CSE 142 - Su 02
Homework 3

Assigned: Wednesday, July 10
Due: Wednesday, July 17, BEFORE MIDNIGHT

** General Comments about the Homework **

All homework is turned in electronically. Go the the class web site and use the link on the homework page to do the turnin. Don't be late! Late homeworks will not be accepted.

The homework assignments all have three parts: Practice Problems, Graded Problems and a Programming Project.

** Homework 3 Practice Problems ** Do not turn in the Practice Problems.

1. Say we run the following lines of code:

```
int x = 5;
double y = 6.0;
boolean z = false;
```

Determine the result of evaluating the following expressions.

```
a. (x > 4) || z
b. (x > 4) && z
c. (x > y) || (!z)
d. ((x > 6) && (y < 6.5)) || (x > 5)
```

2. You are an auto insurance company, and store customer data by maintaining an object of type Customer for each customer. The definition for this class is as follows:

```
public class Customer() {

  String name;    // name of customer
  int age;        // age of customer
  boolean male;   // true = male; false = female
  int tickets;    // number of traffic tickets received

  /* ... constructor and methods here ... */

}
```

Write a method for this class called getPremium() that will return the monthly premium for a customer in dollars. The method used to calculate the premium for a customer is as follows:

- Start with $100 if customer is 18-25 years old, $70 if 25-45 years old, or $50 otherwise. (We will assume all customers are over 18.)
- If customer is male, multiply by 1.2.
- Add $20 for each ticket received.

3. Write legal Java variable definitions and statements to calculate the average area of the following three triangles. Recall that the area of a triangle is 1/2 * base * height. This time you're responsible for defining all the variables you need; we're only giving you the numbers:

```
Triangle 1: base = 3.5, height = 5
Triangle 2: base = 4, height = 6
Triangle 3: base = 2.5, height = 8
```

4. When dealing with objects like a Dog named rover, == and the "equals" method may do different things. If you want special behavior for equals, you need to write a method that implements it. Think about examples where you might use "==" to compare Dog variables, and another example where you might use a new equals() method.

5. When you have finished the requirements for the programming problem below, feel free to extend it. You now have the tools to implement significant and interesting behavior in your various objects.

You can define any keyword/value pairs that you want just by adding them to the constructor and doAction, and then putting some examples in one or more of the property files in the cast directory. You could define additional bounce limits, variable speeds (gravity?), more color changes, age related speed or direction changes, sounds at the bounce, different kinds of moving Shapes (square, polygon, image, text, ...), etc, etc.

If you define an extended class, please duplicate your basic OvalSlider.java and name the duplicate some new name (like XActor.java) and then edit the new file to define the XActor class. Remember that in order to actually get an object of this class, you have to create a property file that contains the line "class : XActor". If you do this, you will hand in both OvalSlider.java with the required basic changes, and XActor.java with your extensions.

Describe your extensions in extensions.txt if you do this.

** Homework 3 Graded Problems ** Turn in your answers to the graded problems in a text file.

Each of the following questions is worth 2 points.

1. Say we run the following lines of code:

```
int x = 5;
double y = 6.0;
boolean z = false;
```

Determine the result of evaluating the following expressions.

```
a. (x > 6) && ((y < 6.5) || (x > 5))
b. (y > 3.0) && (y < 6.5)
c. !(y > 6.2) && (y < 6.5)
d. z || !z
```

2. Write a method named "factorial" that takes an integer n as a parameter, and returns the factorial of n (i.e., the value n * (n-1) * (n-2) * ... * 2 * 1) as an integer result. Include the proper javadoc comments to describe the purpose of the method, the parameter and the result.

3. Write legal Java variable definitions and statements to calculate

(3a) the circumference of a circle (formula: 2 * PI * radius)
(3b) the area of a circle (formula: PI * r * r)

Assume that "radius" has been declared as a "double" variable and that it has a valid positive value. Assume that the value of PI is available as "Math.PI".

The names of the variables you use are entirely up to you, but your solution should be understandable by someone else. Since you won't actually be running

the code, you don't have to compute the results -- just write the code that would do it for you. This is just a sequence of statements, don't worry about making it a complete method.

4. Consider the following pieces of code. For each part, say what the value of the variable "answer" is at the end.

(4a)
```
 int answer = 0;
 int x = 3;
 int y = 1;

 while (y < x) {
  answer += 5;
  y++;
 }
```

(4b)
```
 int answer = 0;
 int x = 0;
 int y = 4;

 if (x == y) {
  answer = 17;
 } else if (x == answer) {
   answer = y;
 } else {
  answer = x;
 }
```

(4c)
```
 int answer = 0;
 int x = 0;
 int y = 1;

 if ((x == y) && (answer < 1)) {
  answer = 4;
 }
```

(4d)
```
 int answer = 0;

 for (int i=0; i<100; i++) {
  answer += 2;
 }
```

5. Consider the class OvalSlider from this homework project. Read through the constructor for OvalSlider and identify the following.

(5a) All the parameter variables used in the constructor
(5b) All the local variables used in the constructor

** Homework 3 Programming Project **

This project implements moving graphics objects. You now know enough about conditional branching and looping to be able to implement some interesting behavior.

You are to modify one existing class (named OvalSlider) to implement some new behavior for objects of that class. You can use text files to define several instances of your objects, then set them all to running and watch the show.

++ List of classes ++

As provided to you, homework 3 is a complete and running program. The overall concept of the program is that it builds a set of Actors which can draw images on the graphics window, and then it loops continuously and calls the doAction() method of each Actor every time the clock ticks. Since Shapes can be moved and recolored while they are on screen, you can control the Shapes and draw simple animated scenes.

The files involved in this program are the following.

* Property files in subdirectory "cast" *

This directory holds one file for each element in your picture. These files define the properties of your object. Each property is defined as a pair of strings. The first string is the "key", the second string is the "value". So, for example, to define the initial color of an Oval, you would have an entry that says

```
color : red
```

in the property file for this actor. You can look at the files that are already in the cast directory to see more examples. Each property file ends with the extension .prop. Each property file must contain an entry that defines the class to use in building the object. In the examples, that entry is the first line that says "class : OvalSlider". The value must be the name of the class to use, spelled correctly with the correct capitalization.

NOTE: The first time you look at one of these .prop files, you need to make sure that you can open it in a text editor. In Windows, you should right-click on one of the property files, and then select "Open with...". In the dialog, scroll down to the end of the list and select WordPad. Then check the box that says "Always use this program to open these files." Then click OK. In the future you can just double-click a .prop file and it will open in WordPad.

* OvalSlider.java *

This is the class that you will modify. There is one constructor and three methods. You should have a copy of OvalSlider.java in front of you while you read this description.

The constructor is called when an object of this class is first created. The constructor is called with one parameter, an object called a Map. The nice thing about a Map is that it can hold many objects. You can retrieve information from the Map by using a key string. Take a look at the existing code to see how this is done.

For example, to retrieve the value that was specified in a property file for the width of a new Oval, the code is as follows.

```
    arg = (String)pMap.get("width");            // get the String associated
with "width"
    if (arg != null) {                          // if there was anything there
...
        width = Integer.parseInt(arg);          // ... decode and store the new
value
    }
```

You can use a little snippet of code like this to retrieve and store any key / value pair that you want to define in the property file. Notice that width is an instance variable, so that the value is available even after the constructor has finished running. Also notice that I set the default value of width up at the start of the constructor.

The constructor is responsible for decoding the information that was provided in the property file and storing it in the appropriate instance variables for later use by the doAction() method. When you define new capabilities for OvalSlider, there will be changes in the constructor to get the property values, and there will be changes in doAction to actually implement the new capabilities.

The doAction(Stage stage) method is called by the animation framework over and over to implement the behavior of your Actors. So doAction is called hundreds of times in the course of one run of this program. Each time it is called, you can move the Shape a little bit, or change its color, or whatever is needed.

As provided to you, the doAction method moves the Shape a little bit each clock tick, and bounces it off the walls of the window frame when it reaches an edge. You should look at the "if" statements that implement this behavior and make sure you understand what is happening.

There are several changes to be made to the doAction method. They are described later.

The addTo(GWindow w) method is called by the animation framework early on when the Actor is first created. You don't need to make any changes here.

The removeFromWindow() method is called by the animation framework to remove an Actor from the stage. You don't need to make any changes here.

* Director.java *

This class contains the main method that is used to start the program running. In BlueJ, you can right-click on the class definition square for Director, and then select "void main(arg)" It will ask you for an argument to pass to main. You can just click Ok, and it will start running. The main method creates a new Play object, which reads in all the property files and creates the new Actors, and then tells the Play to run for 500 clock ticks. You don't need to make any changes to this class.

* Play.java *

This is the class that uses the property files and creates a new object for each one. You should not make any changes to this class.

* DirList.java *

This is a utility class that creates a list of all the property files in the subdirectory. You should not make any changes to this class.

* PropertyMap *

This is the class that actually reads in the properties defined in a property file and creates the Map that is passed to your constructor. You should not make any changes to this class.

++ Project Requirements ++

1. Download the cse142-hw3.zip file and unzip it. The project skeleton is in the directory hw142\hw3. Your task is to open the project using BlueJ and modify OvalSlider to meet the requirements below.

2. Start BlueJ and open the hw3 project. Open the OvalSlider source file and read it. You can see that it will compile and run, but it doesn't implement all the functions that are needed.

3. The function of an Actor class like OvalSlider is to create and then control one or more Shapes that are shown in the graphics window. Some of the capabilities of OvalSlider are already implemented; your job is to implement more capabilities as described below.

3a. Add the ability to bounce off of a limit other than just the right or left edge of the window. In other words, implement new parameters called rightWall and leftWall, and then bounce the Shape when its position is greater than or less than those values.

You should define instance variables to hold the rightWall and leftWall values.

In the constructor, you should provide default values for those variables, then check to see if there are values defined in the property file. If so, store the defined values. Otherwise, use the default values. The default values should be a very large integer for rightWall, and a negative integer for leftWall.

In doAction, you should add a little bit of code to the bounce code to check and see if the position of the object is greater than rightWall or less than leftWall, and if it is, then reverse direction. This is not a lot of code, you are just adding conditions to the existing if statements.

When this is working correctly, the red ball in the cast will bounce in a narrow band in the middle of the window. You can look at zoomer.prop to see how the values for rightWall and leftWall were specified.

3b. Add the capability for the Ovals to age and change as they age. In other words, you need to keep an integer instance variable that is incremented by 1 every time your method doAction is called. When the age (the number of calls) is equal to a value specified as lifeSpan, then you change the object color to Color.gray and divide the deltaX and deltaY values by 2. When the age is equal to twice the lifeSpan, then you restore the original color of the object, and restore the original deltaX and deltaY values, and set the age back to zero to start another cycle.

In order to implement this, you need to make several changes.

In the constructor, you need to check for an integer value for "lifeSpan". Store this in an instance variable. You should initialize it to 0 in case the property file doesn't supply any other value. You need to initialize an instance variable "age" to 0, so that doAction will know how old we are. You need to store a copy of deltaX and deltaY in instance variables initialDeltaX and initialDeltaY so that doAction can restore these values when it needs to.

In doAction, you need to make use of the information that was stored in the constructor. A general description of the logic is:

```
increment the age value by 1
if there is a defined lifeSpan that's not equal zero then
    if the current value of age is equal to lifeSpan then
        change the color of theShape to Color.gray
        divide deltaX and deltaY by 2
    else if the current value of age is equal to 2*lifeSpan then
        change the color of theShape back to the original color
        set deltaX back to initialDeltaX
        set deltaY back to initialDeltaY
        set age to 0
```

When this is working correctly, any Actor that is given a positive lifeSpan in its property file will live for a while, then change to gray and slow down for a while, and then once again get back its original color and speed, in a repeating cycle.