Handout 2
CSCI 136: Spring, 2004
11 February

Lab 1

Due 15 February

Java, Unix, and Growing a Java Garden

1 Getting Started

During lab today, you will learn how to use the Java environment on the CS Lab Mac systems.

1. Go through the Unix tutorial handout. This will teach you how to log in and out of the machines, use basic Unix commands, and edit files with xemacs.

2. Identify the function of and experiment with these Unix Commands:

```
ls         cd         cp         mv         rm         mkdir      pwd
man        chmod      cat        more       grep       head       tail
```

   Identify the function of and experiment with these Emacs Commands:

```
C-x C-s    C-x C-c    C-x C-f    C-x C-w    C-g        C-a        C-e
C-d        C-_        C-v        M-v        C-s        C-r        M-%
```

   Learn these commands – you will use them often. Hints can be found in the Unix and Emacs web pages on the course website. (Recall that the "meta" key on Macs is the "fan" key to the left of the space bar.)

3. Make a directory in your Unix account for CS 136 work (perhaps "`136`" or "`cs136`" might be reasonable). Make a subdirectory `lab1` in this new directory for files related to this lab.

4. To set up your account to run the correct version of Java, you should enter the command

```
source /private/Network/Servers/cortland.cs.williams.edu/Volumes/Courses/cs136/bin/136
```

   each time you log in. Rather than type this command every time, you can make it happen automatically by going into xemacs and adding it to the file `.local_bashrc` in your home directory.

5. Create a subfolder inside `lab1` called `Odd`*`MyName`* where you should replace *MyName* with your last name. In that folder, write, compile, and run a Java program under Unix that prints the first ten odd numbers. Call it `Odd.java`. Turn in your Java source code and compiled code (contained in the generated file, `Odd.class` by dragging the folder containing these to the folder CS136DropOff. This will be how you turn in your work for this course.

2 Lab Program

You are going to write a series of classes that will emulate the growth of a garden. Recall that each class will go in a separate file and that the file name must be the same as the class name, except that ".java" must be tacked onto the end of the file. If you mess this up you will get error messages about not being able to find your class so don't mess it up!

2.1 Getting started

I will help you to get started by providing you with the code for an interface:

```
/**
* @author kim
*
* Interface for items to be planted in a garden
*/
public interface PlantInterface {

    // modify the size of the plant to reflect growing "days" days.
    void grow(int days);

    // modify the rate of growth of the plant to reflect "inches" of rain.
    void rain(double inches);

    // modify the size of the plant to reflect the effects of a frost.
    void frost();

    // display the plant with its current location and size.
    void draw();
}
```

Use xemacs to create a file `PlantInterface.java` in a new subdirectory of `lab1` called `Garden`*`MyName`*. To make sure you haven't made any mistakes in typing it, execute:

```
javac PlantInterface.java
```

If you get any error messages from the Java compiler, go back and fix the errors and recompile until there are no more errors.

2.2 Writing the Flower class

Now it is time for you to write a class `Flower` that implements `PlantInterface`. The constructor should take parameters that indicate the x and y coordinates (as integers) of the base of the flower. It should set the initial height of the flower to be 30 and its width to be 8 (use constants that are doubles!). The initial growth rate should be half of its height. For each day that the plant grows (via the `grow` method), the height should increase by the growth rate (i.e., two days growth should result in twice as much change as one day). The width of the flower does not change as it grows. Rain has the effect of increasing the growth rate by `10*inches` percent. A frost kills the plant, resulting in setting the growth rate, height, and width to 0.

Because we are not including any graphics, the `draw` method should simply result in writing (using `System.out.println`) a message saying that it is a flower and what its location and size is. To better prepare for later parts of this lab, first write a method`public String toString(){...}` that composes the string to be printed and then the body of the `draw` method will just be `System.out.println(toString());`.

Test your program by adding the following method as the last method in the class:

```
public static void main(String[] args) {
    Flower myFlower = new Flower(20,40);
    myFlower.draw();
    System.out.println("growing 2 days");
    myFlower.grow(2);
    myFlower.draw();
    System.out.println("raining and growing 2 days");
    myFlower.rain(5);
    myFlower.grow(2);
    myFlower.draw();
```

```
        System.out.println("growing 1 day");
        myFlower.grow(1);
        myFlower.draw();
        System.out.println("Frost happens");
        myFlower.frost();
        myFlower.draw();
        System.out.println();
    }
```

Compile it, and run the program by typing `java Flower`.

Adding a method like this at the end of a class is a common trick of Java programmers to incrementally test classes before assembling them into a larger program. Make sure that the results you get are correct.

Be sure to document your class with appropriate comments, including a general description at the top of the file and a description of what each method and constructor does. Comments for most methods can be copied from the interface comments. Be sure to use descriptive variable names and provide a comment on all variables and constants to make clear what they stand for. The code in your constructor and method bodies typically will not need comments unless there is something tricky going on that needs extra explanation. Do *not* add comments that simply repeats what is obvious in the code.

2.3 Preparing a garden

Now add a new class `FlowerGarden` that represents a collection of plants. The garden should have the capacity to hold 100 flowers (in an array of objects of type PlantInterface). For reasons that will become apparent only later, declare `FlowerGarden` to implement `PlantInterface`. When the `grow` message is sent to a `FlowerGarden`, the garden should send a `grow` message (with the same parameter) to all of the flowers in the garden. Similarly for `rain`, `frost`, and `draw`. Also add a method `plantNewPlants` that can be called to add several new plants to the garden. The constructor for the garden should take as parameters the coordinates of the upper left hand corner of the garden, the width and height of the garden, and the number of flowers to be initially planted in the garden. The constructor (perhaps via a call to `plantNewPlants`) should create the appropriate number of flowers whose x and y coordinates are within the confines of the garden. Use a random number generator from class `Random` (look it up and the `nextInt` method in the on-line Sun Java documentation) to generate random starting locations for the flowers. (Don't worry if they overlap each other!).

Test the class by adding the following method to your class:

```
    public static void main(String[] args) {
        FlowerGarden backyard = new FlowerGarden(0,0,400, 600, 6);
        backyard.draw();
        FlowerGarden sideyard = new FlowerGarden(400,0,400, 600, 6);
        System.out.println("growing");
        sideyard.grow(1);
        backyard.grow(1);
        backyard.draw();
        sideyard.draw();
        backyard.rain(5);
        System.out.println("growing");
        backyard.grow(1);
        backyard.draw();
        System.out.println("Frost happens");
        backyard.frost();
        backyard.draw();
        System.out.println("Adding new flowers to backyard");
        backyard.plantNewPlants(12);
        backyard.draw();
    }
```

Compile and run the program to make sure your program is working properly. [As a safeguard, from now on compile all of the ".java" files rather than just the most recently changed one. You can do this by typing `javac *.java`. This keeps you from using inconsistent versions of classes if you recompile one that has been used in another class.]

2.4 Adding bushes to the garden

Design a new class `Bush` that implements `PlantInterface`. Rather than just growing straight up, bushes also get wider as they grow. As a result the methods from Flower will have to be modified slightly to take into account the changes in width. Also the `toString` method will have to be modified to print out that the object is a bush rather than a flower. You can choose the initial width, height, and growth rate. Compile and test this to make sure it works.

Now modify (temporarily) your `FlowerGarden` class to grow bushes instead of flowers. In my version of this class, I only needed to change the call of the constructor in the `plantNewPlants` method. Hopefully you will also manage with a small change like this one. Notice that because your array holds items of type `PlantInterface`, it is quite happy to hold bushes as well as flowers. Test this to make sure the garden of bushes grows properly, and then change the `FlowerGarden` class back.

2.5 Using inheritance to factor out common code

You will have noticed that the `Flower` and `Bush` classes share most of their code. It is considered to be bad style to have lots of shared code between classes. Instead it is preferable to move the shared code into a *superclass*.

Design a new *abstract* class, `Plant`, which will implement `PlantInterface`. It's header should look like

```
public abstract class Plant implements PlantInterface {
```

Copy all of the instance variables and methods from `Flower` to `Plant` except for the `toString` method and the constructor. Also do not copy the static `main` method. The constructor for `Plant` should take four parameters: the width, height, and x and y coordinates, each of which should be saved in an instance variable. While the instance variable for growth rate will be declared in the `Plant` class, it will not be initialized. It is important to make sure that all of your instance variables are declared to be `protected` and not `private`.

Also add the method declaration:

```
public abstract String toString();
```

This indicates that some of the other methods in the class may depend on a method `toString`, but it is not supplied. It is also the reason that the class must be declared to be abstract.

While the abstract class `Plant` has a constructor, you may not invoke it because the class is abstract (because of the missing implementation of `toString`). Instead the constructor is defined solely so that it can be extended by other classes.

Now go back to the class `Flower` and change the header so that it extends `Plant`. You no longer need to say that it implements `PlantInterface`, as Java can deduce that from the fact that it extends a class that implements the interface. Now erase all of the methods except for `toString` and `main`. The other methods will be *inherited* from `Plant`. Also erase all of the instance variables. [Very bad things happen if you repeat the declaration of instance variables from a superclass into a subclass – the compiler treats them as distinct from those declared in the superclass.]

The constructor of `Flower` can now be greatly simplified, as you need only invoke the constructor of the superclass, using the appropriate parameters. The constructor of `Flower` takes two parameters, the x and y coordinates of the flower, as all flowers have the same initial width and height (which hopefully you declared as constants when you originally wrote the class). You can now replace almost the entire constructor body by a call of the superclass constructor, written as:

```
super(...,...,x,y);
```

where the "..." are replaced by the constants representing the inital width and height of the flower. The call to `super` must be the first statement of the `Flower` constructor. The only other thing that must be done is to initialize the growth rate to half of the initial height.

Notice that the constructor and `toString` method refer to instance variables declared in `Plant`. This is possible because they are declared to be `protected` rather than `private`. Protected variables are accessible from within subclasses, while private variables are not. Again, make sure you have not repeated the declarations in the subclass.

Now, if you have done all of this correctly, you should be able to compile all of the classes and execute the `Flower` class, and get the exact same results as you did before. If you have errors, track them down – especially making sure that you got rid of all of the instance variables from `Flower`.

This was a lot of extra work, but if you had done this originally, it would have been much easier to write the `Bush` class and any other classes with similar behavior. Now let's change the `Bush` class so that it also inherits from `Plant`. Start by changing the header to declare it extends `Plant`. Again, all of the necessary instance variables were declared in `Plant`, so should not be repeated in `Bush`. The constructor will be modified so that it is similar to that in the revised `Flower` class.

The `Bush` class will need two methods, `toString` and `grow`. The `toString` method will remain as it was (indicating that it is a "bush"), while the `grow` method is different from that of `Plant`, in that now the width must also increase as it grows. Thus the `grow` method needs to perform the same actions as the method in `Plant`, but then also do more. If we add a `grow` method to the `Bush` class, then it will replace the `grow` method inherited from `Plant`. However, we would like to perform the actions of the `grow` method from the superclass, as well as the extra operations. We can execute the code in the superclass, by including the line:

```
super.grow(days);
```

inside the method body if `days` is the name of the formal parameter of the method. Then just add the extra code to update `width` after that line.

2.6 Going further ...

Of course now we'd like to create gardens that use bushes rather than just flowers. I'd like you to create a class `BushGarden` that is similar to `FlowerGarden`, but which grows bushes. You can probably see that this could be easily done by just copying the code in `FlowerGarden` and changing the line or two that involves calling the `Flower` constructor. However, now that you know about how to create abstract superclasses so that code can be shared, I'd like you to try that.

Create a new abstract class `Garden` that implements `PlantInterface` and contains all of the code from `FlowerGarden` that can also be reused in `BushGarden`. (Start by just copying and pasteing the code from `FlowerGarden`.)

Here is a hint to get the greatest mileage out of constructing this class. Add a new abstract method

```
public abstract PlantInterface newPlant(int x, int y);
```

and replace all occurrences of "`new Flower(...,...)`" by a call of the `newPlant method`. Then in `FlowerGarden` you need only have a constructor that calls the superclasses constructor (recall `super(...)`) and write the method `newPlant` so that it returns the result of creating a new `Flower`. (All other methods and instance variables will appear only in the `Garden` class.) You can then write `BushGarden` similarly.

That is all you need to turn in for this program. However, consider what you would need to do to make the following additions to the program.

1. Suppose you wished to create a garden in which half the plants were flowers and half were bushes. How could you accomplish this by defining a subclass of `Garden`? *(Hint: You need only modify* `newPlant`*).*

2. Suppose you wished to create a class `GridGarden` which is composed of *numGards* gardens in a row – where the smaller gardens alternate between being flower gardens and bush gardens. Interestingly the hardest part of defining this new class as a subclass of `Garden` is that Java *requires* the first statement in the constructor to be a call to the superclass constructor. To make this work, you might need to move constructor code down from `Garden` to `FlowerGarden` and `BushGarden` or alternatively insert a new abstract class called `RandomGarden` between `Garden` on the one hand and `FlowerGarden` and

BushGarden on the other. The constructor from Garden could be moved down to RandomGarden. GridGarden could extend Garden, while FlowerGarden and BushGarden could extend RandomGarden. (Notice that this exercise is easier because the gardens each implement PlantInterface, and hence can be stored in the array.)

If you would like to add these extensions, they would be worth a modest amount of extra credit. However, I would be happy just to have you think about how these extensions might be accomplished.

One idea that I hope you have seen in doing this lab is that you should not be afraid to modify (*refactor*) a program as you proceed in its development. While the classes you defined for this lab were pretty trivial, in big programs these refactorings can result in programs that are much easier to maintain because you aren't keeping multiple copies of blocks of code. A change in one place will impact all the sites where it is used.

When you are finished, be sure that all of the classes are well documented (including putting your name in the comments in each) and turn in the entire folder in the same way you did before. As in all labs, you will be graded on design, documentation, style, and correctness. **This program is due by 11:59pm on Sunday, 15 February.**