



## CS 101 - Problem Solving and Object-Oriented Programming

### Lab 2 - Build a Snowman

**Due: September 25/26/27/28, 11:55 PM**

#### Pre-lab Preparation

Before coming to lab, you are expected to have:

- Read Bruce chapters 3 & 4
- Do exercises 4.2.2 and 4.3.3 in the book. If you are uncertain of the answers, enter the programs to see what happens. You should turn in your answers to these questions on paper at the beginning of lab.

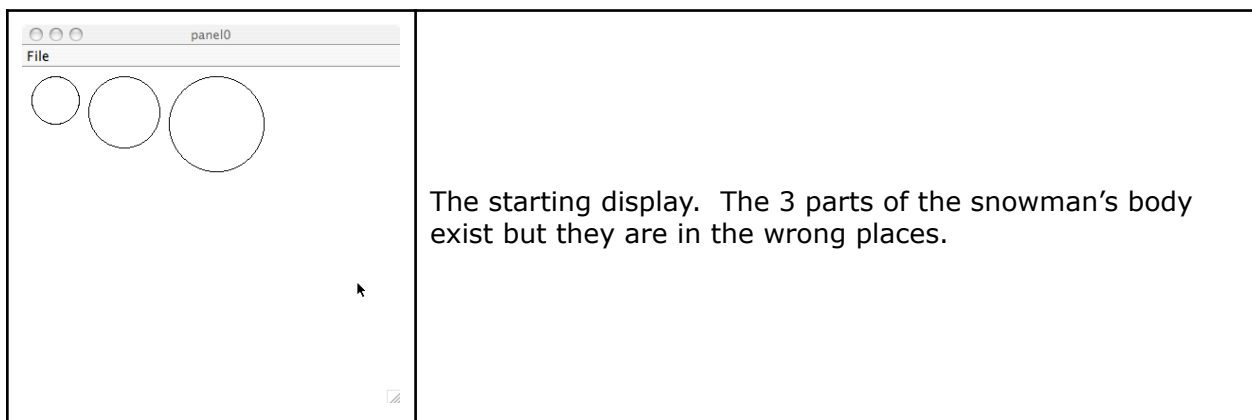
#### Goals:

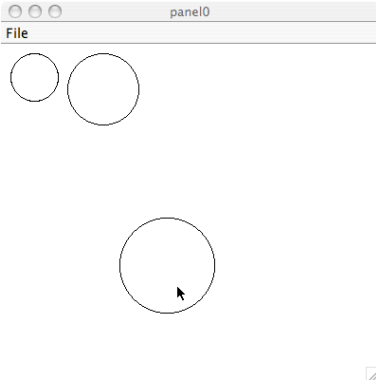
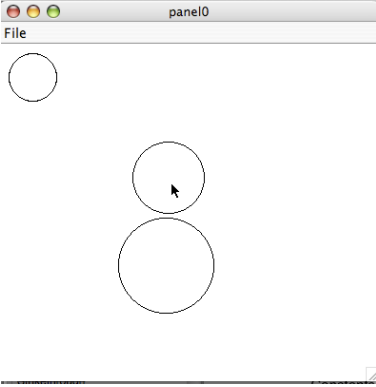
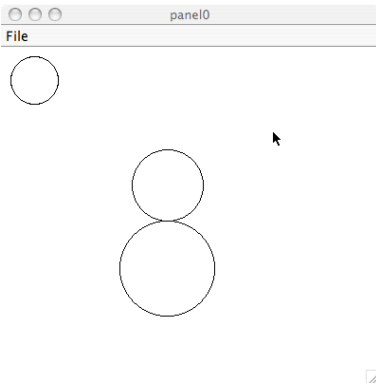
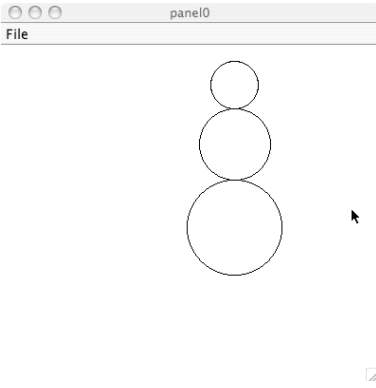
- To get practice with conditionals and if-statements
- To implement centering and dragging algorithms

#### Introduction to the Assignment

In this lab, you will create a program that will allow a user to build a snowman from 3 circles. When the program begins, it will display 3 circles. The user's job is to move the circles into position to form a snowman. The program helps by "snapping" pieces together when they are in nearly the right position.

Here are some snapshots from this program.



	<p>The user has dragged the bottom to a better location.</p>
	<p>Next, the user drags the middle snowball into nearly the right position.</p>
	<p>When the user releases the mouse button, the middle ball snaps into position since it was close to its proper location. If the mouse button is released with the middle ball out of position, it would not move.</p>
	<p>Now the user has moved the head into place.</p>

## Writing the Program

To get started, create a project named Lab2. Within the project, create a class called BuildASnowman. Replace the template that BlueJ gives you with the following template, like you did last week:

```
import java.awt.*;
import objectdraw.*;

/**
 * Write a description of class BuildASnowman here.
 *
 * @author (your name)
 */
public class BuildASnowman extends WindowController
{

}
```

### Step 1: Creating the initial display

Recall that the initial display is defined by the begin method. Therefore, you should add the following to your class between the { }.

```
/**
 *
 */
public void begin () {

}
```

In the initial display, you should draw 3 framed ovals across the top as shown above. You should define some constants (private static final) to help with the locations and sizes of these ovals.

You will want to save each framed oval in a separate instance variable so that you will be able to move it later. For example, creating the oval that will become the head will look something like this:

```
public class BuildASnowman extends WindowController
{
    /* The distance from the left of the display for the first ball
       initially. */
    private static final int LEFT = 10;

    /* The distance from the top of the display to the tops of the balls
       initially. */
    private static final int TOP = 10;

    /* The diameter of the snowman's head */
    private static final int HEAD_SIZE = 50;

    /* The ball that represents the snowman's head */
    private FramedOval head;
```

```

/**
 *
 */
public void begin () {
    head = new FramedOval (LEFT, TOP, HEAD_SIZE, HEAD_SIZE, canvas);
}
}

```

Do something similar to create the middle and bottom snowman balls, saving them into instance variables.

Run your program to be sure that the display looks roughly like the first screenshot in this hand-out.

## Step 2: Dragging a ball

Users should drag the balls into position to build a snowman. Recall from the example in class that you will need an instance variable to label the last mouse position before the drag so you can determine how far to drag the ball. You will also need to know which ball to drag. As a result, you will need 3 boolean variables to remember which of the 3 balls is grabbed, if any. If the user presses the mouse button down outside all 3 balls, none of them should move.

Your ball dragging code will need to be spread out over 3 methods:

- In `onMousePress`, remember which ball was clicked on, if any. To do this, create 3 boolean instance variables: `headGrabbed`, `bellyGrabbed`, `feetGrabbed`. To determine if a ball is clicked on call the `contains` method defined for `FramedOvals`. The `contains` method is already defined. It has the following signature:

```
public boolean contains(Location point)
```

You do not type this. This is information to tell you how to call the method. Specifically, it is telling you that the name of the method is "contains". The method returns a boolean value. This means that it can be called as the condition of an if-statement. It takes a single argument whose type is `Location`.

The `contains` method will return true if the `Location` object passed in as a parameter is within the shape that the method is called on. It will return false if it is outside the shape. Put a call to the `contains` method inside an if statement to determine which ball to drag, like this:

```

if (head.contains (point)) {
    headGrabbed = true;
}
...

```

Note that if two balls overlap and the user presses the mouse in the area where they overlap, only one of the balls should move.

- In `onMouseDown`, use the value of the "grabbed" variables to determine which ball to move. Then, mimic the dragging code that we did in class to actually move the ball.
- In `onMouseRelease`, set the 3 "grabbed" variables to false to let your program know that the user is no longer dragging any of the balls.

Start just trying to drag 1 ball. Compile and run your program. Make sure that ball moves when it is dragged. Make sure no other balls move. Also, make sure that if you press & drag outside that ball, it does not move. When your code works for a single ball, modify it to work with any of the balls.

### Step 3: Moving a ball into position

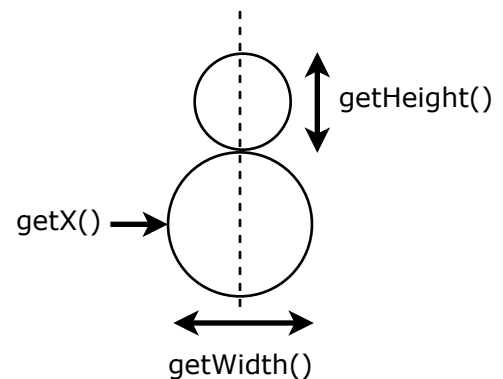
Some drawing programs are helpful about moving objects together when they are near each other, rather like magnets. Your snowman drawing program should be helpful in this way. In particular, if the user gets a ball within 10 pixels above/below and 10 pixels left/right of where it should be, your program should snap it into position when the user releases the mouse.

Let's first see if we can get our program to move a ball into position no matter where it is when we release the mouse button. To do this, we need to figure out where it should be. The "right" location for the head is centered in the horizontal dimension with respect to the belly and with the bottom of the head being at the same y coordinate as the top of the belly.

Since the user can drag the belly around, there is no fixed correct location. Therefore, you can not define named constants for the correct location of any of the body parts. The correct location of the head depends on where the belly currently is.

To determine the correct x coordinate for the head, we need to center the head over the belly. We have seen how to do this when we are working with constant values. We do something very similar in this case, except that we need to first ask the belly where it is. Fortunately, there are two methods defined on all of our 2D shapes that give us this information:

- `public double getX()`
- `public double getY()`



These methods are already defined, so you do not type the lines above. Again, this just gives you information about how to call them. Specifically, the names of the methods are "getX" and "getY". The methods take no arguments. The methods return a double (a number with a decimal point). You can call these methods anywhere that it would be legal to use a double value.

For example, we could use these methods to determine where the center of the belly is:

```
double bellyCenterX = belly.getX() + BELLY_WIDTH / 2;  
double bellyCenterY = belly.getY() + BELLY_HEIGHT / 2;
```

To move the head so that it is centered, we cannot tell it where the center should go. Instead, we need to tell it where the left edge of the head should go. If we want the head and belly to be centered, where should the left edge of the head be? It should be  $\frac{1}{2}$  the width of the head to the left of the head's center. Where should the head's center be? Horizontally, the head's center should be the same as the belly's center. So, we will want to move the head to:

```
double headLeft = bellyCenterX - HEAD_WIDTH / 2;
```

Now that we know where the left edge of the head should be, how do we actually move it there? There is a method called "moveTo" that is defined on the 2D shapes that allows us to move an object to a specific coordinate. Note how this differs from the "move" method, which moves an object a distance relative to its current location. The signature of the moveTo method is:

```
public void moveTo (double x, double y)
```

This tells us that we can call `moveTo` by passing it 2 arguments: a number representing the x coordinate the object should move to and a number representing the y coordinate the object should move to.

If we pass `headLeft` as the first argument in a call to `moveTo`, the head should now move to the appropriate x coordinate, but what about the y coordinate? We want the head to sit directly on top of the belly. We can call the `getY()` method to find where the top of the belly is. To determine where the top of the head should be, we need to do a little work. We need to subtract the height of the head from the top of the belly:

```
double headTop = belly.getY() - HEAD_SIZE;
```

Putting these together in a `moveTo` call should move the head when we release the mouse button:

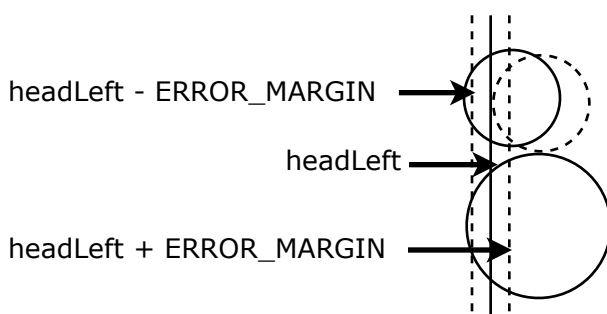
```
head.moveTo (headLeft, headTop);
```

Try it! Remember that it will always move the head when you release the mouse button at this point, not just when the head is in nearly the right position.

After this works, write similar code that moves the “feet” into position below the belly when the “feet” are dragged and released.

#### Step 4: Snapping a ball into place

At this point, our program is doing too much for the user. Let’s change it so that the user must get the ball being dragged close to the right position before snapping it into place.



Assuming the user is dragging the head, we want to figure out if the head is close enough to the belly so that it should snap into position. From the calculations above, we know where the head should be. The `headLeft` variable is holding the x coordinate of where the head should be. We can use the `getX()` and `getY()` methods on the head to find out where it currently is. If it is within some error margin of the correct location, we should move the head into position. So, the `moveTo` statement you previously entered should be inside an if-statement where you check for the

correct position:

```
if (head.getX() > headLeft - ERROR_MARGIN &&
    head.getX() < headLeft + ERROR_MARGIN &&
    // Do something similar for the y values ) {
    // move the head using your earlier moveTo statement
}
```

`ERROR_MARGIN` is a constant that you need to define. `&&` means and. The entire condition will be true only if all the parts of the condition are true.

In the diagram above, the dashed head shows where a properly placed head would go. The solid head shows where the user dropped the head. The solid vertical line indicates the proper x co-

ordinate for the head. The two dashed vertical lines indicate the minimum and maximum x values that we will consider close enough to allow snapping into place. In this case, the head should NOT snap into position because it is too far to the left.

Next, create a similar if-statement to snap the feet into position if they are near their correct location centered below the belly.

Doing the belly is a bit trickier. If the belly is moved near the correct position below the head, it should snap into place there. Otherwise if it is moved near the correct position above the feet, it should be moved there. If it is near neither, it should remain wherever it is when the user releases the mouse.

Notice that if you drag one of the balls after they have snapped together, they will come apart again. That is fine.

## **Extra credit**

The purpose of extra credit is to give you an opportunity to challenge yourself. As a result, we expect extra credit to be done independently. TAs and instructors will not provide assistance with extra credit.

For extra credit on this lab, you can implement the following feature. When two shapes are snapped together, dragging one shape should cause both shapes to move. When all three shapes are snapped together, a text message saying "Good job!" should appear and it should no longer be possible to move the snowman at all.

## **Style Issues**

### **Variable names**

In addition to the style issues from last week, this week you should also be careful in choosing the names of your variables. They should be descriptive of their purpose. In addition, they should follow this naming convention. A variable name should use all lower case unless it is two or more words put together. In that case, the interior words should be capitalized to make the name easier to read. This style for variable names is called camelcase, appropriately enough! For example:

- `int age;`
- `int ageInDogYears;`

### **Variable scope**

Variables should be declared as local variables whenever possible. That is, if a variable is needed only within a single method it should be declared in that method.

If the same variable is needed in multiple methods, it should be declared as an instance variable. That is, its declaration should be inside the class but before all of the methods in the class.

Instance variables should always be declared as "private". Local variables do not have "private" in their declarations.

## **If Statements**

When you have a number of choices where you know that at most one can be true, you should use "if" for the first condition and "else if" for the later conditions. If there is a final catch-all

that should be done if none of the conditions is true, you simply say "else" with no condition. For example:

```
if (age < 16) {
    // Do something for people too young to drive
}
else if (age < 18) {
    // Do something for people old enough to drive but too young to vote
}
else if (age < 21) {
    // Do something for people old enough to vote but too young to drink
}
else {
    // Do something for people old enough to drink
}
```

You need to be careful how you order the parts of your if-statement. For example, if you wrote the following, you would not get the desired effect. The 2<sup>nd</sup> and 3<sup>rd</sup> parts of the if-statement would never be executed, no matter what value age had.

```
if (age < 21) {
    // Do something for people old enough to vote but too young to drink
}
else if (age < 18) {
    // Do something for people old enough to drive but too young to vote
}
else if (age < 16) {
    // Do something for people too young to drive
}
else {
    // Do something for people old enough to drink
}
```

## Conditions

Conditions should be expressions that evaluate to true or false. They can be boolean variables, like `headGrabbed`, or method calls that return boolean values like "contains". You should never say:

```
if (headGrabbed == true)
```

That is equivalent to the simpler

```
if (headGrabbed)
```

Similarly, do not say:

```
if (headGrabbed == false)
```

That is equivalent to the simpler



```
if (!headGrabbed)Grading
```

## Grading

2	Pre-lab work
5	Step 1: Drawing the initial display
10	Step 2: Dragging the balls
10	Step 3: Moving the balls into position
10	Step 4: Snapping the balls into position when close
10	Compiles without error
5	Comments
4	Constant declarations
4	Indentation
5	Variable naming
5	Variable scopes
5	If statements
5	Conditions
<b>80</b>	<b>Total</b>
<b>5</b>	<b>Extra credit</b>

## Turning in Your Work

Your work will be automatically collected from your dev/cs101/Lab2 folder at the time that it is due. Please be sure your files are in the right place.

Over the course of the semester, you may have up to 5 late days total. Use them wisely! To request a late day, fill out the Google form on the course website so that we do not collect your assignment when it is due.