CIS 201 – Computer Science I
Laboratory Assignment 6

Introduction

In this lab, you will experiment with creating your own Java classes.

As we have described in lecture and your text, a Java `class` definition is a blueprint for creating objects. When an object is created from a `class` definition, that object has properties described by the values of its field variables and provides the services advertised by its methods. We think of the properties as attributes (*e.g.*, a FANG `Sprite` has a `Location2D` and a `Color`) and the methods as verbs (*e.g.*, you can `translate` and `rotate` a FANG `Sprite`).

The *constructor* of a class is a special method that describes how to create an *instance* (*i.e.*, an object) of that class. A constructor, called when you use the `new` keyword, is used only once – when the object is created, whereas an object's other methods can be used with the object during its entire lifetime.

You have been using constructors for FANG objects since the very first day of lab. The following line, for example, constructs a new `OvalSprite` and assigns a reference to it to the locally declared variable `round`. This line is an example of a *declaration* (the stuff before the `=` sign) and a *definition* (the stuff beginning with `round = new ...`).

```
OvalSprite round = new OvalSprite(0.10, 0.10);
^          ^     ^ ^   ^          ^^^^^^^^^^
|          |     | |   |                    \_ parameters to the
|          |     | |   |                       constructor
|          |     | |   \_ Name of constructor to call
|          |     | |      (same as name of class to construct)
|          |     | \_ keyword "new", so Java calls a constructor
|          |     \_ assignment operator (puts the reference to
|          |         the new object into the box associated with
|          |         the variable named on left of assignment)
|          \_ name given to the variable that will hold a reference
|             to the object (an OvalSprite)
\_ type of the variable being declared
```

To use a Java class defined by FANG (or anyone else), you create one or more instances of the class and then use the instances (the objects) to carry out useful work. Remember that an object can exist only when it has been created using the `new` operator.

When you create your own class, you have control over the properties (field variables) that objects of that class should have and the services (methods) that objects of that class should provide.

Many times, you create your own class in a way that `extends` another class – as in our `NewtonsApple` class that `extends` the `Game` class. That way we can use the properties and methods that the other class describes, while adding our own properties and methods that makes our class special. As another example, we could have a `Building` class whose properties include number of rooms, number of stories, and so forth. A `House` class could extend the `Building` class by adding additional properties such as the number of bedrooms.

How to make a class

To create a Java class, we first create a Java source file (of the form `<classname>.java`). After we have created this source file, we use the Java compiler to create a Java class file (of the form `<classname>.class`). The class file consists of a binary representation of the class, including encoded descriptions of the field variables and methods. You would not generally want to look at a class file using an editor, but if you did, you would see the names of your field variables and methods.

Remember that the Java compiler requires that the name of the file must match the name of the class: `public class X` must be defined in `X.java` (and will compile to the class file `X.class`). Convention (and grading criteria in this class) require that the names of classes (and thus the files in which they are defined) begin with capital letters and appear in camel-case.

Start coding by creating a `Lab06` directory in which you and your partner will save your work. Note that this time there will be *multiple* `.java` files in this directory.

First, create a Java source file that defines the class `MyName` by creating (using emacs or your favorite editor) a file named `MyName.java` that has the following contents:

```
public class MyName {

  private String myName;

  public String getName() {
    return myName;
  }

  public String getXYZZY() {
    return "XYZZY";
  }
}
```

Compile this program and verify that your working directory now has a class file named `MyName.class`.

Checkpoint 1

Run this program from within `emacs` or from a shell window.
What happens?
Show us your results at this point.

Using MyName in a game

Now start editing a new file named `MyNameGame.java`, with the following contents. *If you are using emacs, use the existing emacs process to edit this file instead of launching a separate emacs process.*

```
import fang.core.Game;
import fang.sprites.StringSprite;

public class MyNameGame extends Game {

  private MyName mn; // declare a reference to a MyName object
  private StringSprite mnSprite;

  public void setup() {
    mn = new MyName();
    mnSprite = new StringSprite();
    mnSprite.scale(0.05);
    mnSprite.leftJustify();
    mnSprite.setLocation(0.1, 0.5);
    mnSprite.setColor(getColor("white"));
    mnSprite.setText(mn.getName());

    addSprite(mnSprite);
  }
}
```

Save and compile this program. You should see no errors (assuming you've typed in everything correctly). You should also notice that you have a class file named `MyNameGame.class` in your working directory.

Checkpoint 2

Run the `MyNameGame` program from within `emacs`.
What happens?
Show us your results.

Null Pointer Exceptions

As you saw, the Java runtime system encountered a problem when you tried to run your program. Notice that your program *compiled* correctly but that it failed to *run* correctly, and that at the top of the (long) list of problems, the Java runtime system said "Null Pointer Exception". This means that it tried to refer to an object, but that there was no object to refer to.

In this case, the offending part was the value of `mn.getName()`. The `getName` method in your `MyName` class returns the value of the `myName` variable, but this variable was only declared and never defined. Java automatically sets references to `null` when they are not defined.

You can fix this in `MyNameGame.java` (temporarily) by replacing the `getName` call with `getXYZZY`, which *does* return a value. Compile and run your program after having made this change, and verify that it displays `XYZZY` on the screen.

Now modify this program by commenting out the line that says

```
mn = new MyName();
```

Compile and run your program, and observe that you get another Null Pointer Exception error. This time, the error is that `mn` starts off with a value of `null`, and when you finally refer to it in the `setText` line, nothing is there.

Checkpoint 2

Show us your work at this point.

Setting your name

Be sure to un-comment the `new MyName()` call before you proceed.

There are two ways to give a value to your `myName` field in the `MyName` class. The first is to create `setName` method, as follows:

```
public void setName(String n) {
  myName = n;
}
```

Notice that this method has a `void` return type, which means that its only purpose in life is to have a *side-effect*; it does not determine or return a value. In this case, the side-effect is to modify the `myName` field.

Add this method to your `MyName.java` file, either before or after the `getName` method (you choose), and compile `MyName.java`. You should not see any errors.

Now return to editing your `MyNameGame.java` file. In `setup`, just after you create the `MyName` object, add the following line:

```
mn.setName("<your name goes here>");
```

Of course, replace `"<your name goes here>"` with whatever string represents *your* name. Now, change the method call `getXYZZY` back to `getName`.

Compile and run your `MyNameGame.java` program.

Checkpoint 3

Show us your results at this point.

Constructing your name

Methods like `setName` and `getName` are called (appropriately!) *setters* and *getters*, respectively.

They provide services to those using object of the class which allow the user to *get* or *set* the value of a field variable, which is (in our case) has `private` visibility and can't be otherwise seen from outside the object.

Data hiding or the use of `private` data fields is an example of using abstraction to overcome the complexity of a computer program. By making all changes to `myName` through methods defined in `MyName`, we can make sure that if we trust the `MyName` class, no other class can violate the integrity of the fields in the class.

Instead of using a setter to set the `myName` field, we can set this value when we initiall construct the object using the `new` operator.

To do this, we will need to add a new “method” to our `MyName` class, called a *constructor*. While a constructor looks like other methods, it has no return type. Also a constructor *always* uses the name of the class as its name. Finally, a constructor will almost always have a `public` visibility.

So in our case, we can define a constructor in our `MyName.java` file as follows:

```
public MyName(String n) {
  myName = n;
}
```

Put this code just after the definition of the `myName` field at the beginning of your program.

Compile your `MyName.java` program and be sure it doesn’t have any errors.

Now change your “client” program `MyNameGame.java` by replacing the lines that say

```
mn = new MyName();
mn.setName("<your name goes here>");
```

with the *single* line that says

```
mn = new MyName("<your name goes here>");
```

(Again, replace the `"<your name goes here>"` with a string that has your name in it.)

Compile you program and run it.

Checkpoint 4

Show us your results at this point.

Religion

Should you have both a constructor and a setter in your `MyName` class?

If you *never* expect to change the value of `myName` after creating a `MyName` object, then setting the value in the constructor would work best. (If you keep the constructor but remove the setter, no client program can ever change the value of `myName`.)

If you *ever* expected to change the value, then you should keep the setter, and (I would suggest) remove the constructor.

My reason: its best to have just one way to do something, so that you'll never have surprises. But this is a religious issue, not a Java standard. Your mileage (and that of your instructor) may vary.

Silly extension

Now create a new Java file named `StrSpr.java`, with the following code:

```
import fang.sprites.StringSprite;

public class StrSpr extends StringSprite {
}
```

Compile this program. (Don't try to run it...)

You have defined a new class named `StrSpr` whose objects have inherited *all* of the properties and methods of `StringSprite`, and has changed/added nothing.

Now return to your `MyNameGame.java` program and make the following changes:

1. comment out the import line that refers to `fang.sprites.StringSprite`
2. change the type of `mnSprite` from `StringSprite` to `StrSpr`
3. change the line where the `StringSprite` constructor is called to refer instead to the `StrSpr` constructor.

Compile and run your program.

Checkpoint 5

Describe to us what happens.
In particular, explain what the `StrSpr` class accomplished that is different from the `StringSprite` class.

Assignment 3

Begin work on Assignment 3.