CIS 201 Computer Science I
Fall 2009 Lab 04

September 21, 2009

In this lab, we will experiment with:

- local variables and field variables.
- primitive data types and object references.
- private and public fields and methods.

Activity 1 All methods in the class can access fields

A *field* is a variable that is declared inside the class but outside of any method defined in the class. You will normally put field declarations near the top of your Java source file, just after the open brace following your `class` declaration.

Fields can have `public` and `private` visibility. You will normally want to give fields `private` visibility in order to follow the software engineering principle of *information hiding*, a topic that will make more sense to you as you learn more about computer science.

Set up. Create a directory named `Lab04` in your CIS 201 directory. In the `Lab04` directory, create a Java source file named `Fields.java` that defines a `class` named `Fields` and that extends the FANG `Game` class. As usual, you will need to `import` the appropriate FANG library at the top of your file. The body of the `Fields` class should be empty: there should be no code between the opening and closing curley braces.

Your Java source file should be complete enough at this point to compile and run – so **compile and run your program** to be sure you don't have any errors. Close out the game window before you proceed.

One integer variable. Inside your `Fields` class, put the following code:

```
private int x;

public void setup() {
    x = 3;
    System.out.println("x=" + x);
}
```

On a blank sheet of paper, draw a diagram representing the memory cell that contains the integer variable `x`. Use a rectangle for the memory cell, and label the rectangle with the name `x`.

A diagram such as this can help you to visualize the values of the variables in your program by displaying them in the diagram's memory cells as your program executes. The contents of the cells change over time, so you will need to cross off or erase the old values and replace them with new values as you step through your program. The contents of the memory cells at any moment in time is called the *state* of your running program, and the diagram is often called a *state diagram*.

When a field such as `x` is declared, a memory cell gets created, and the Java runtime system associates the name `x` with that cell. Numeric fields (such as integers and doubles) always get initialized with the value zero – unless otherwise defined.

Checkpoint 1: Show us your diagram illustrating the state of your running program after executing the assignment statement in `setup`. Also, show us the results of running your program.

Two integer variables. Change your code by adding a second integer variable `y`, so that the body of the class looks like this:

```
private int x;
private int y;

public void setup() {
    x = 3;
    y = x;
    System.out.println("x=" + x + " y=" + y);
}
```

On your state diagram, add a memory cell for `y`.

Checkpoint 2: Show us your state diagram after the second assignment statement in `setup`. Also, show us the results of running your program.

Modify a variable. Add one line to your `setup` method, so that it looks like this:

```
public void setup() {
    x = 3;
    y = x;
    x = x + 1;
    System.out.println("x=" + x + " y=" + y);
}
```

Start with a fresh, new state diagram for this program. with memory cells for `x` and `y`. Then step through the `setup` method and modify the contents of the memory cells appropriately as each of the assignment statements is executed.

Checkpoint 3: Show us your resulting state diagram. Be prepared to explain the changes, if any, to the values of `x` and `y`.

Add a method. Create a `void` method `setup1` that does exactly the same as `setup`, and have `setup` simply call this method. Here is what the entire body of the `Fields` class should look like:

```
private int x;
pritave int y;

public void setup() {
```

```
    setup1(); // call the setup1 method
}

private void setup1() {
    x = 3;
    y = x;
    x = x + 1;
    System.out.println("x=" + x + " y=" + y);
}
```

The `setup` method is declared `public` since the FANG engine needs to call it when the "game" runs. The `setup1` method is declared `private` since it is called only inside the `Fields` class and nothing outside of this class needs to know of its existence.

Checkpoint 4: Show us that, with this change, the output of your program is the same as before.

Move fields to local variables. Let's move the `x` and `y` variables from being fields (belonging to the entire class) to being local variables belonging only to the `setup` method. To do this, remove the `x` and `y` field variable declarations, and change the `setup` method to look like this:

```
public void setup() {
    int x = 0;
    int y = 0;
    setup1(); // call the setup1 method
}
```

Now your state diagram will look different. Start with a fresh sheet of paper, and draw a large box labeled `setup`. This large box represents the environment that `setup` may create. In this example, `setup` creates two variables `x` and `y` before calling the `setup1` method. So inside the `setup` box, put smaller boxes representing memory cells `x` and `y`. When the `setup` method is called (by FANG, when it starts up), the *local* variables `x` and `y` spring into existence – but with undefined values. You can show this by putting a question mark `?` inside of the boxes. In this code, we initialize both `x` and `y` to zero.

Draw another large box labeled `setup1`. This box will represent the environment created by `setup1` when it is called. Since `setup1` doesn't create any variables (it does *use* variables, though), you will not put any smaller boxes inside of it.

So here is the **rule for accessing variables in methods**:

1. If the variable by that name is defined in the method, use that variable.
2. Otherwise use the field variable by that name in the class.
3. Otherwise it's an error.

Checkpoint 5: Show us your state diagram showing boxes for `setup` and `setup1`. Compile this program, and explain to us what happens and why.

Move the variables into `setup1`. Change your code so that it looks like this:

```
public void setup() {
    setup1(); // call the setup1 method
}
```

```
private void setup1() {
    int x = 0;
    int y = 0;

    x = 3;
    y = x;
    x = x + 1;
    System.out.println("x=" + x + " y=" + y);
}
```

Draw a new state diagram showing boxes for `setup` and `setup1`.

Checkpoint 6: Show us your state diagram showing boxes for `setup` and `setup1` and the values of variables just before the `println` statement. Compile this program, and explain to us what happens and why. Show us the results of running your program.

Parameterize things. Change your code to look like this:

```
public void setup() {
    int x = 0;
    int y = 0;

    setup1(x, y); // call the setup1 method

    // print the values of x and y
    System.out.println("x=" + x + " y=" + y);
}

private void setup1(int x, int y) {
    x = 3;
    y = x;
    x = x + 1;
}
```

Do not compile your program yet!

Your `setup1` method has *formal parameters* `x` and `y`, so your state diagram for this method should also have boxes for `x` and `y`. When the method is called, the contents of these cells will be populated with the values of the *actual parameters*.

The variables `x` and `y` are referred to inside the body of `setup1`. They are not local variables, but they are formal parameters. So we need to expand slightly the **rule for accessing variables in methods**:

1. If the variable by that name is defined in the method or is a formal parameter, use that variable.

2. Otherwise use the field variable by that name in the class.

3. Otherwise it's an error.

Do not compile your program until we come over for your checkpoint!

Checkpoint 7: Show us your state diagram when `setup` calls `setup1`, just before `setup1` returns. Predict what output your program should produce. Compile and run your program, and compare your prediction with the program's output.

Activity 2 Object references as variables

Replace the entire code in the body of your `Fields` class with the following:

```
private RectangleSprite r;

public void setup() {
    r = new RectangleSprite(0.2, 0.2);
    r.setColor(getColor("red"));
    r.setLocation(0.5, 0.5); // center of canvas

    addSprite(r);
}
```

You will need to add an `import` statement at the top of your program file to use the `RectangleSprite` class.

Compile and run your program and verify that a red rectangle appears in the middle of the game canvas.

Move the field to a local variable. Modify your code so that the field variable `r` becomes a local variable in the `setup` method. Here is what your entire class code will look like:

```
public void setup() {
    RectangleSprite r;

    r = new RectangleSprite(0.2, 0.2);
    r.setColor(getColor("red"));
    r.setLocation(0.5, 0.5); // center of canvas

    addSprite(r);
}
```

Checkpoint 8: Predict what will happen when you run this program. Explain your prediction to us, and show us your actual results.

Create a new method, again. Next create a `setup1` method, similar to what we did before, so that your program code will look like this:

```
public void setup() {
    RectangleSprite r;
    r = new RectangleSprite(0.2, 0.2);

    setup1();

    addSprite(r);
}

private void setup1() {
    r.setColor(getColor("red"));
    r.setLocation(0.5, 0.5); // center of canvas
}
```

Compile this program.

Checkpoint 9: Explain what is happening here, and why.

Parameterize the `setup1` method. Now add a parameter to `setup1`, as follows:

```
public void setup() {
    RectangleSprite r;
    r = new RectangleSprite(0.2, 0.2);

    setup1(r);

    addSprite(r);
}

private void setup1(RectangleSprite r) {
    r.setColor(getColor("red"));
    r.setLocation(0.5, 0.5); // center of canvas
}
```

Draw a state diagram showing the values of variables just before the `setup1` method returns. Compile this program and run it.

Checkpoint 10: Show us your state diagram. Explain what happens here, and why.

A final weirdness. Change your `setup1` method as follows. Do *not* make any changes to `setup`:

```
private void setup1(RectangleSprite r) {
    r = new RectangleSprite(0.2, 0.2);
    r.setColor(getColor("red"));
    r.setLocation(0.5, 0.5); // center of canvas
}
```

Draw a state diagram showing the values of variables just before the `setup1` method returns. Predict what will happen when you run this program.

Checkpoint 11: Show us your state diagram. Explain what happens, and why.

Activity 3 Clean up!

Log off your session, clean up your debris, and push your chairs in.