IAT 265 Week 3

Transformations and Vectors

Topics

- Transformation: Translation, Rotation, Scale (in Java)
  - *Graphics2D* for transformation
  - Case study: translate, rotate, scale a ladybug
- AffineTransform
- Vectors and targeted motion
  - Velocity operations for moving and chasing
  - PVector (from Processing) for implementation
  - Case study: Ladybug approaches a Seed

Graphics2D

- *Graphics2D* class is a newer version drawing tool than *Graphics* class

- It is a subclass of *Graphics* but is more powerful in the sense that:
  - It draws or fills primitives in a more flexible way
  - It can draw more primitives than its parent class, e.g. curves

    (These two topics will be covered next week)
  - It provides transformation functionalities (*translate, rotate, scale etc.*)
  - It can draw with better quality (anti-aliasing)

Transformation

- Translation
  - Is about shifting the drawing space to a new location
  - For graphics programming, this means moving the origin (0, 0) to a different location in our window

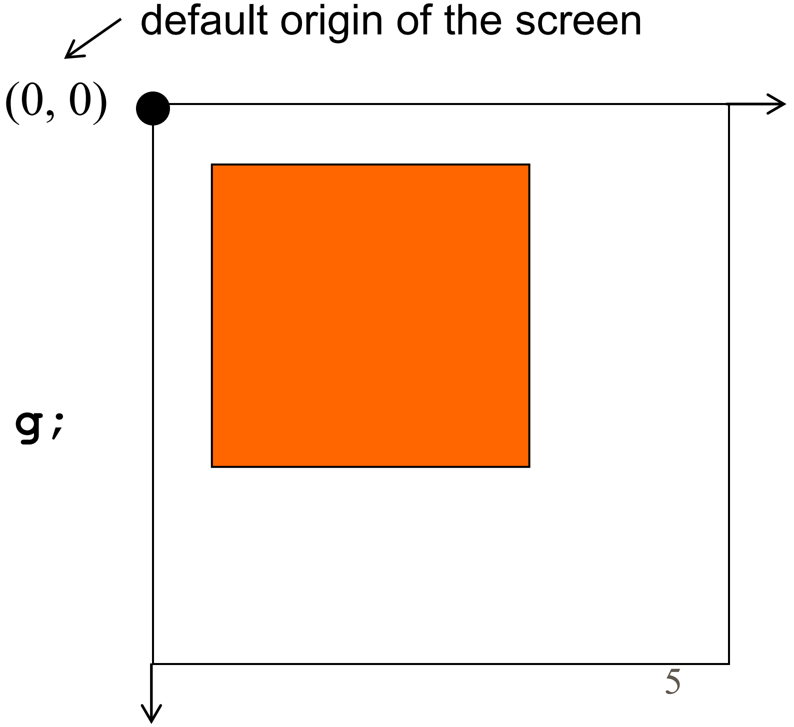
- Rotation
  - Is about rotating the drawing space around its origin by an angle

- Scaling
  - Is about increasing/decreasing the objects' appearances by stretching/shrinking the drawing space

Translation

- Translation gives us another way of drawing at a new location
- Before that, let's examine drawing directly in the default drawing space:

```
paintComponent(Graphics g){
   //create an instance of
     Graphics2D by type casting
   Graphics2D g2 = (Graphics2D) g;

   g2.fillRect(10, 10, 50, 50);
```

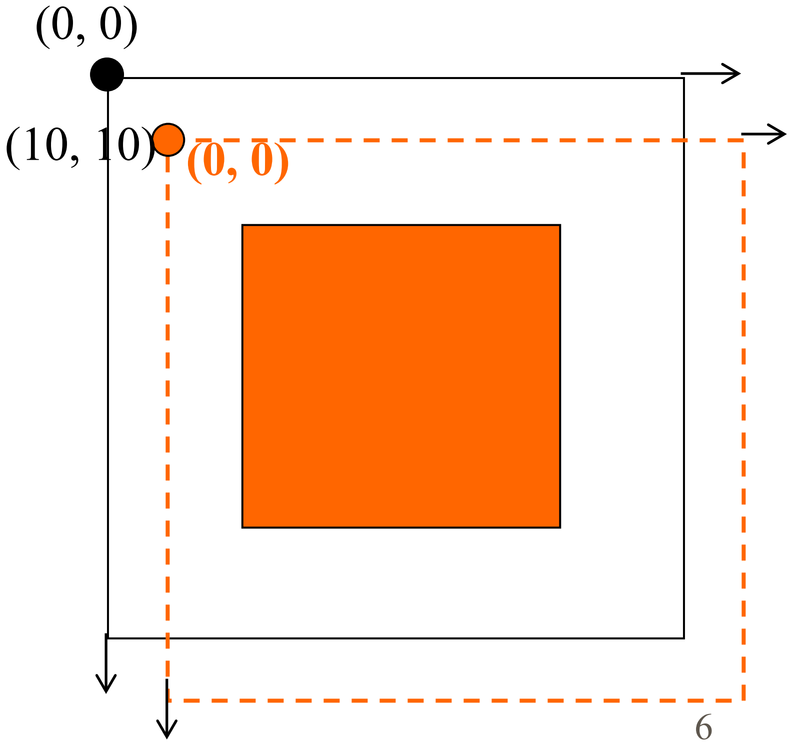


default origin of the screen

Translation

- Now call `translate()`, any drawing done thereafter will treat the *location translated to* as the new origin (0, 0)

```
//To draw the rect using the same
  coordinates after translation
g2.translate( 10, 10 );
g2.fillRect(10, 10, 50, 50);
```

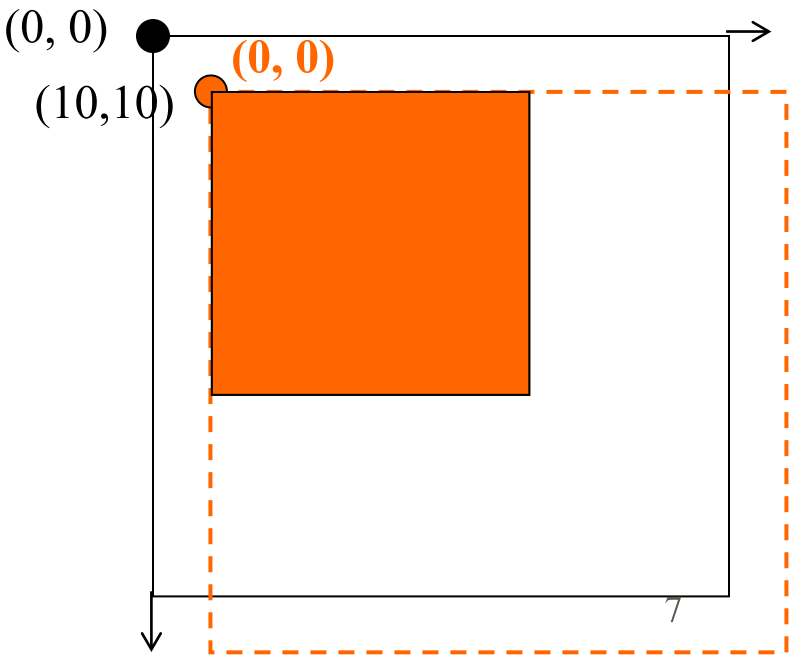


Translation (2)

- What if we want to, after the call to `translate(10, 10)`, draw at the same location as before the translation?

```
//To draw the rect at the same
//location as before
g2.translate( 10, 10 );
g2.fillRect(0, 0, 50, 50);
```

- Making use of new (0, 0) to specify the location of shapes is what we should do



Rotation

- Much like Translation, *rotation* rotates the drawing space, so that we can draw at different orientation

- Most of the time, you'll want to use rotation in conjunction with translation
  - Otherwise `rotate()` rotates around the top-left corner of the screen *– the default origin*
    - This won't be what you want in most cases!!

Rotation

- Let’s use translation before rotation

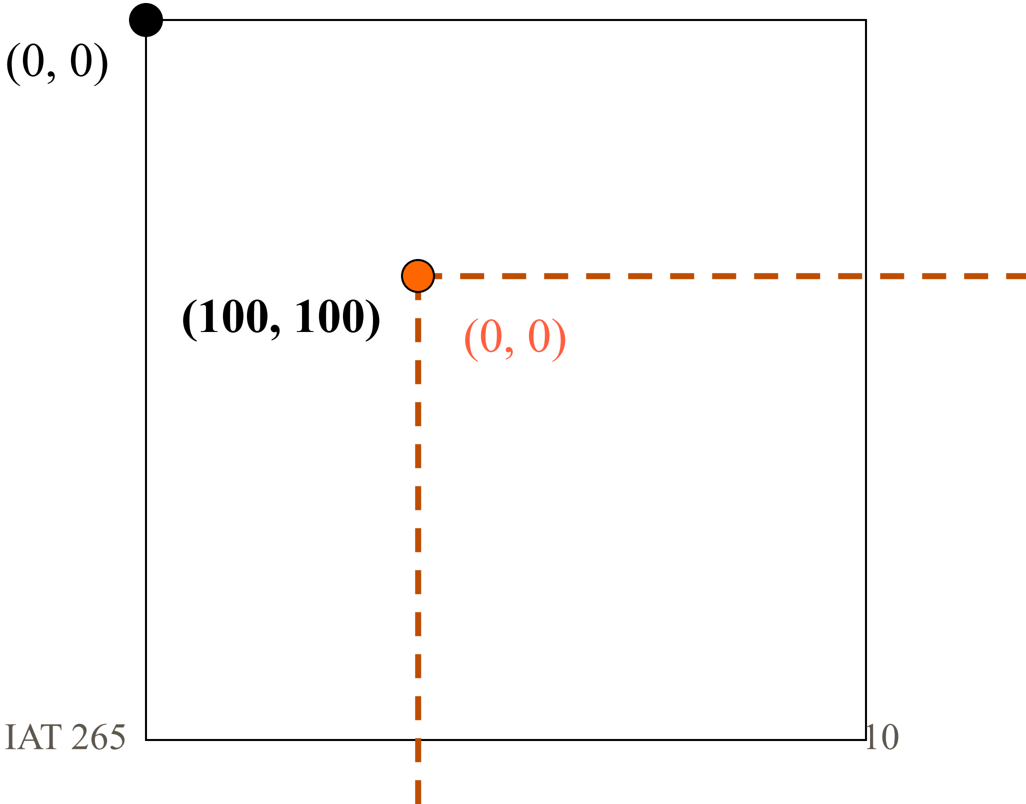
- Where should we translate to?
  - The point **around** which we want to rotate
    - So let’s try and rotate around the center of the square

    - This means shifting the origin to somewhere on the screen, and drawing the square around it

Rotation

- Let’s start with setting our rotation point:

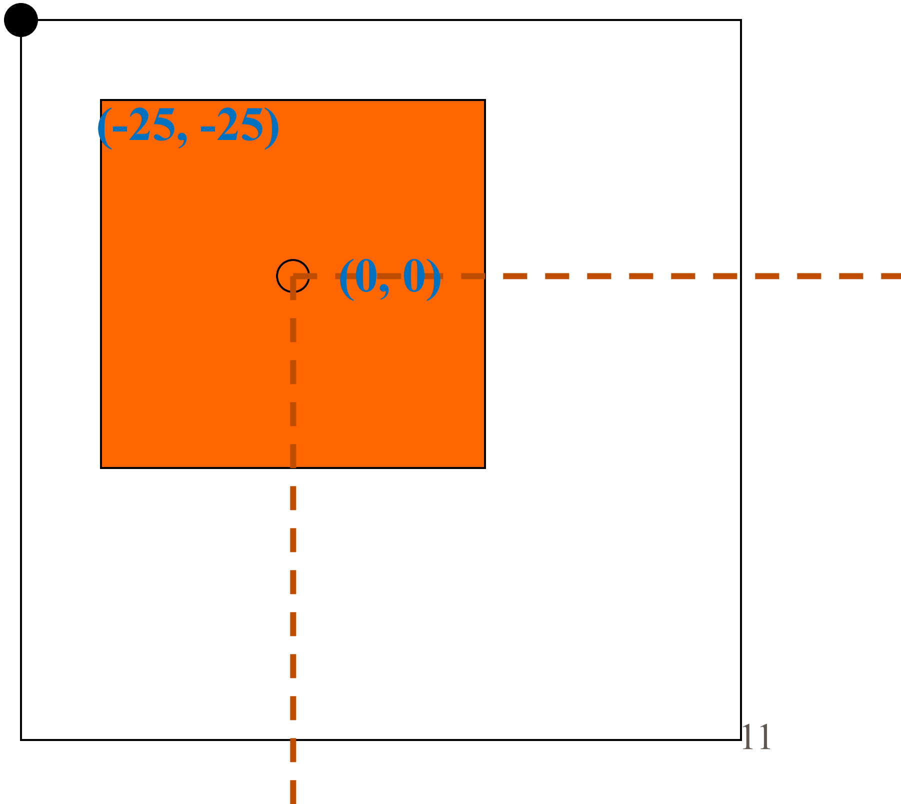
```
g2.translate(100, 100);
```



Rotation

- To draw a square with the new origin being its center, we need:

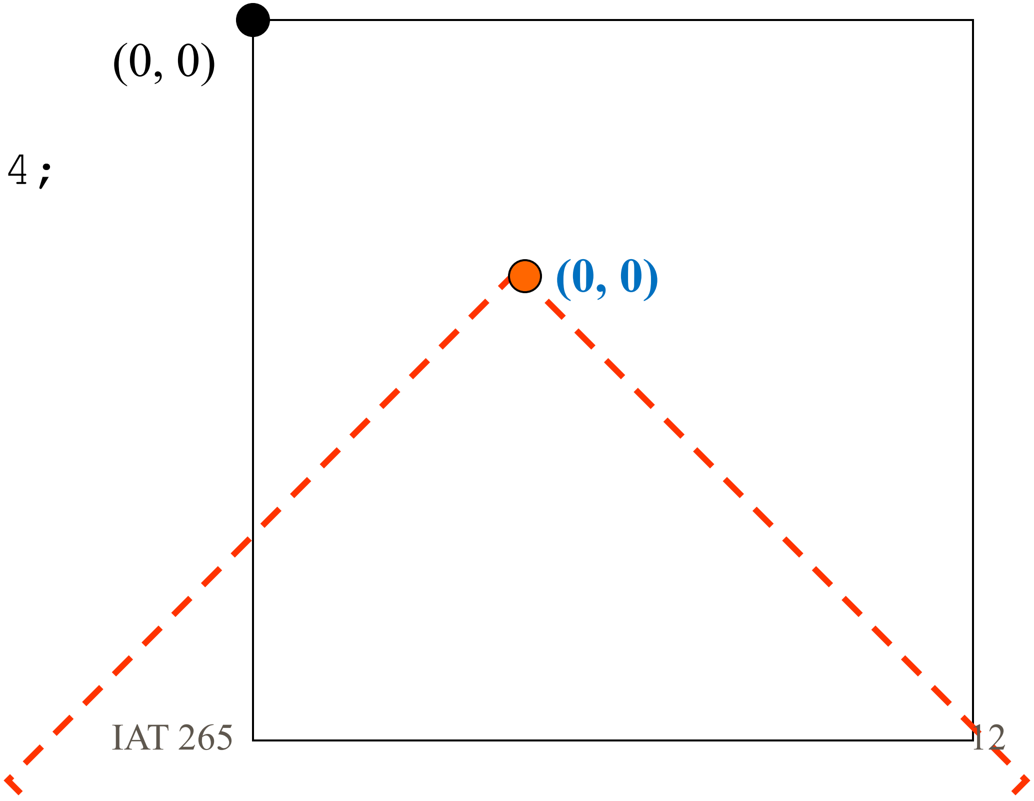
```
g2.fillRect(-25, -25, 50, 50);
```



Rotation

- Now let's rotate the drawing space after translate:

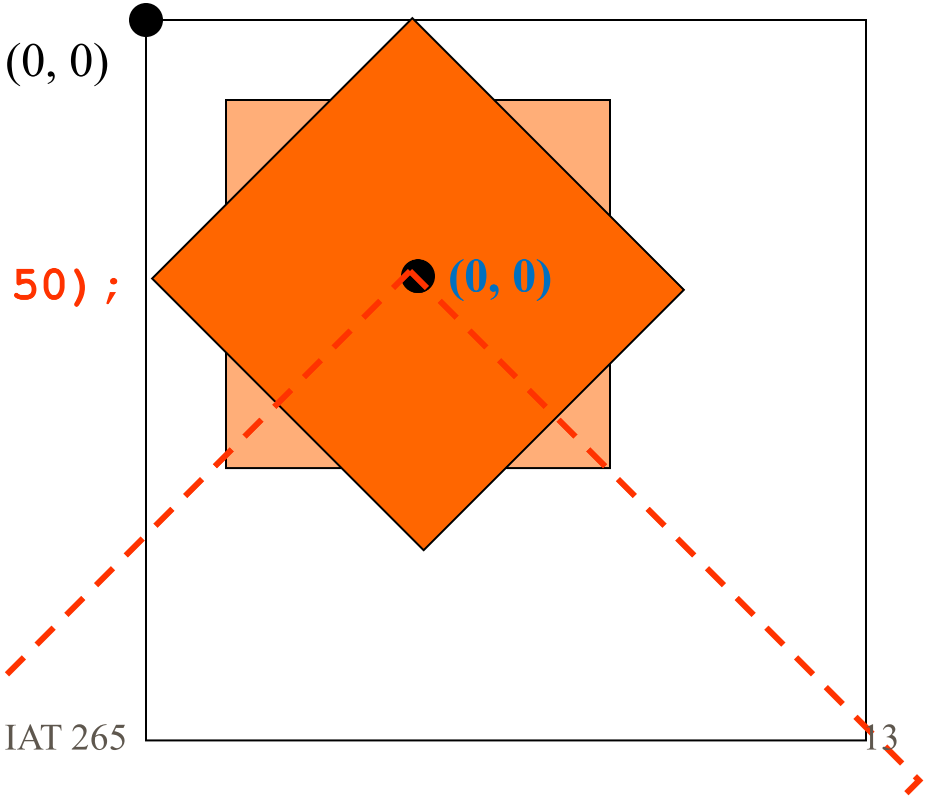
```
g2.translate(100, 100);

double angle = Math.PI/4;
g2.rotate(angle);
```



Rotation

- Then we draw the same square as before, it will have the same center point but rotated

```
g2.translate(100, 100);

double angle = Math.PI/4;
g2.rotate(angle);
g2.fillRect( -25, -25, 50, 50);
```



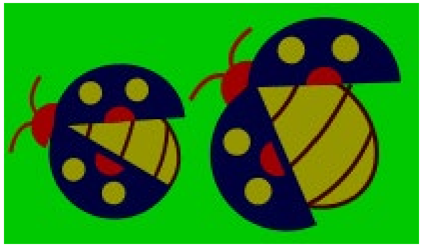
Try and see the effect …

```
double angle = 0;

public void paintComponent(Graphics g) {
    super.paintComponent(g);                //Call JPanel's method to clear
                                              the background
    Graphics2D g2 = (Graphics2D) g;
    g2.translate(200, 200);
    g2.rotate(angle);
    g2.setColor(new Color(255, 128, 0));
    g2.fillRect (-75, -75, 150, 150);
 }

public void actionPerformed(ActionEvent e) {
     angle += 0.01;
     repaint();
}
```

Rotation of Figures

- Figure consists of complex shapes, which you may need to rotate it as a whole or part of it (micro-animations)
- For rotating the whole figure:
  - Translate to the rotation point, and then draw each of its body parts w.r.t. this new (0,0)
    - Snowman: rocking vs swinging from thread

- For rotating a body part, translate to where the joint point is supposed to be, draw the part w.r.t. it
  - E.g. make wings flap around its joint point with the body



Summary on Translation and Rotation

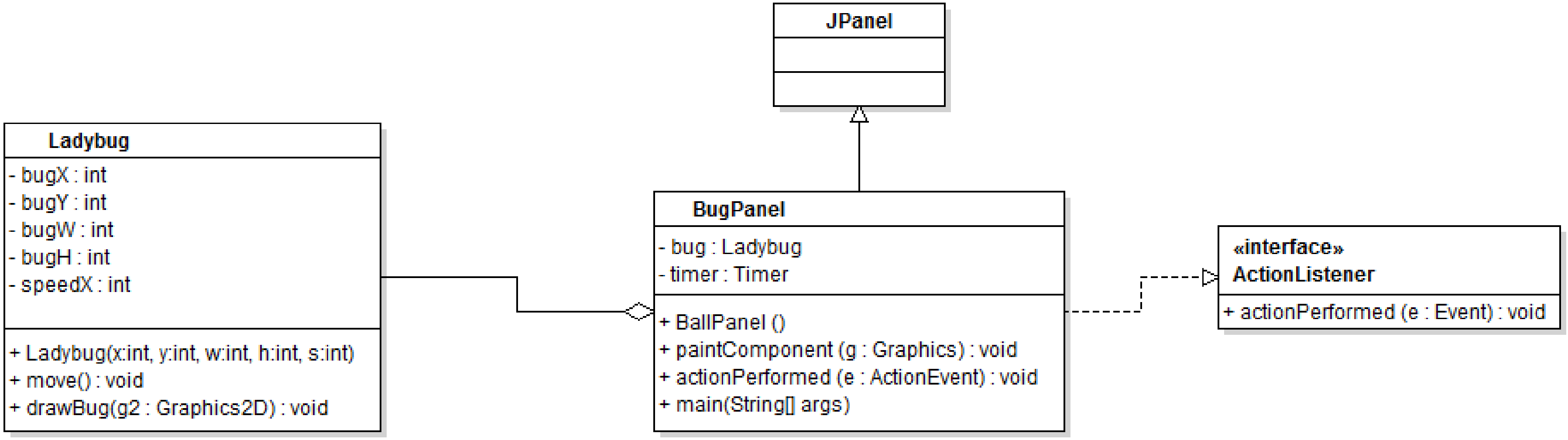
- Translation moves the drawing space to a specified location
  - The location that was translated to represents the new origin (0, 0)

- Rotation is about rotating the drawing space around its (desirably translated) origin by an angle

- For most cases, you should do rotation in conjunction with translation, to make your shape or figure rotate around a point that is desirable to the visual effect

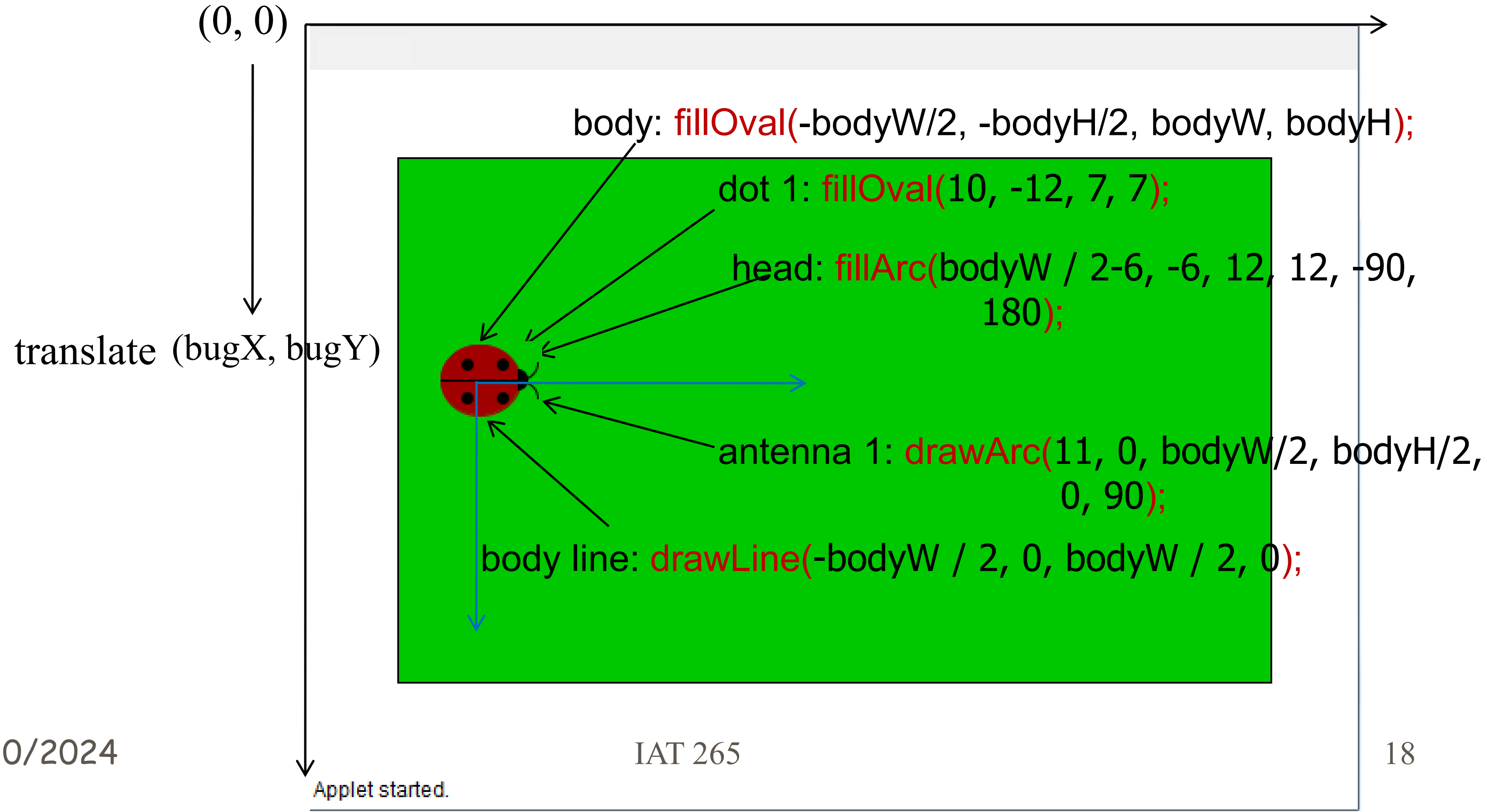
Case study: translate & rotate a Ladybug object

- We'll design and animate a Ladybug object by way of *translate* and *rotate* methods of *Graphics2D*


JPanel
Ladybug
- bugX : int
- bugY : int
- bugW : int
- bugH : int
- speedX : int
+ Ladybug(x:int, y:int, w:int, h:int, s:int)
+ move() : void
+ drawBug(g2 : Graphics2D) : void
BugPanel
- bug : Ladybug
- timer : Timer
+ BallPanel ()
+ paintComponent (g : Graphics) : void
+ actionPerformed (e : ActionEvent) : void
+ main(String[] args)
«interface»
ActionListener
+ actionPerformed (e : Event) : void

Draw the ladybug with translate

- Step 1: *translate to (bugX, bugY)* which will be used as the center of the bug's body, and then draw all shapes around the new (0, 0)



body: fillOval(-bodyW/2, -bodyH/2, bodyW, bodyH);

dot 1: fillOval(10, -12, 7, 7);

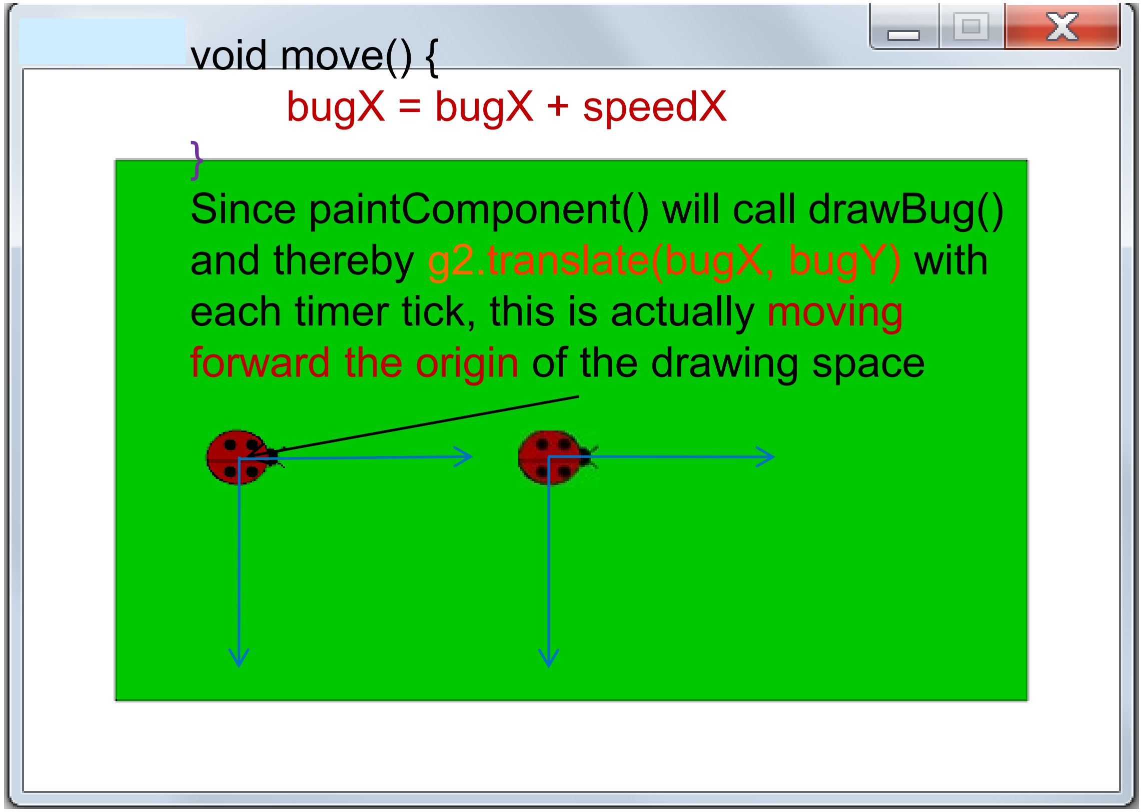
head: fillArc(bodyW / 2-6, -6, 12, 12, -90, 180);

antenna 1: drawArc(11, 0, bodyW/2, bodyH/2, 0, 90);

body line: drawLine(-bodyW / 2, 0, bodyW / 2, 0);

Move the ladybug

- Step 2: create a *move()* method to move the bug by *speedX*

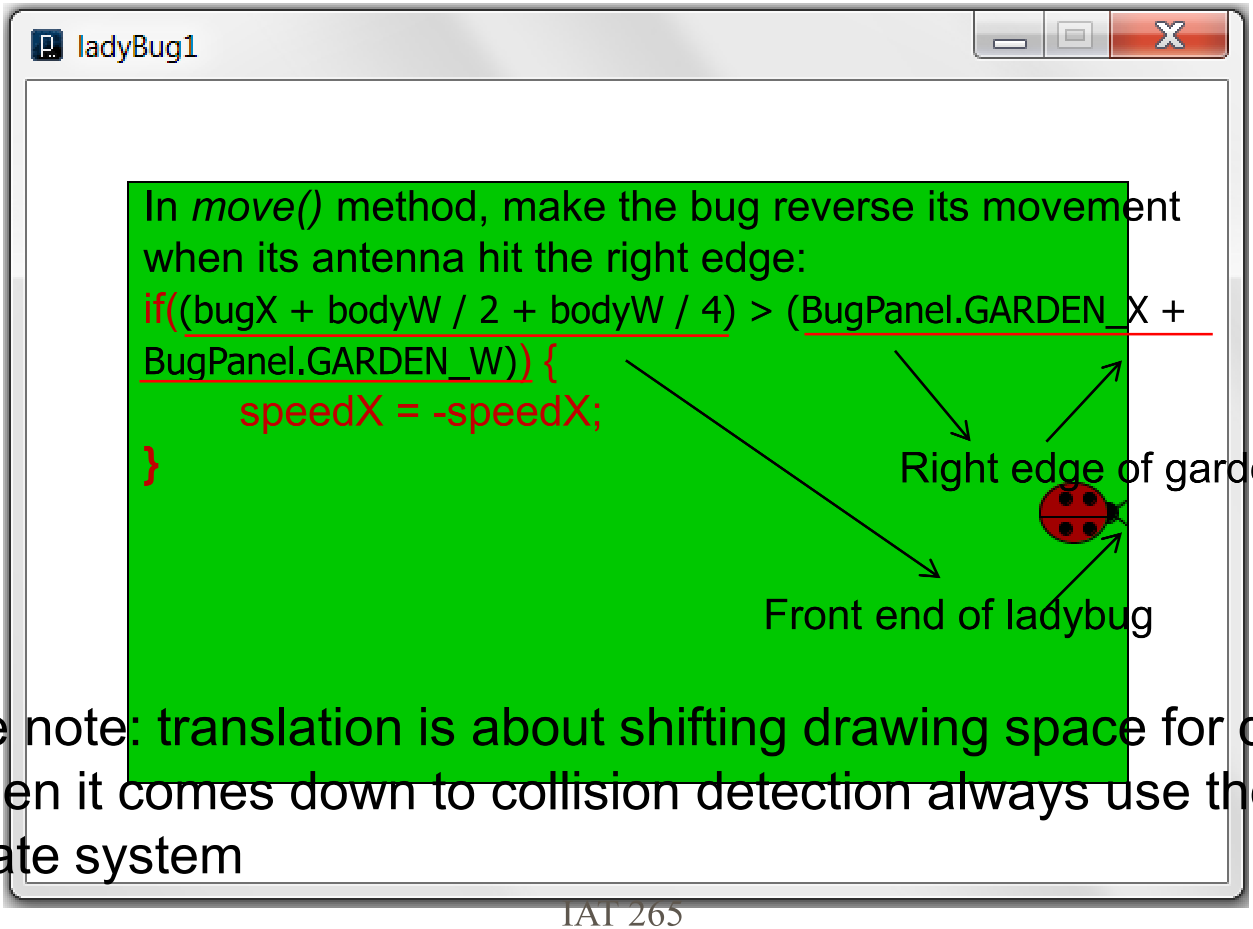


```
void move() {
        bugX = bugX + speedX
}
```

Since paintComponent() will call drawBug() and thereby g2.translate(bugX, bugY) with each timer tick, this is actually moving forward the origin of the drawing space

Right Edge Detection

- Step 3: Right edge detection*



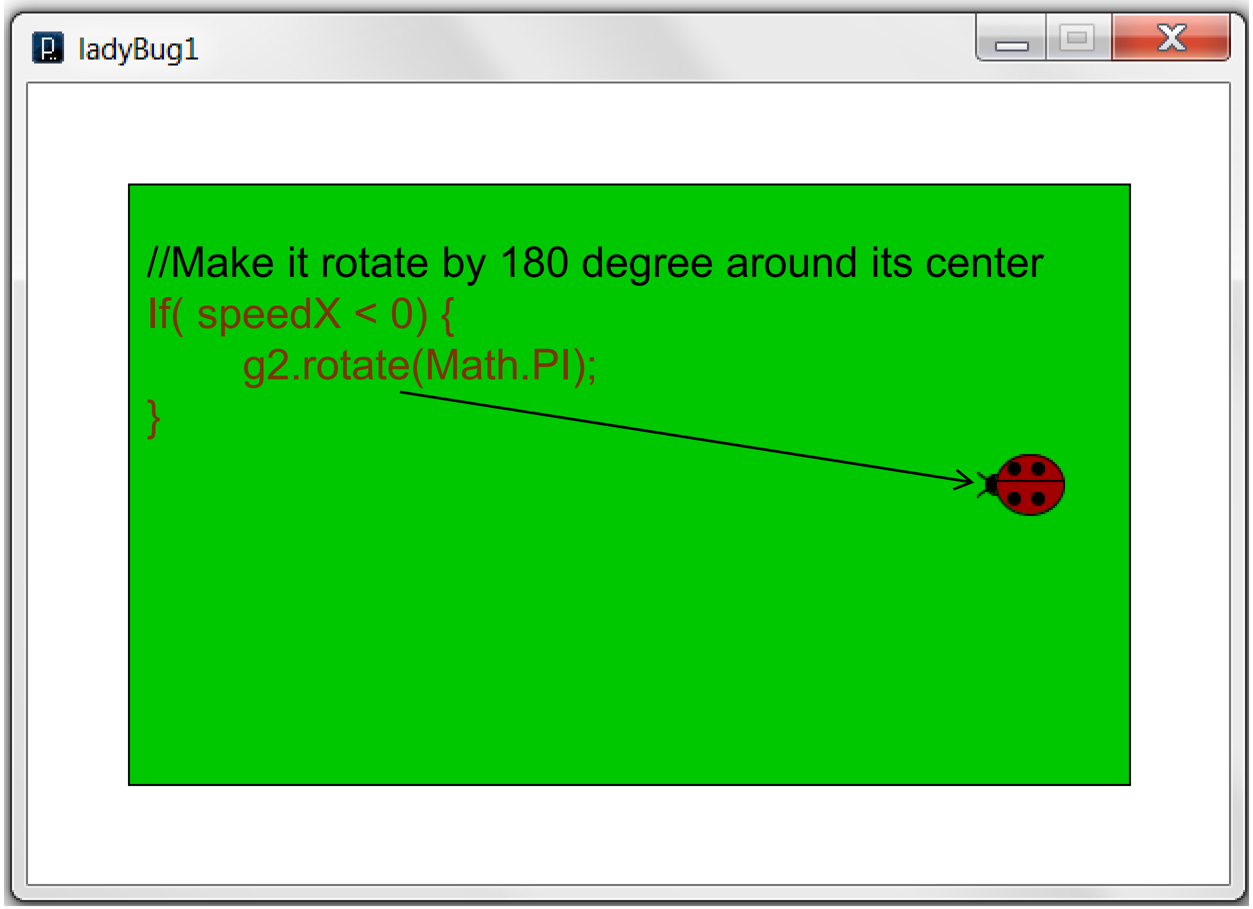
In *move()* method, make the bug reverse its movement when its antenna hit the right edge:

```
if((bugX + bodyW / 2 + bodyW / 4) > (BugPanel.GARDEN_X +
BugPanel.GARDEN_W)) {
        speedX = -speedX;
}
```

* Please note: translation is about shifting drawing space for display only, when it comes down to collision detection always use the default coordinate system

Rotate the ladybug when moving backward

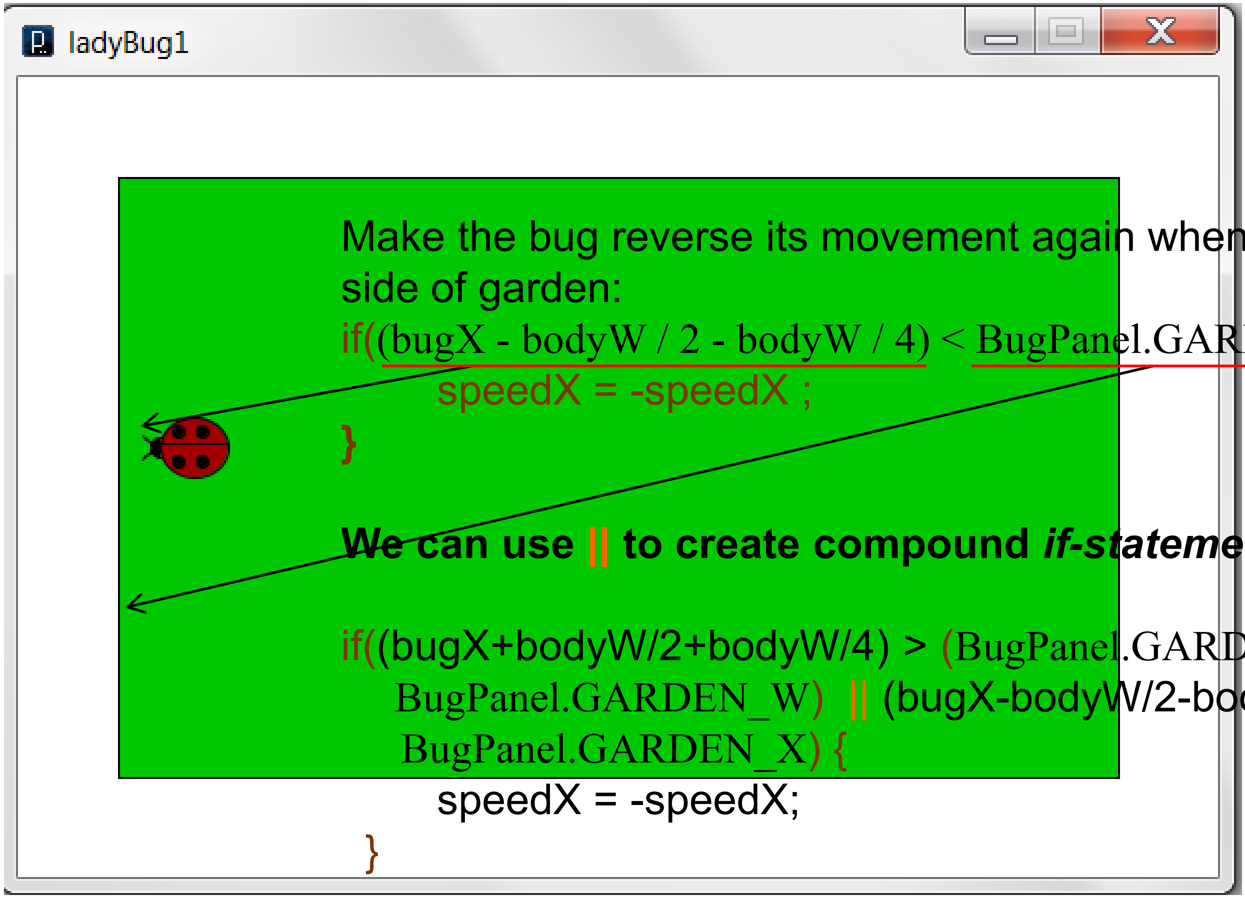
- Step 4: make the bug rotate when reversing



```
//Make it rotate by 180 degree around its center
If( speedX < 0) {
        g2.rotate(Math.PI);
}
```

Left Edge Detection

- Step 5: Left Edge detection



Make the bug reverse its movement again when hitting left side of garden:

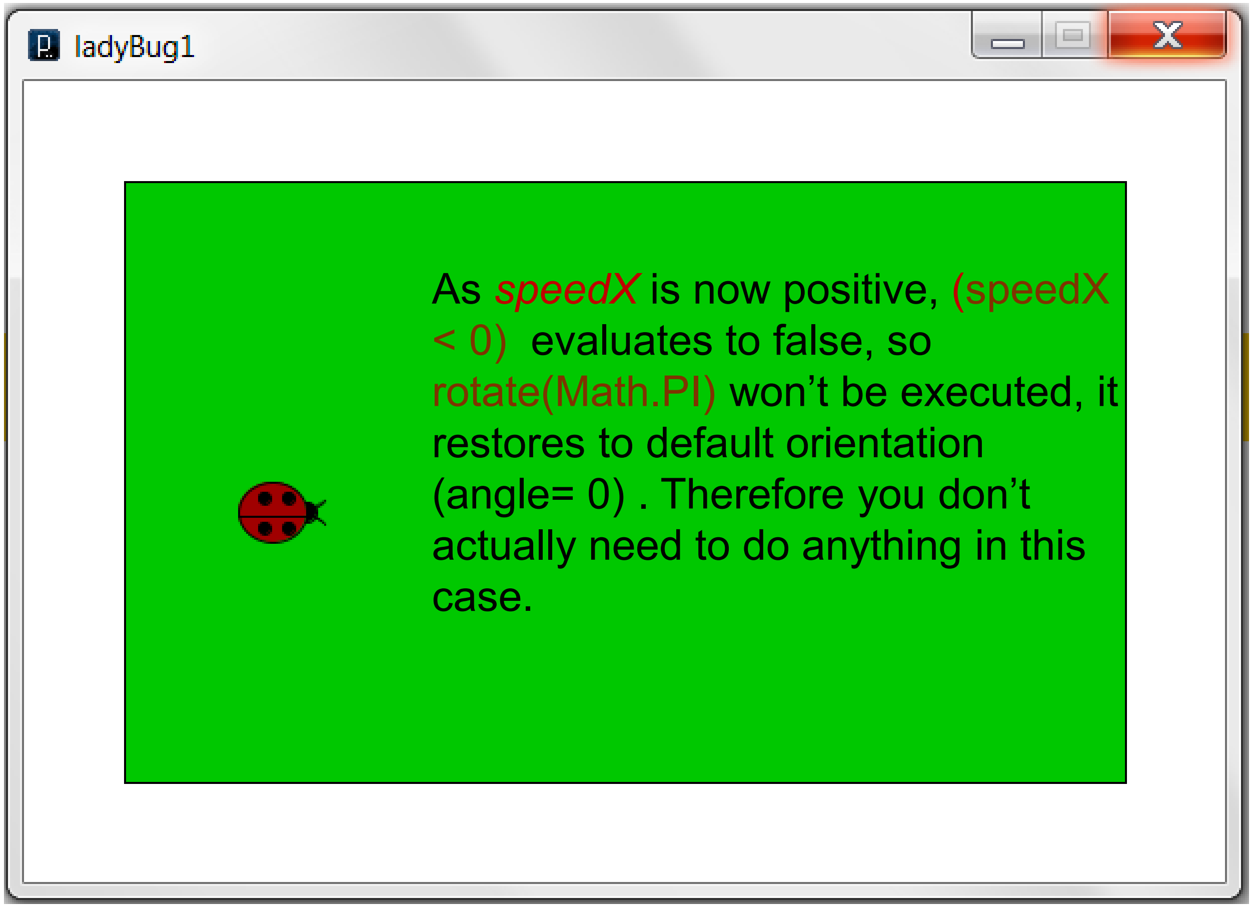
```
if((bugX - bodyW / 2 - bodyW / 4) < BugPanel.GARDEN_X)) {
        speedX = -speedX ;
}
```

We can use || to create compound *if-statement:*

```
if((bugX+bodyW/2+bodyW/4) > (BugPanel.GARDEN_X +
    BugPanel.GARDEN_W)  || (bugX-bodyW/2-bodyW/4) <
    BugPanel.GARDEN_X) {
        speedX = -speedX;
 }
```

Last step : rotate again?

- After it hits the left edge of the garden



As *speedX* is now positive, (speedX < 0) evaluates to false, so rotate(Math.PI) won't be executed, it restores to default orientation (angle= 0) . Therefore you don't actually need to do anything in this case.

Scale

- Graphics2D object has one method for scaling that is defined as follows

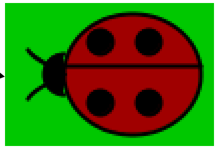
- *scale(double sx, double sy)*
  - sx: percentage to scale along x dimension
  - sy: percentage to scale along y dimension
  - e.g. *g2.scale(2.0, 0.5)* scales the objects up to 200% along x, and down to 50% along y respectively

- This is done by scaling the drawing space by the factors – may cause tricky issue for objects' collision detections (which involves the original coordinate system only)

Scale the ladybug

- Scale the bug by a scale factor of 2.0

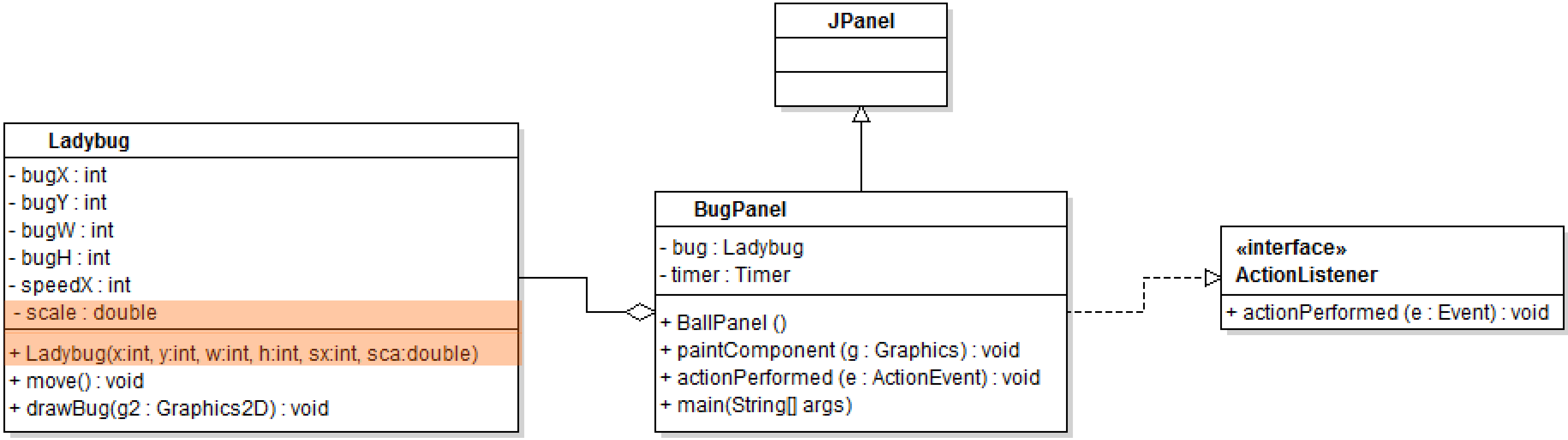
```
double scale = 2.0;
//Scale the bug's view by scale
g2.scale(scale, scale);
```



- Two possibilities:
  - Scale both the bug and the garden with the same factor (no boundary detection issue in this case)
  - Scale the bug only NOT the garden (need to adjust boundary detection *conditionals* with the *scale factor*)
    - In doing all our assignments and projects, we would like to apply scale to the objects ONLY not including the background

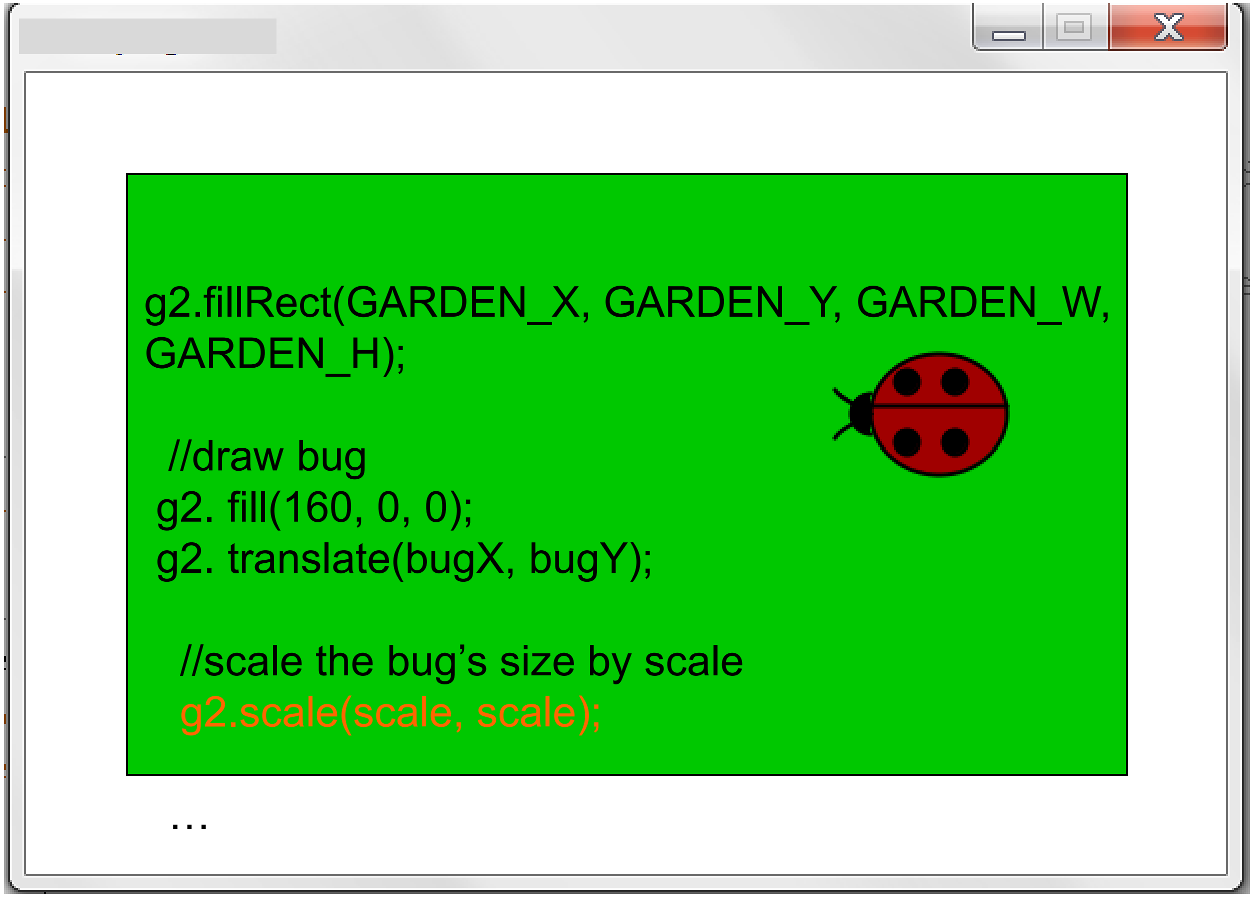
Update the Design to include Attribute for Scale

- Add to Ladybug class a field of *scale*, and a *parameter* to its constructor for initialization


JPanel
Ladybug
- bugX : int
- bugY : int
- bugW : int
- bugH : int
- speedX : int
- scale : double
+ Ladybug(x:int, y:int, w:int, h:int, sx:int, sca:double)
+ move() : void
+ drawBug(g2 : Graphics2D) : void
BugPanel
- bug : Ladybug
- timer : Timer
+ BallPanel ()
+ paintComponent (g : Graphics) : void
+ actionPerformed (e : ActionEvent) : void
+ main(String[] args)
«interface»
ActionListener
+ actionPerformed (e : Event) : void

Use the *scale* field to scale the bug only

- Call scale() after translating the bug

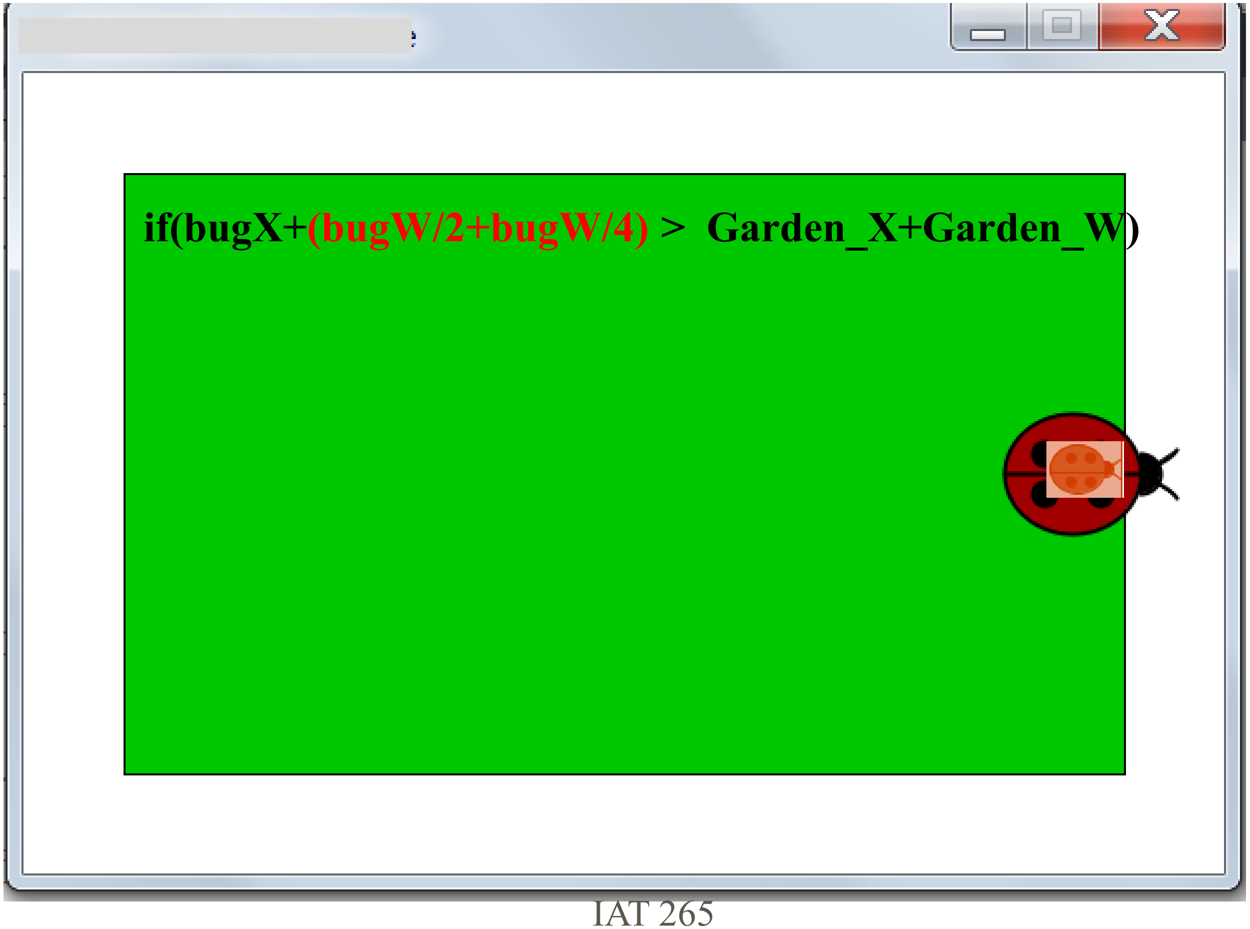


```
g2.fillRect(GARDEN_X, GARDEN_Y, GARDEN_W,
GARDEN_H);

 //draw bug
g2. fill(160, 0, 0);
g2. translate(bugX, bugY);

  //scale the bug’s size by scale
  g2.scale(scale, scale);
```

…

Tricky issue with boundary detection

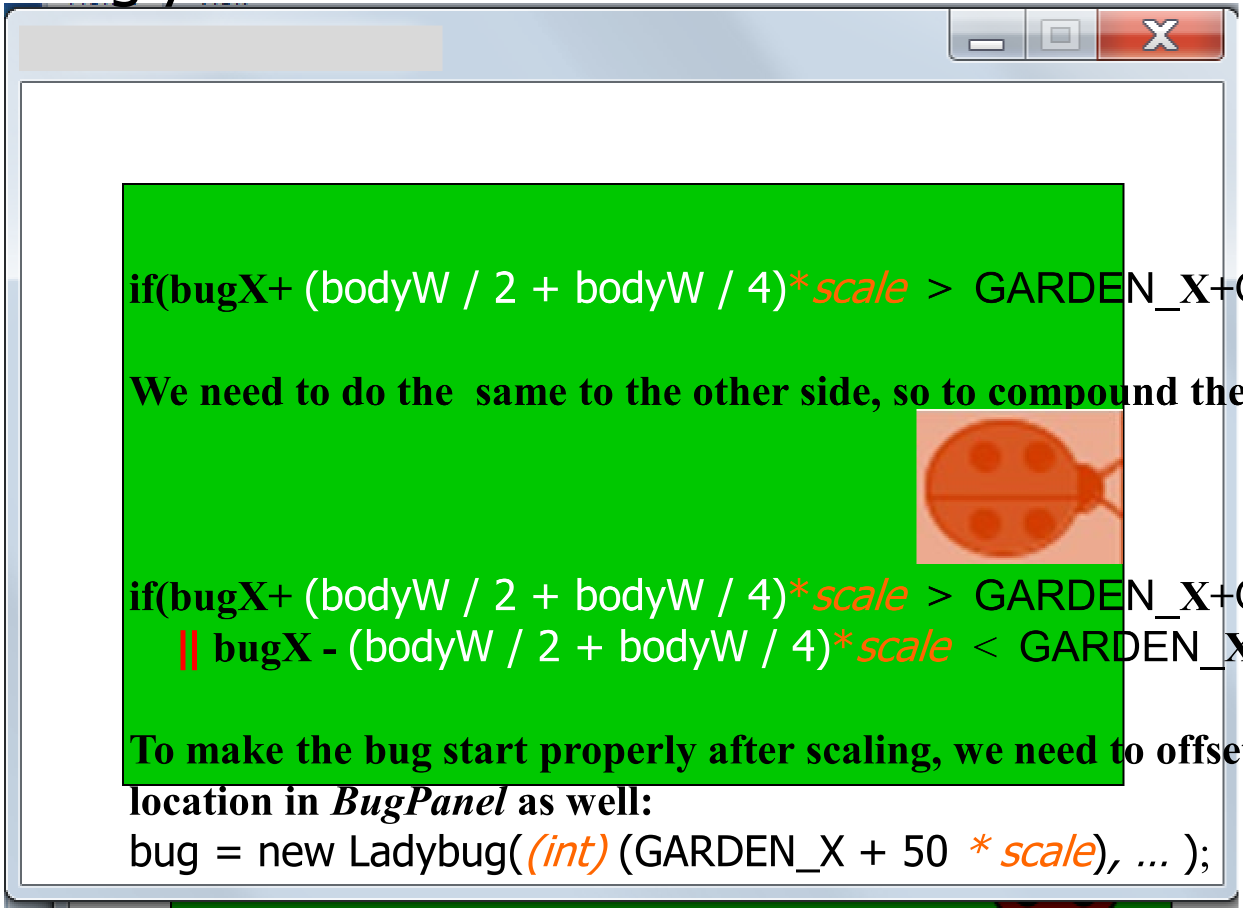
- Appearance is now enlarged but boundary detection is still based on the original size



```
if(bugX+(bugW/2+bugW/4) >  Garden_X+Garden_W)
```

To Fix …

- Multiply the bug's total width with the scale factor accordingly



```
if(bugX+ (bodyW / 2 + bodyW / 4)*scale  >  GARDEN_X+GARDEN_W)
```

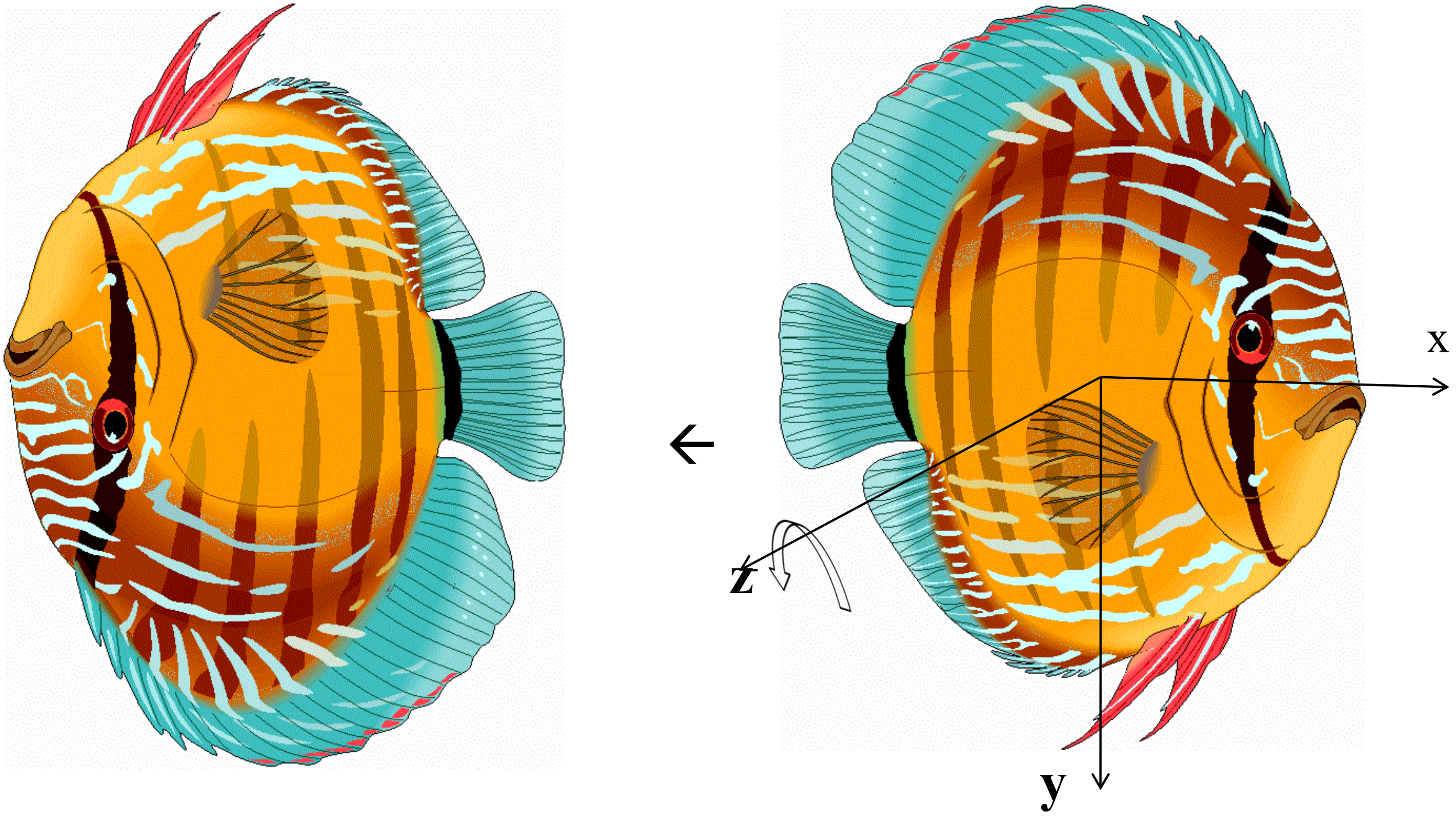
We need to do the same to the other side, so to compound them, we get:

```
if(bugX+ (bodyW / 2 + bodyW / 4)*scale  >  GARDEN_X+GARDEN_W
   || bugX - (bodyW / 2 + bodyW / 4)*scale  <  GARDEN_X )
```

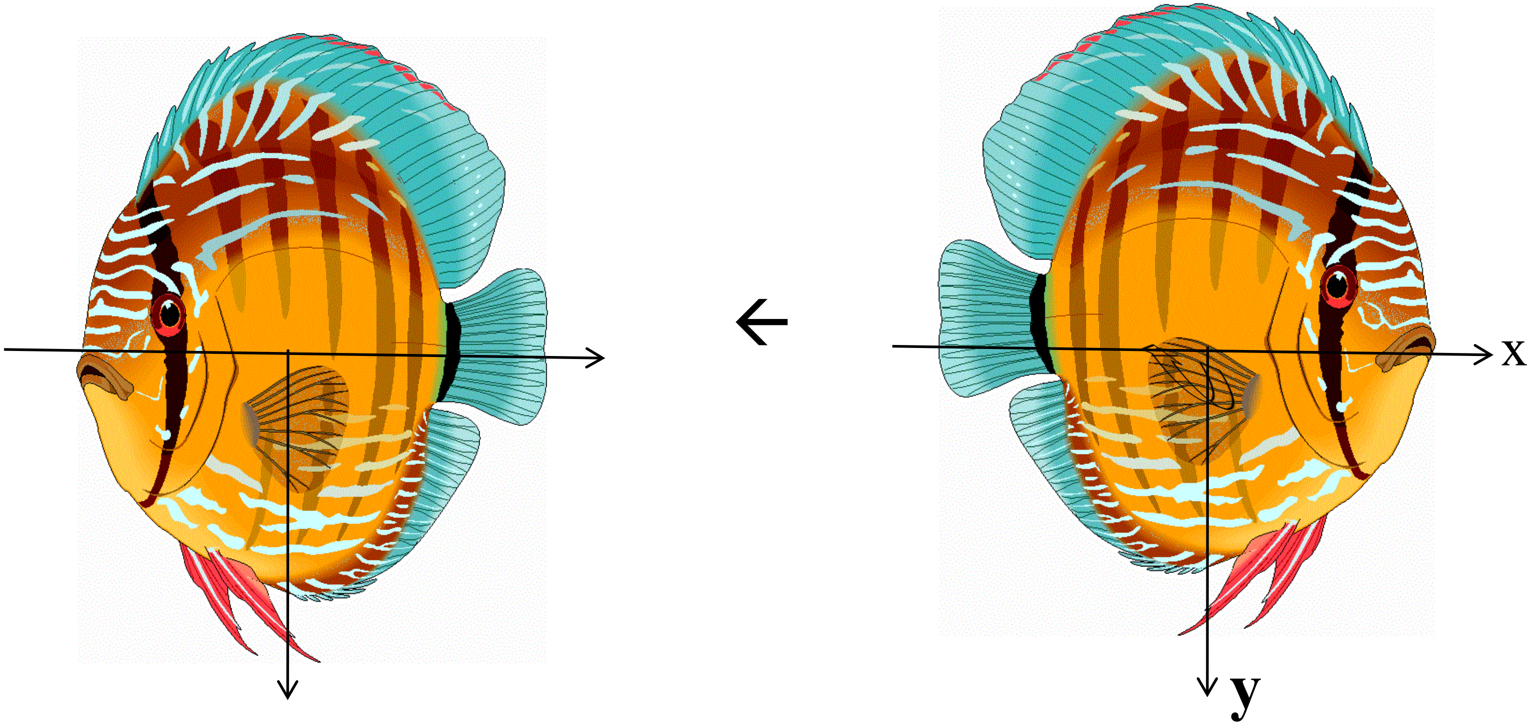
To make the bug start properly after scaling, we need to offset its initial location in *BugPanel* as well:

```
bug = new Ladybug((int) (GARDEN_X + 50  * scale), … );
```

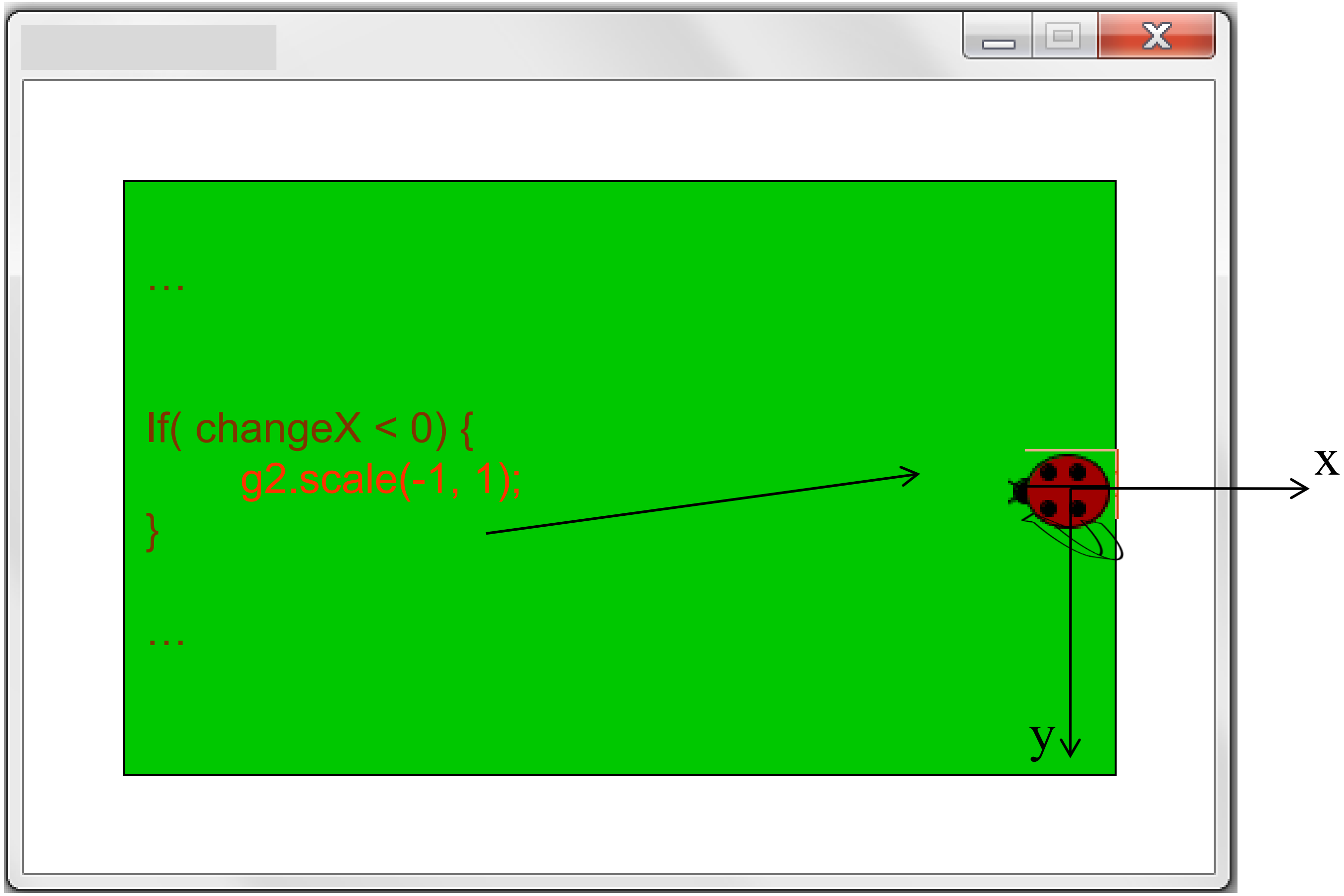
How to deal with the upside-down issue after rotation?



To resolve: flip the drawing space along x scale(-1, 1)



scale(-1, 1) to flip the drawing space along x

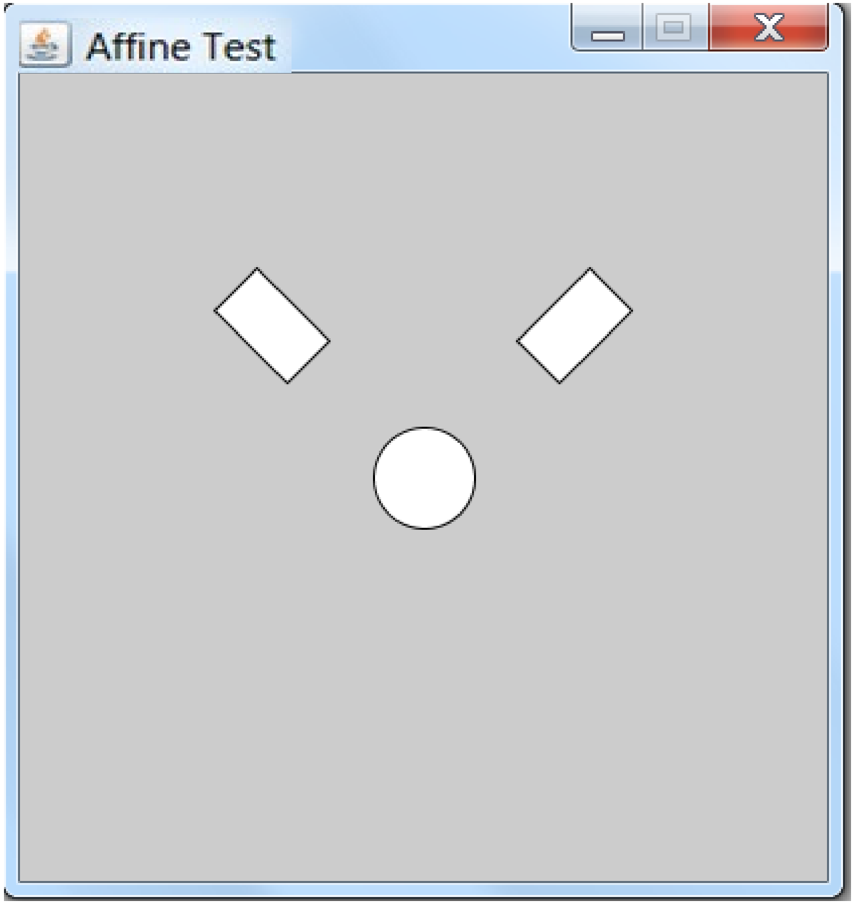


```
…

If( changeX < 0) {
        g2.scale(-1, 1);
}

…
```

Issues with Multiple Transformations

- Transformations (including translations, rotations & scaling) are cumulative
  - This means when multiple transformations are done for different shapes (e.g. animal and food), the transformation done for one shape would affect those after it, which would result in unexpected results

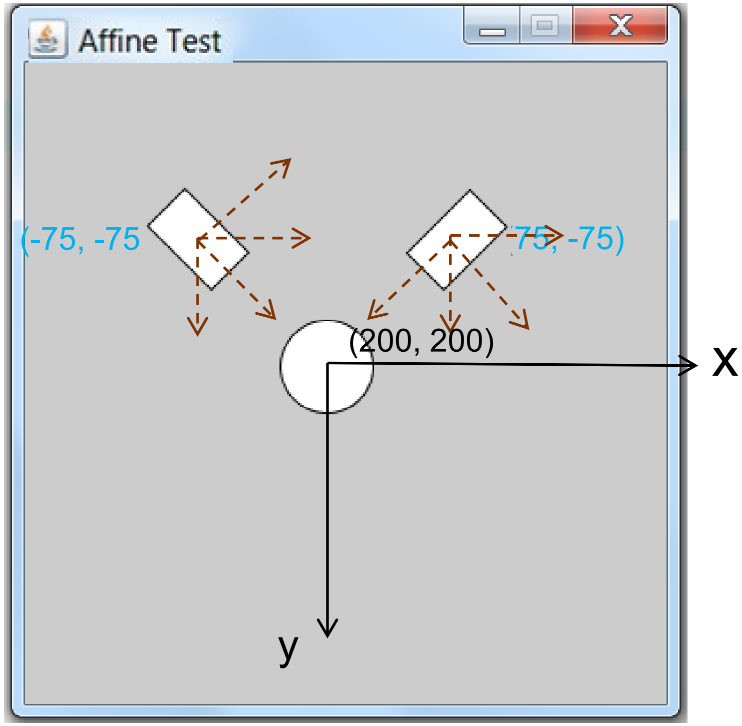
Ex: How to create a figure like this?



Ex: How to create a figure like this?

- Just do the followings?

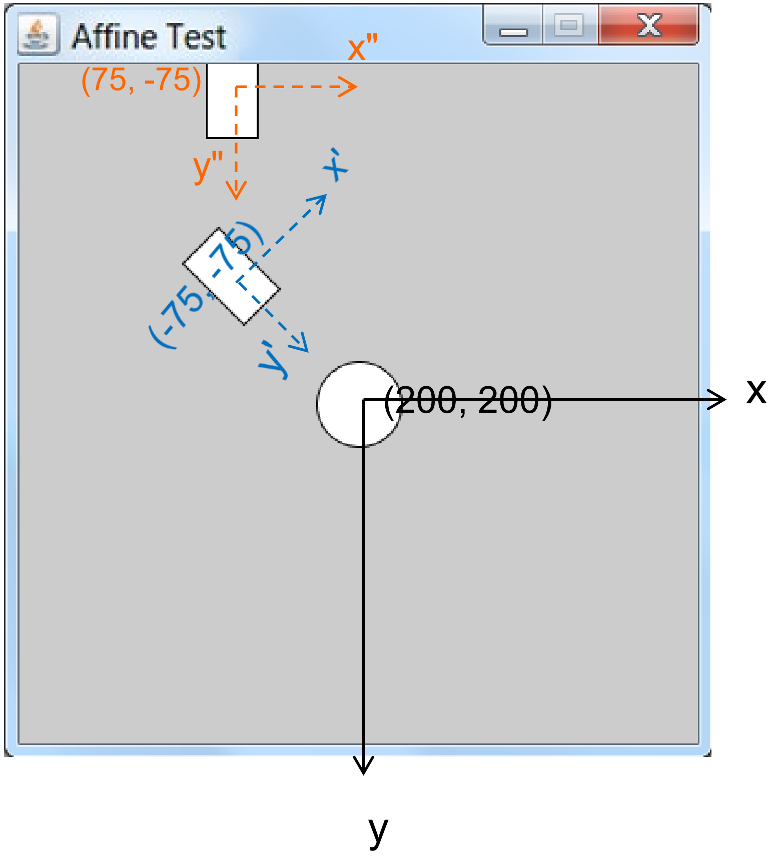
```
g2.translate(200, 200);
g2.fillOval(-25, -25, 50, 50);

g2.translate(-75, -75);
g2.rotate(-Math.PI/4);
g2.fillRect(-15, -25, 30, 50);


g2.translate(75, -75);
g2.rotate(Math.PI/4);
g2.fillRect(-15, -25, 30, 50);
```



However the result is …

- g2.translate(200,200);
- g2.translate(-75,-75);
  g2.rotate(-Math.PI/4);
- g2.translate(75,-75);
  g2.rotate(Math.PI/4);

- This is because transformation is cumulative → (x″~y″) is based on (x′~y′), rather than (x~y)



Solution: save & restore *transform attributes* for transformations

- In Java, all transformations are associated with *transform attributes* (regarding its location, orientation, scaling etc.), can be stored in an instance of the *AffineTransform* class

- What we need here is to:
  1. Save the *transform attributes* before we do transformations for a shape
  2. Do transformations for the shape and then its rendering
  3. Restore the *transform attributes* to what it once was after the rendering

- We can then transform and render another shape, so that it will start from the same *transform attributes*

Implementation with AffineTransform

- Pseudocode Programming Process (PPP):

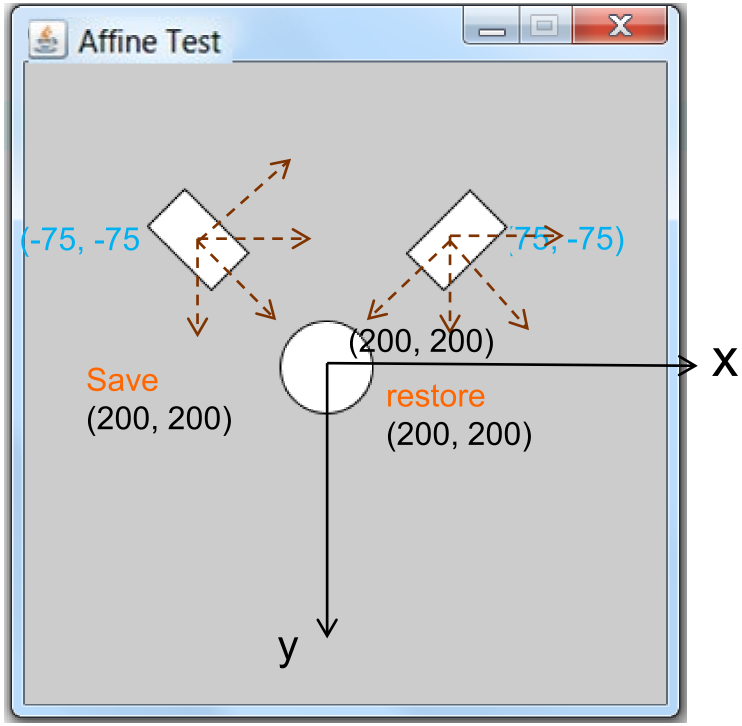
1. We call *Graphics2D*'s *getTransform()* method to retrieve the current *transform attributes* and save it into an *AffineTransform instance* before we do transformations for a shape

2. When finished transforming and drawing for the shape, we call *Graphics2D*'s *setTransform(AffineTransform tx)* and set it back to the saved *AffineTransform instance* to restore the previous state of the *transform attributes*

3. Do so for each of the rest shapes that needs its own transformation - except for the last shape involved

Implement per the PPP

```
g2.translate(200, 200);
g2.fillOval(-25, -25, 50, 50);

AffineTransform transform =
g2.getTransform();    //save(x~y)
g2.translate(-75, -75);
g2.rotate(-Math.PI/4);
g2.fillRect(-15, -25, 30, 50);
g2.setTransform(transform); //restore(x~y)


g2.translate(75, -75);
g2.rotate(Math.PI/4);
g2.fillRect(-15, -25, 30, 50);
```



Summary on Transformations

- Each time you *translate*, *rotate* or *scale*, it's referred to as a *transformation*

- When different transformations for multiple shapes are involved, you should call *getTransform()* to save the *transform attributes* before each transformation, render your shape, and then call *setTransform(AffineTransform tx)* to restore the previous *transform attributes*
  - Except for the last one. But to make thing easier, even if you include it, it won't hurt much overhead-wise

- Make sure that each call to *getTransform()* has a matching call to *setTransform(AffineTransform tx)*
  - Nest *getTransform()* and *setTransform(AffineTransform tx)* when it's necessary

How to move toward a Target?

- Need a velocity with direction toward the target location

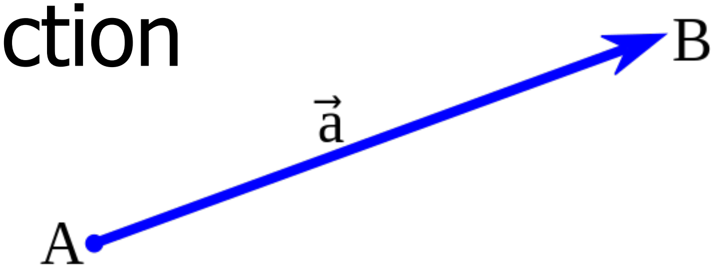
- Better to use vectors to model
  - Location
  - Velocity

Vectors

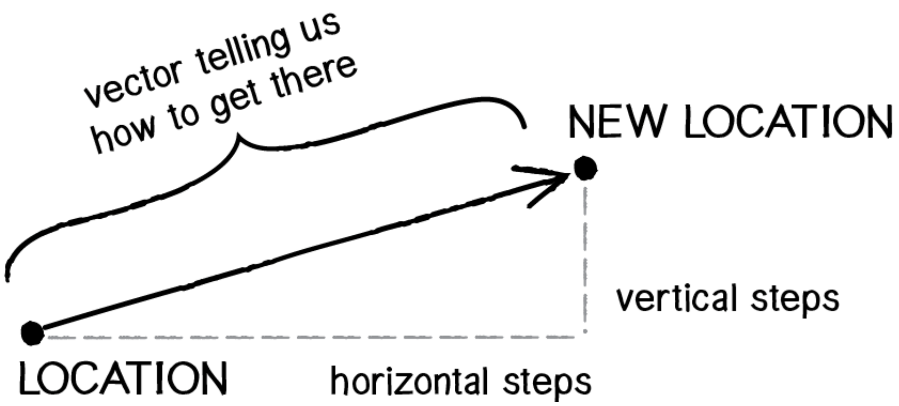
- What is a vector
- Vectors for motion
  - Location
  - Velocity
  - Acceleration

What is a vector?

- Euclidian vector: An entity that has both magnitude and direction

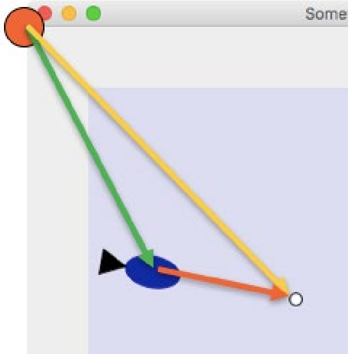


- In motion:



Vector operations

- Add – allow for displacement
  - location+velocity → new_location



- Subtract – allow for targeted motion
  - target_location - source_location → *path* vector – allowing to move toward a target

- Magnitude – the length of the vector
  - Magnitude of the *path* vector is the distance between *source location* and *target*

PVector from Processing core library

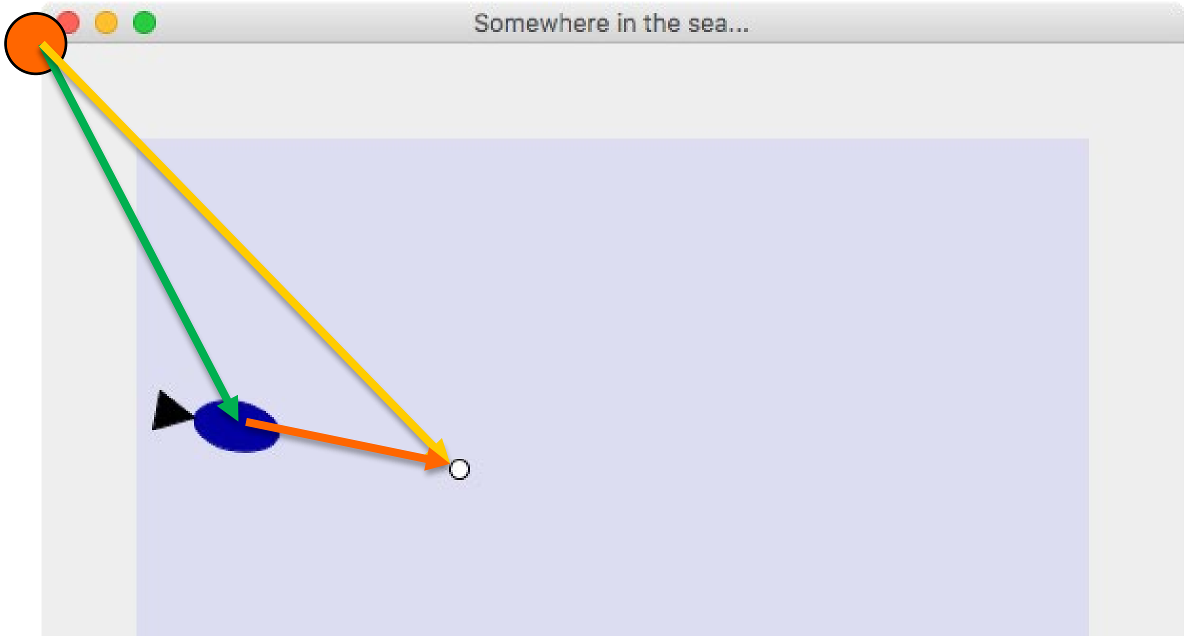
- Constructor:

```
PVector(float x, float y)
```

- Make an object move

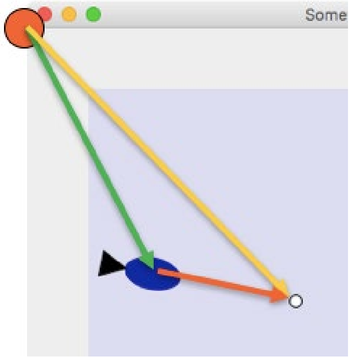
```
PVector location = new PVector(50.0,100.0)
PVector velocity = new PVector(3.0,-1.0)
…
location.add(velocity) //updates location vector
```

Subtraction – path to move



```
PVector path = targetPos.sub(fishPos);                  //WRONG
//because non-static sub(PVector p) returns void

PVector path = PVector.sub(targetPos, fishPos);        //RIGHT
//static sub(PVector p1, PVector p2) returns a PVector object
```

Path to Move: get Angle and Distance



```
float angle = Math.atan2(path.y, path.x)
```

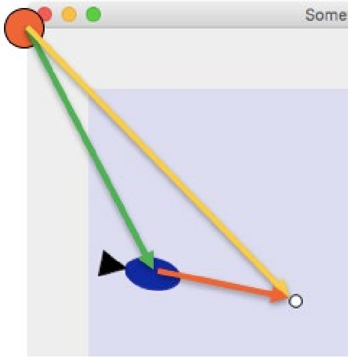
- PVector provides a shortcut:

```
float angle = path.heading();
```

- To calculate distance of *path*:

```
float distance = path.mag();
```

Path to Move: set vel



- *static PVector fromAngle (float angle)*
  - Returns a new *unit vector* (i.e. *length =1*) from the specified angle

```
//create a velocity toward the target
PVector vel = PVector.fromAngle(angle);
vel.mult(2);        //beef it up
```

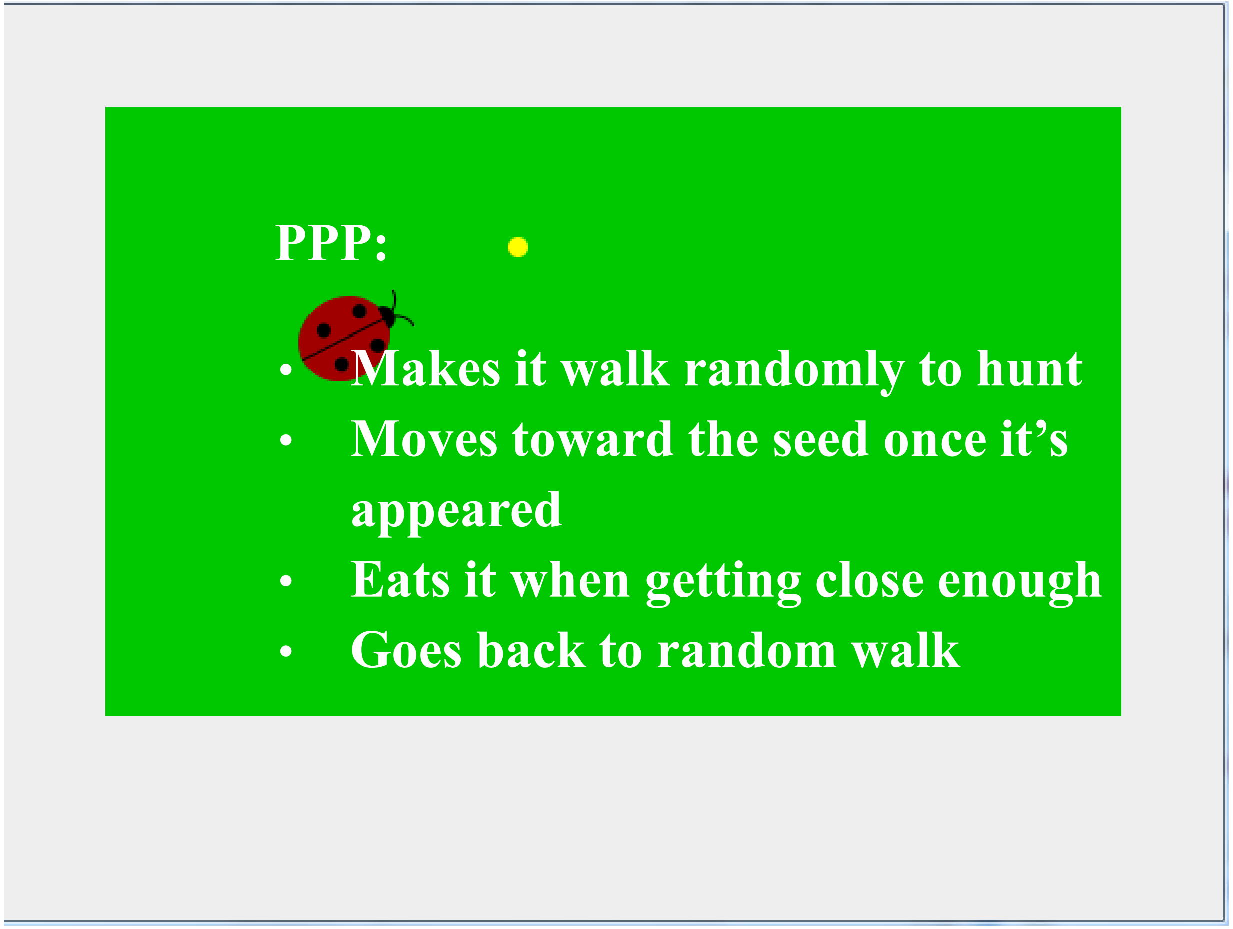
- Or alternatively:

```
PVector vel = PVector.sub(targetPos, fishPos);
vel.limit(2);
```

- Finally move it by vel:

```
location.add(vel);
```

Case study: Ladybug approaches a Seed

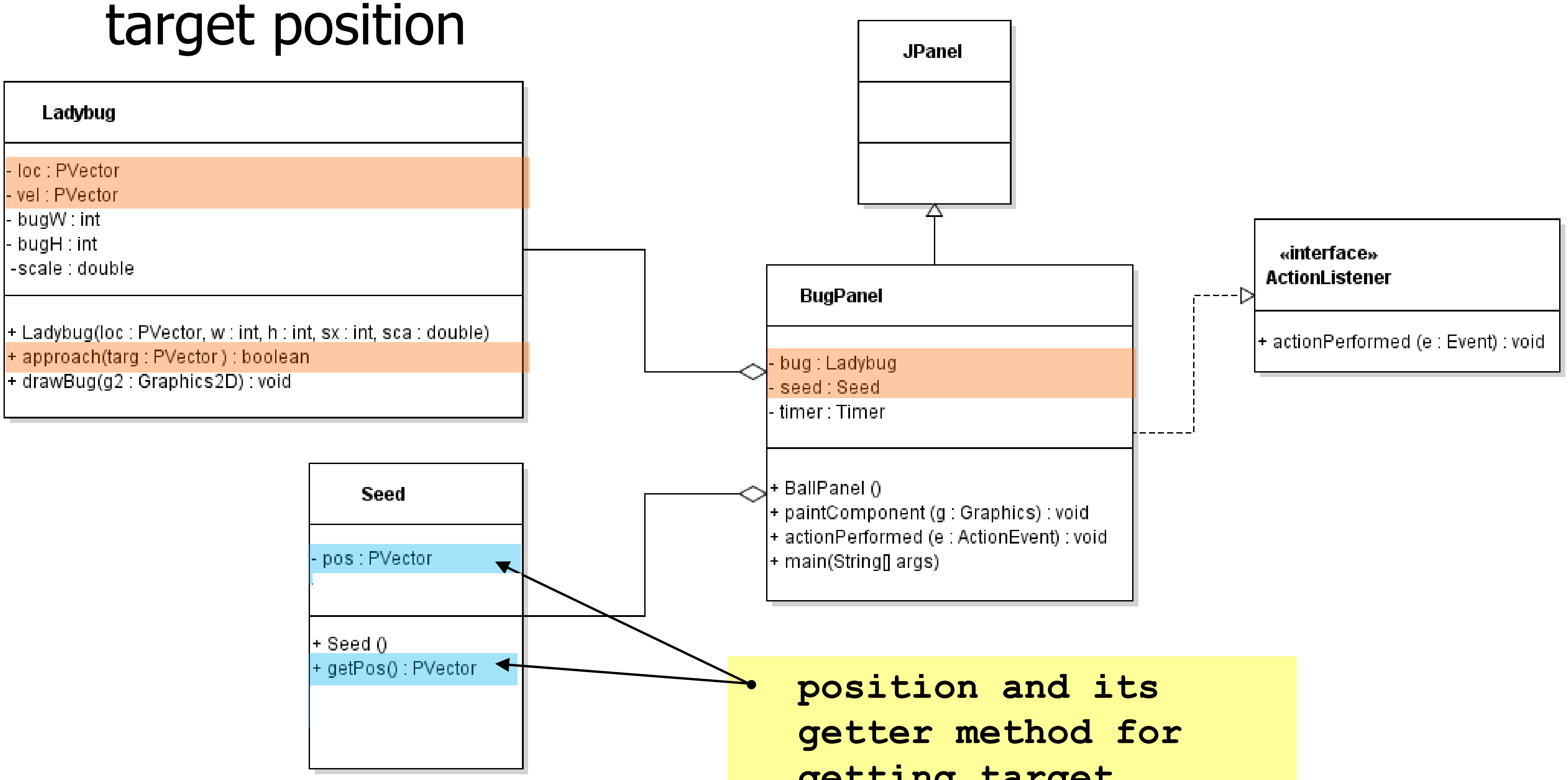


PPP:

- Makes it walk randomly to hunt
- Moves toward the seed once it's appeared
- Eats it when getting close enough
- Goes back to random walk

Design the two Classes

- Remodel *Ladybug*'s motion attributes with *PVector*
- Create a method *approach* that would move toward a target position



JPanel

Ladybug

- loc : PVector
- vel : PVector
- bugW : int
- bugH : int
-scale : double

+ Ladybug(loc : PVector, w : int, h : int, sx : int, sca : double)
+ approach(targ : PVector) : boolean
+ drawBug(g2 : Graphics2D) : void

«interface»
ActionListener

+ actionPerformed (e : Event) : void

BugPanel

- bug : Ladybug
- seed : Seed
- timer : Timer

+ BallPanel ()
+ paintComponent (g : Graphics) : void
+ actionPerformed (e : ActionEvent) : void
+ main(String[] args)

Seed

- pos : PVector

+ Seed ()
+ getPos() : PVector

position and its getter method for getting target position in panel

Codify the approach method

```
class Ladybug {
   …

  boolean approach(PVector targ) {
     boolean reach = false;

     //calculate the path to target point
     PVector path = PVector.sub(targ, loc);

     //returns the direction as angle
     float angle = path.heading();

     //make a vel that points toward the
       target
     vel = PVector.fromAngle(angle);
     vel.mult(2);

     loc.add(vel);    //move toward target

     //check if bug reaches target
      if (path.mag()- bodyW/2 <= 10 ) {
         reach = true;
      }
     return reach;
  }
}
```

```
//in BugPanel class
  …
public void actionPerformed(ActionEvent
e){
  //move the bug
    …

  //generates a seed every 10 seconds
    …

 //If bug catches seed eat it by
   setting it to null
    …

  repaint();
}
```

Readings

- Required
  - Readings in Canvas