CIS 201 Computer Science I
Fall 2009 Lab 05

September 27, 2009

- Build a solution with multiple classes.
- Write a class with a constructor.
- Create new instances of your class and use them in a game.

Background

A Java `class` definition is a blueprint for instantiating objects. When a new object is constructed, the properties of the object are the values of the fields declared in the `class` and its available services are just the `public` methods defined by the `class`. For example, a `Sprite` has a `Location2D` field called `location` which keeps its (x, y) location on the screen and a `Sprite` has methods to `rotate`, `translate`, and `scale` (among others).

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 class. The following line, for example, constructs a new `OvalSprite` and assigns a reference to it to the locally declared variable `round`:

```
OvalSprite round = new OvalSprite(0.10, 0.10);
^          ^     ^ ^   ^          ^-parameters for 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 box that can hold a reference to
|             an OvalSprite
 - type of the variable being declared
```

To use a `class` whether defined by FANG, the Java standard library, or even yourself, you create one or more instances of the object and use the methods of the instances to carry out useful computation.

When you create your own class, you have control over the properties that objects of that class should have and the services that objects of that class 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.

How to make a class

To create a Java class, we first create a Java source file (of the form `<filename>.java`), and then we use the Java compiler to create a Java class file (of the form `<filename>.class`).

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.

Checkpoint 1 Multiple `.java` files.

Start coding by creating a `Lab05` directory in which you and your partner will save your work. Note that this time there will be *multiple* `.java` files in this directory.

Now create your first non-`Game` class, `MyName`.

Start editing a file named `MyName.java` and give it the following contents:

```
public class MyName
  extends Object {

  private String myName;

  public String getName() {
    return myName;
  }

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

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

Run the program, `MyName` from within `emacs`. What happens? Any idea why?

Show your work on Checkpoint 1 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 2 Using `MyName` in a `Game`.

Now start editing a new file named `MyNameGame.java`, with the following contents:

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

public class MyNameGame extends Game {

  private MyName mn; // refers to a MyName object
  private StringSprite mnSprite;

  public void setup() {
    mn = new MyName();
    mnSprite = new StringSprite();
    mnSprite.scale(0.05);
    mnSprite.leftJustify();
    mnSprite.setLocation(0.0, 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.

Run `MyNameGame`. What happens? Why?

Show your work on Checkpoint 2 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 3 "codeNullPointerException

The Java runtime system encountered a problem when you ran your program. `MyNameGame` *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()`, which refers to the value of the field variable `myName`, which was never given a value.

You can fix this in `MyNameGame.java` (temporarily) by replacing the call to `getName` call with `getXYZZY`, which *does* return a value. Compile and run this program, 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 `NullPointerException`. 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.

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

Show your work on Checkpoint 3 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 4 Setting your name.

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. You are typing a *literal string*: do you need quotes around what you type? Now, change the method call in `MyNameGame` from `getXYZZY` back to `getName`.

Compile and run your `MyNameGame` program.

Show your work on Checkpoint 4 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 5 Constructing your name

Methods like `setName` and `getName` are called (appropriately!) *getters* and *setters*, 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" `MyNameGame` 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>"); // <- use your name there
```

(`MyNameGame` is referred to as a *client* of `MyName` because it *uses* the class definition. By this definition, `NewtonsApple` was a client of much of FANG and `MyNameGame` is also a client of `fang2.core.Game`.)

Compile and run `MyNameGame`.

Show your work on Checkpoint 5 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

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 works best: with a constructor to set the value but no setter, no client program can change the value of the `myName` field.

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.

Checkpoint 6 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 the `StrSpr` class (to check for typos).

`StrSpr extends StringSprite`; this means that `StrSpr` offers *exactly the same* services as `StringSprite` because every `public` method of `StringSprite` is a `public` method of `StrSpr`. The new class does not provide any new functionality.

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

1. comment out the import line that refers to `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.

Describe to us what happens.

What has the `StrSpr` class accomplished that is different from the `StringSprite` class?

Show your work on Checkpoint 6 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Log off of the lab computer you are using before leaving they lab. Anyone entering the lab has unlimited access to your files if you remain logged on. **DO NOT** turn off lab computers! They are a shared resource and there might be someone else logged in to "your" machine.

Clean up your work area, push in the chair, and have a good week.