method that requires more than one argument by listing multiple formal parameter names, separated by commas.) The variables `friend` and `spoken` are *local variables*. They exist only temporarily, and are forgotten as soon as JavaTown finishes executing the method. To test our new method, let's make a second `Noob` again.

```
n2 = new Noob();
```

Now try *calling* the `askFriendToSpeak` method as follows, and stepping slowly through the resulting computation. What value is *passed* to the `askFriendToSpeak` method? Where do the local variables appear on the screen?

```
speech = n1.askFriendToSpeak(n2);
```

The name `n2` is *not* passed to the method. Instead, JavaTown first finds the value of `n2`, to be some memory address. It's the *address* that gets passed to the method. The passed values are then associated with the formal parameter names (`friend`) inside the thought-bubble you saw (along with any other local variables, like `spoken`).

Background: Instance Variables

Now let's define a new class called `Lamp`. A `Lamp` is different from a `Noob`, because a `Lamp` must *remember* whether it is on or off. Each `Lamp` will store this information in an *instance variable* (although we will also sometimes refer to it as a *field*, *member*, or *attribute*).

```
public class Lamp
{
     private lightOn;

     public turnLightOn() { lightOn = true; }
     public turnLightOff() { lightOn = false; }
}
```

Try making a couple `Lamp` objects and calling these methods. Notice that the `lightOn` instance variable is written to the chalkboard and remains there after the `turnLightOn` and `turnLightOff` methods are executed.

Background: Constructors

In addition to instance variables and methods, a class also has *constructors*. A constructor lets us tell JavaTown to run a certain code segment every time an instance of the class is constructed. A constructor for the `Lamp` class might appear as follows.

```
public Lamp( … ) { … }
```

When you evaluate `new Lamp()`, we are actually calling a constructor in `Lamp` that takes no argument and executes no code. You get this constructor for free, without having to actually declare it in your class definition. If we wanted to write it out, it would look like this:

```
public class Lamp
{
     private lightOn;

     public Lamp() {}

     public turnLightOn() { lightOn = true; }
     public turnLightOff() { lightOn = false; }
}
```

But why would you want to use a constructor? Well, notice that the first time you create a `Lamp`, its `lightOn` variable initially contains the value `null`. This is the case for all instance variables in JavaTown. (The situation will be a little more complicated in Java.) But it would make much more sense to initialize `lightOn` to be `false`, meaning that the lamp is initially off. In other words, every time a new `Lamp` is created, we want JavaTown to run the following code:

```
lightOn = false;
```

We can achieve this behavior by defining our constructor as follows.

```
public class Lamp
{
     private lightOn;

     public Lamp() { lightOn = false; }

     public turnLightOn() { lightOn = true; }
     public turnLightOff() { lightOn = false; }
}
```

Go ahead and test that, when you construct a *new* `Lamp`, your `isLightOn` instance variable is now correctly initialized to false.

Like methods, constructors can also be declared to use argument values. For example, we might modify `Lamp`'s constructor to read as follows.

```
public Lamp(init) { lightOn = init; }
```

Now, when we construct a Lamp, we can write `new Lamp(true)`, or `new Lamp(false)`, or `new Lamp(x > 2)`, etc. But we can't write `new Lamp()` anymore. We lose the default constructor as soon as we explicitly declare a constructor of our own. If we want, though, we may choose to declare multiple constructors.

Background: JavaTown Documentation Requirements

JavaTown does not support block comments. That means you cannot use /* and */ to delimit comments and you cannot use the javadoc format /**. **YOU ARE NOT RELIEVED OF THE RESPONSIBILITY OF DOCUMENTING YOUR PROGRAM. YOU ARE REQUIRED TO DOCUMENT.** The symbol pair // denotes the beginning of a single line comment.