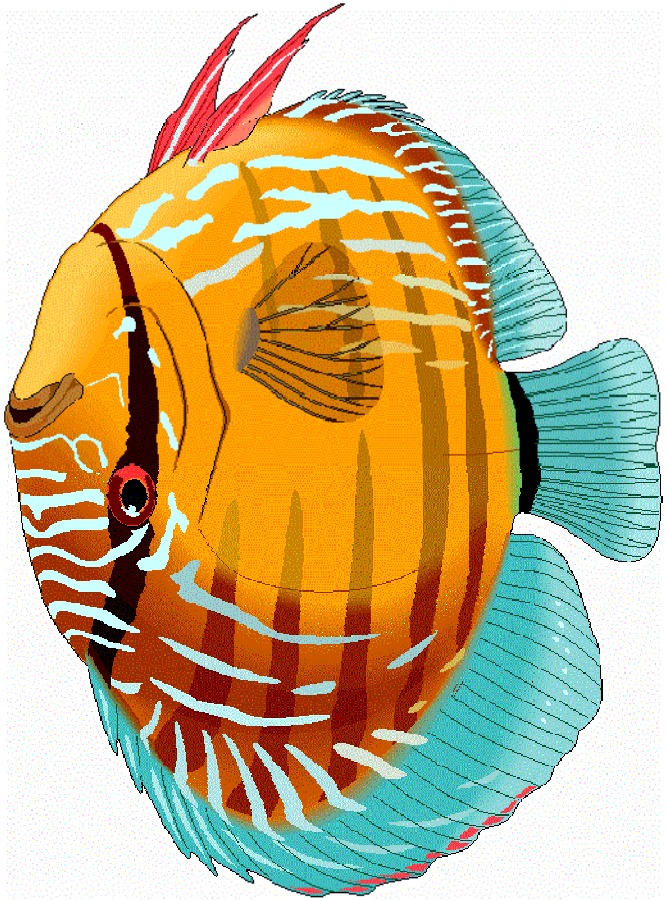
IAT 265

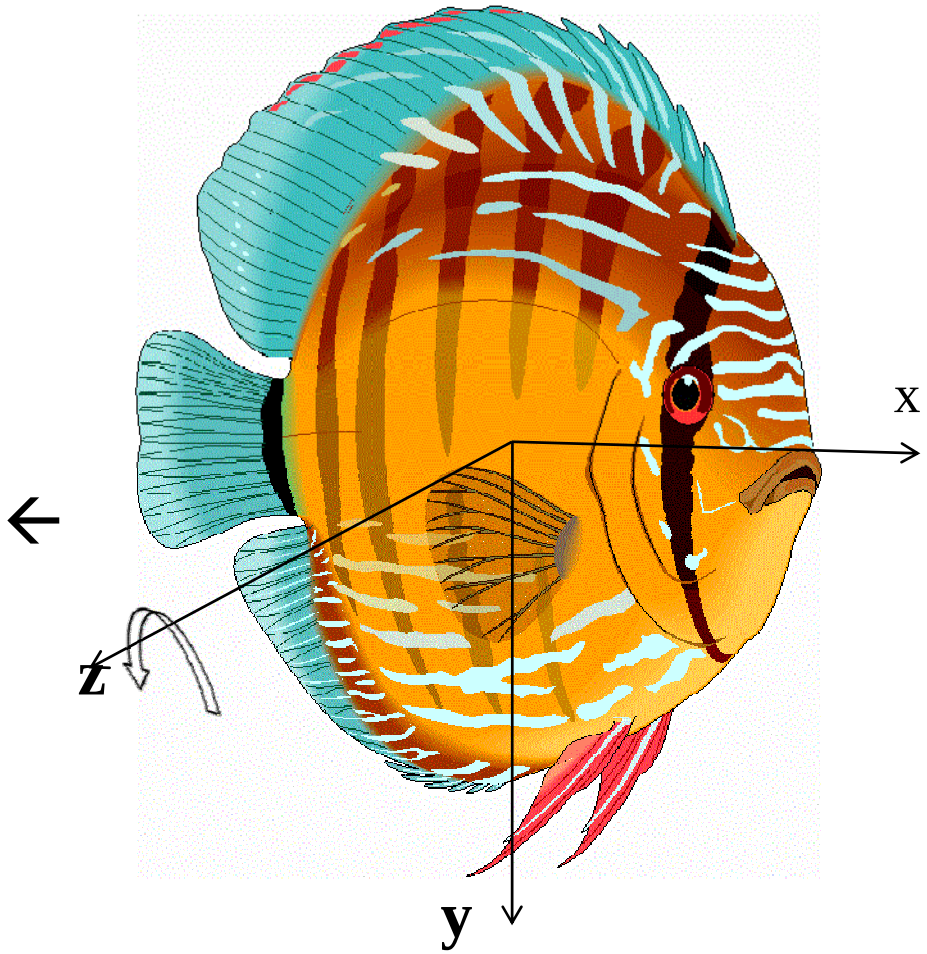
More transformations & Java Loops

Topics

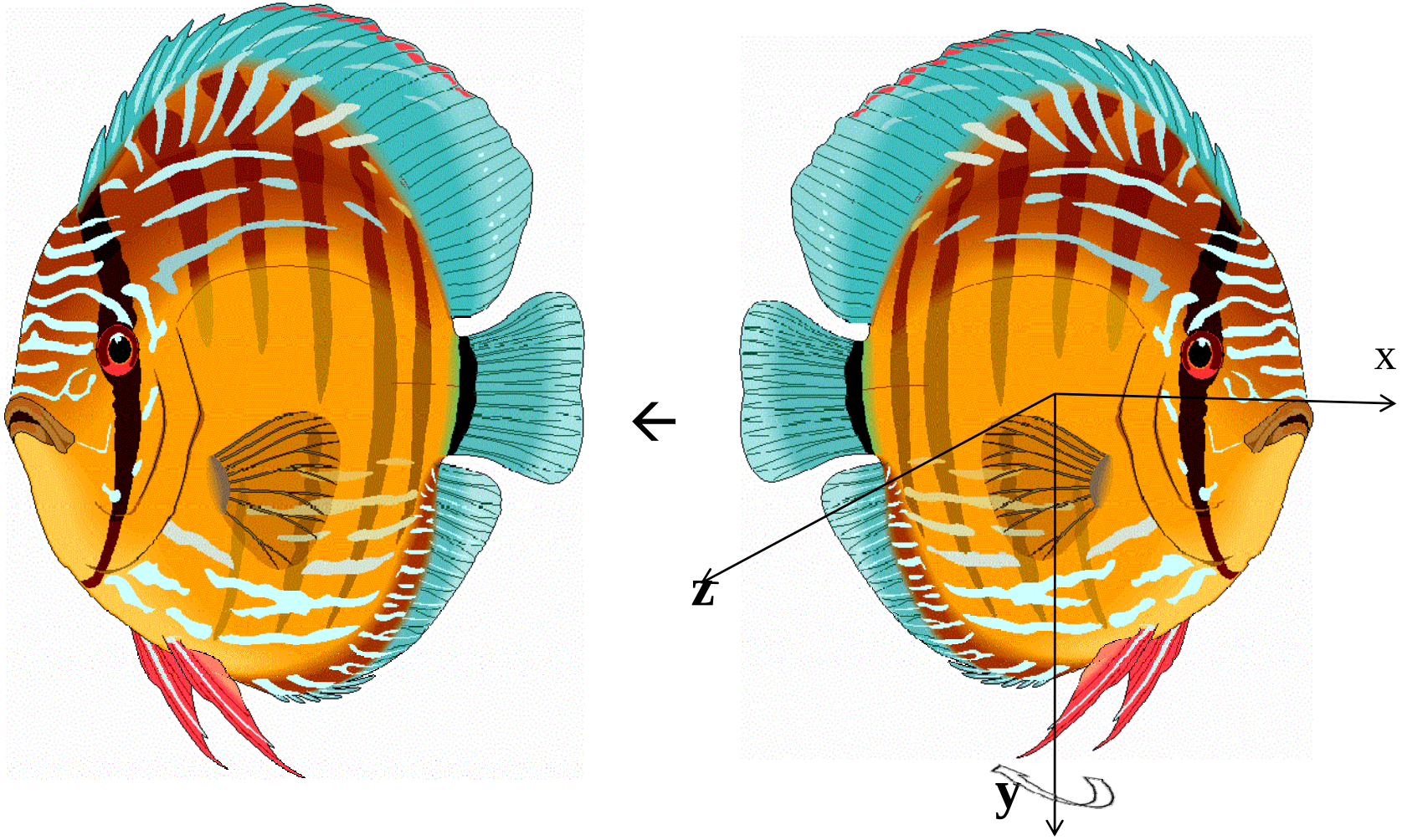
- More transformations
  - Rotate around different axes
  - Scale
  - Handle multi-transformations with pushMatrix/popMatrix
- Java control structures and arrays
  - Loops (*while*-loop, *for*-loop)

Did you run into the scenario after rotation?



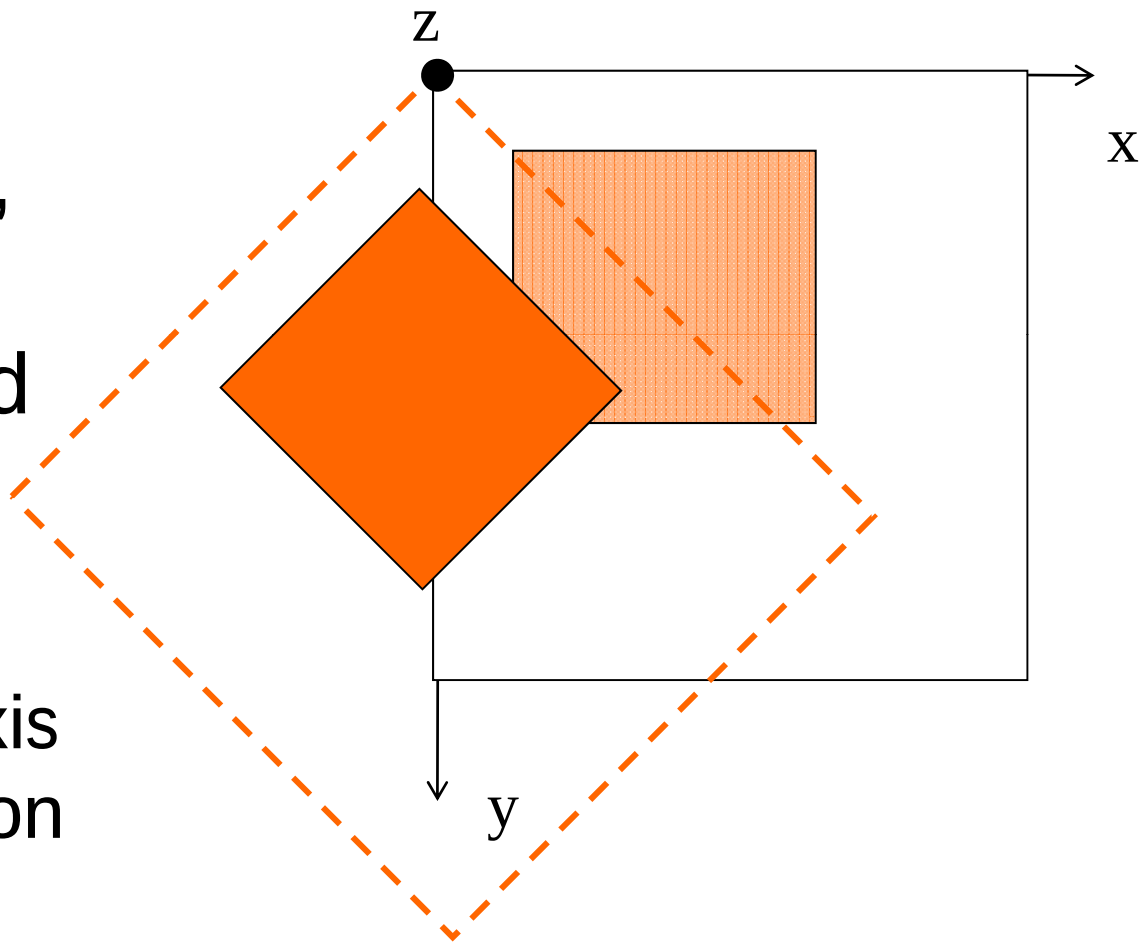


Can we make it flip around y-axis instead?



More Transformations

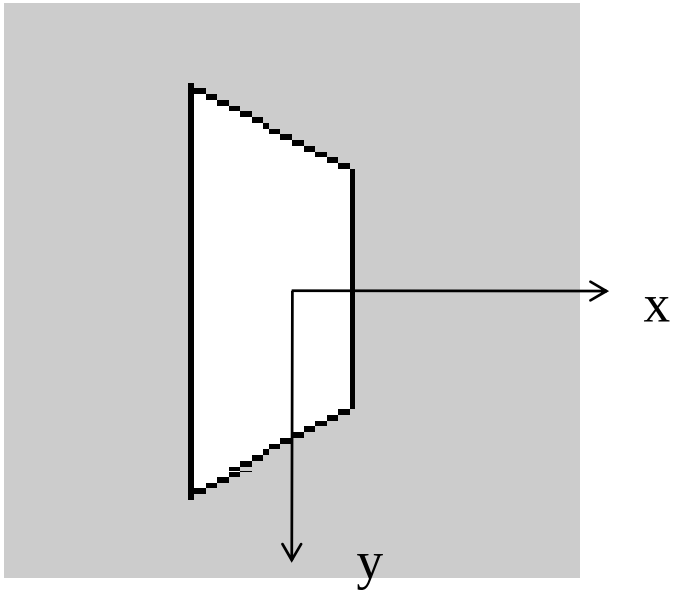
- In last lecture, all our shapes rotated around Z-axis
  - the default axis for 2-D rotation



Rotation around different Axes

- Processing also allows for rotation around x or y-axis with *rotateX()* and *rotateY()*
  - Requires P3D or OPENGL mode

```
size(100, 100, P3D);
translate(width/2, height/2);
rotateY(PI/3.0);
rect(-26, -26, 52, 52);
```



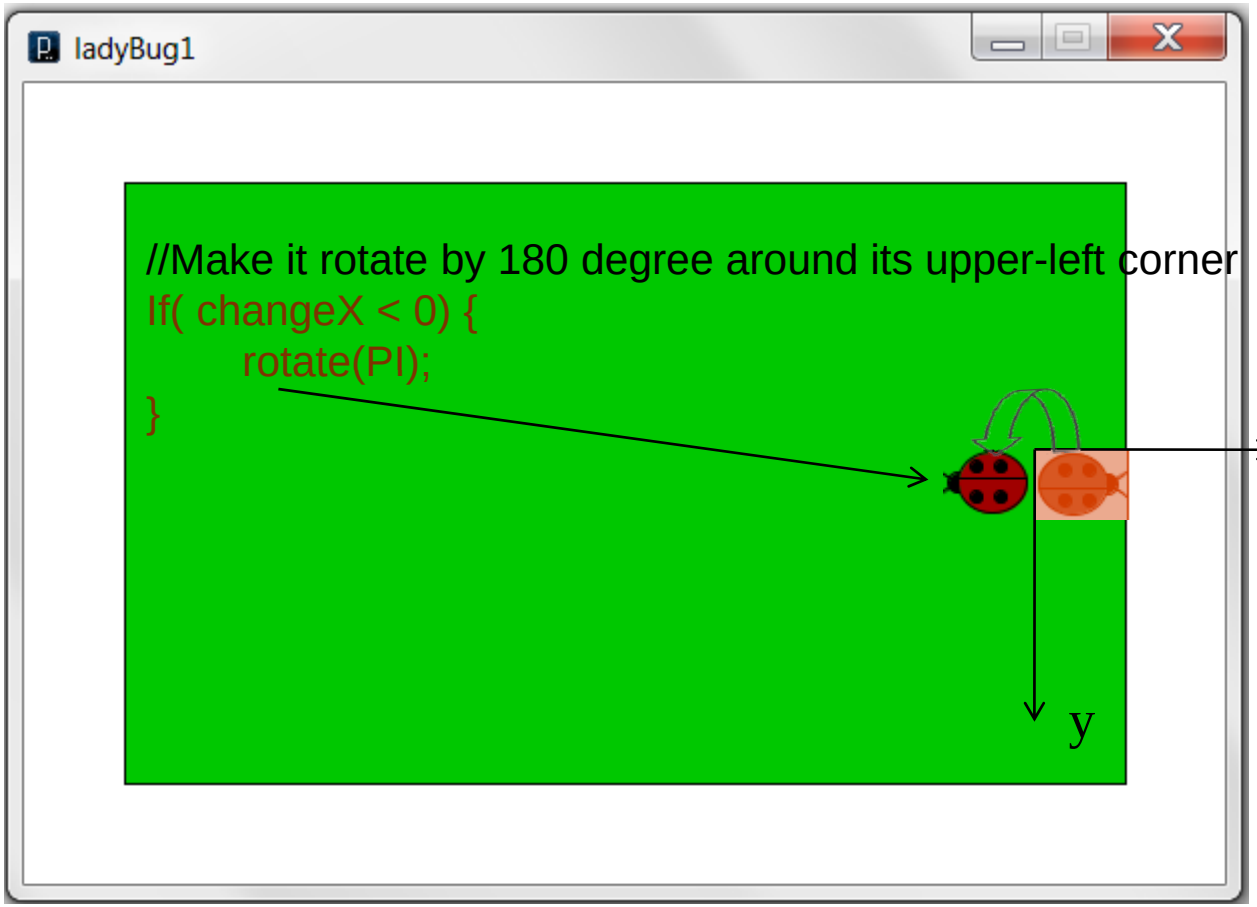
P3D vs. OPENGL

- P3D: the default 3D renderer of Processing
  - Note that anti-aliasing (enabled with *smooth*()) is not working with P3D

- OPENGL: a 3D renderer that employs hardware acceleration
  - Have the best performance on displaying large numbers of shapes in high-resolution
  - Need to: import processing.opengl.PGraphicsOpenGL;

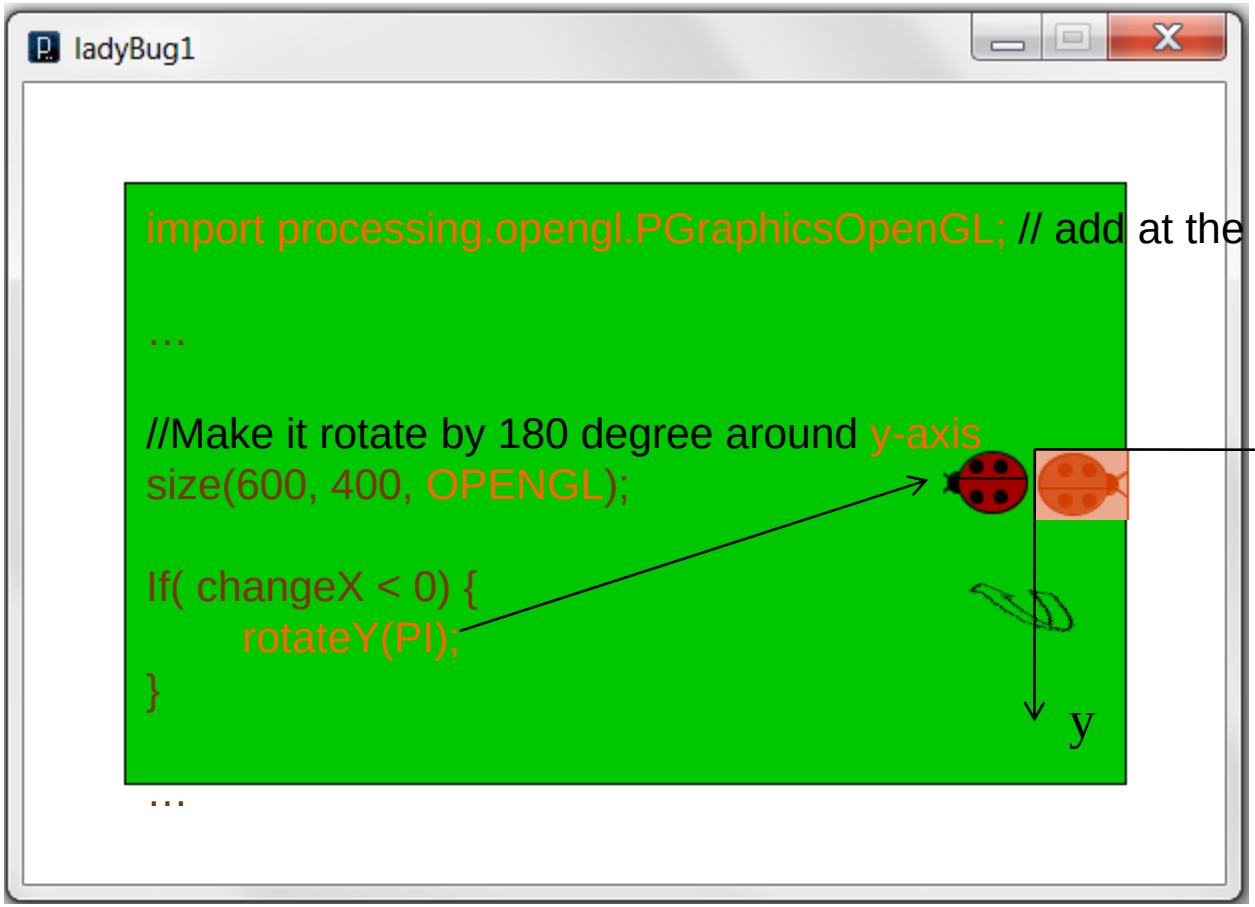
Review: translate & rotate a ladybug

- Step 4: make the bug rotate when reversing



Flip ladybug around y-axis

- Step 4: make the bug flip when reversing

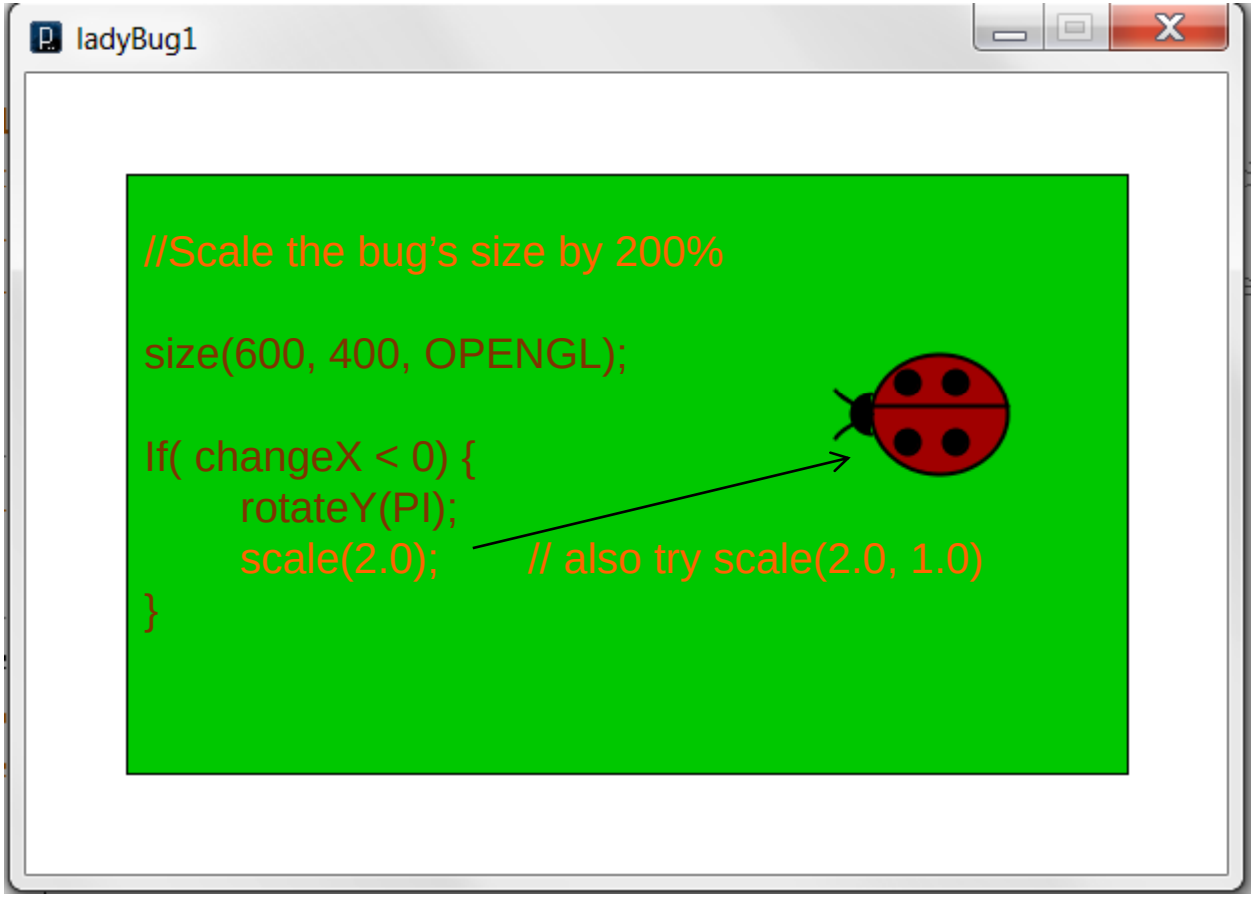


More Transformation: Scale

- scale() increases or decreases the size of objects on screen. Syntax:
- scale(float *size*);
  - size: percentage to scale the object.
  - e.g. scale(2.0) or scale(0.5) scales the objects' size by 200% or 50%
- scale(float *x*, float *y*);
  - x: percentage to scale in the x-axis
  - y: percentage to scale in the y-axis

Scale the ladybug

- Scale the bug by 200% when reversing



How to deal with Multiple Transformations?

- When multiple translations/rotations apply to different shapes, these transformations may affect each other in unexpected ways
  - So the question is: how can we translate and rotate individual shapes without affecting others?
  - The answer: Save the **current transformation matrix** before a transformation and restore it afterwards
  - Transformation matrix: keeps track of location (origin) & orientation of the coordinate system

Pushing and Popping

- Pushing is a way to say:

  “Remember the current origin & orientation!”

```
pushMatrix();//Processing function
```

- Popping is a way to say:

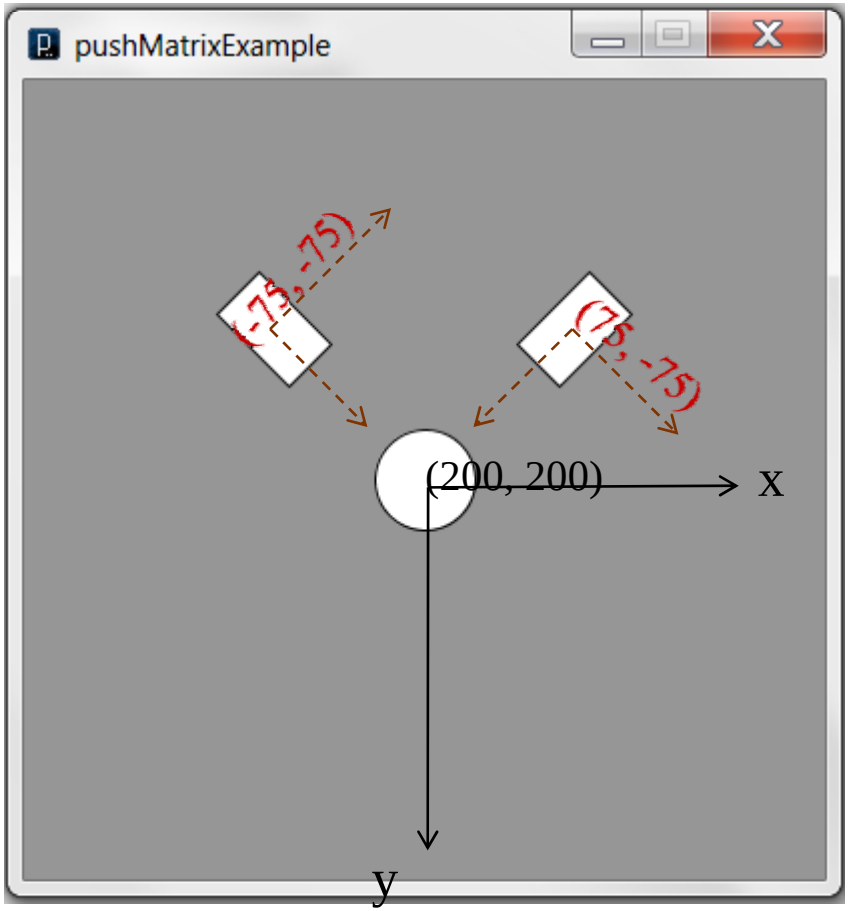
  “Take me back to the way things once were!”

```
popMatrix();//Processing function
```

Ex: How to create a figure like this?

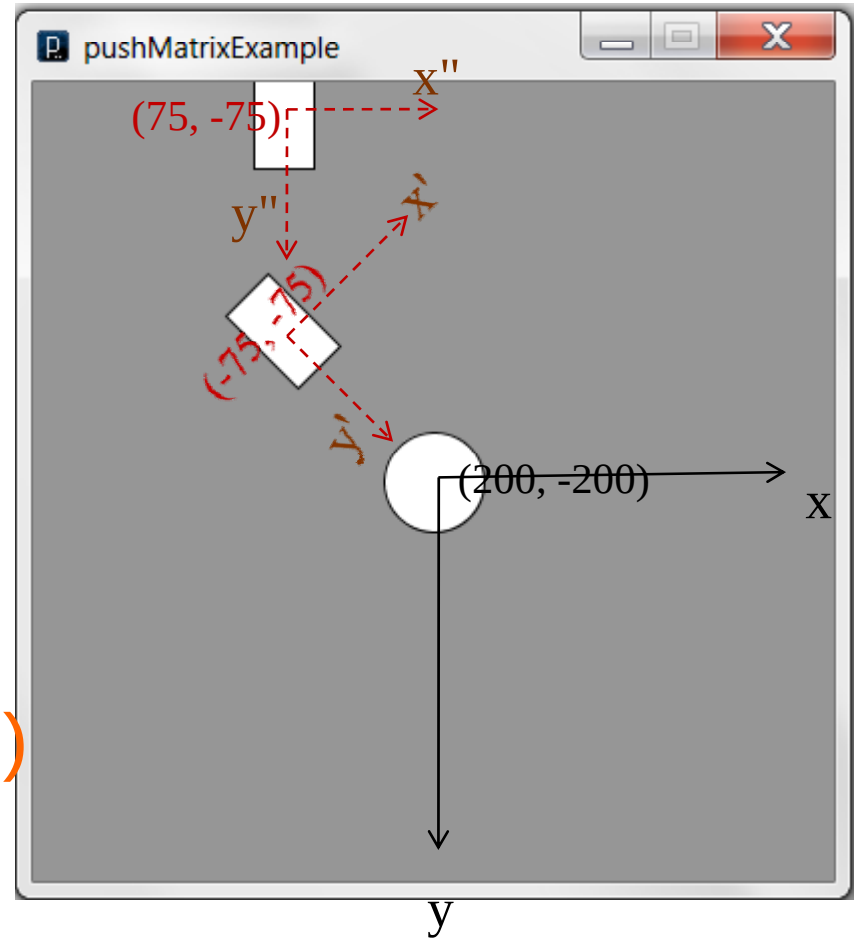
- Just do the followings?

```
size(400, 400);
rectMode(CENTER);
translate(200,200);
ellipse(0, 0, 50, 50);
translate(-75,-75);
rotate(-PI/4);
rect(0,0,30,50);
translate(75,-75);
rotate(PI/4);
rect(0,0,30,50);
```



However the result is …

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

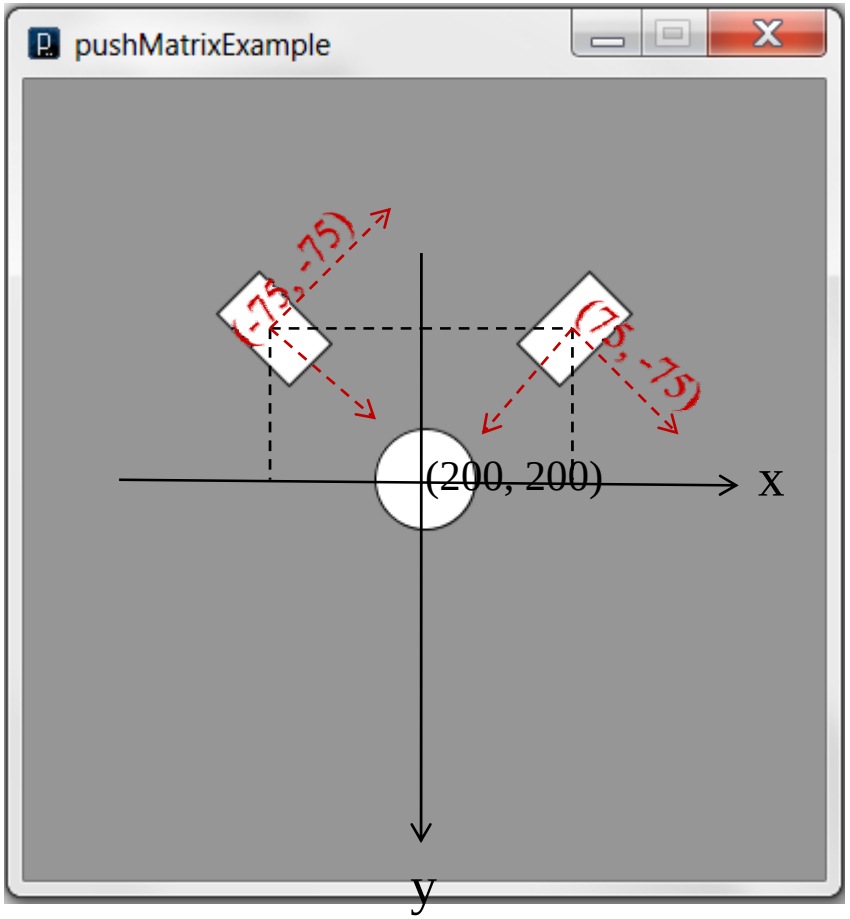


- This is because (x”~y”) is based on (x’~y’), rather than (x~y)

Solution: use push & pop

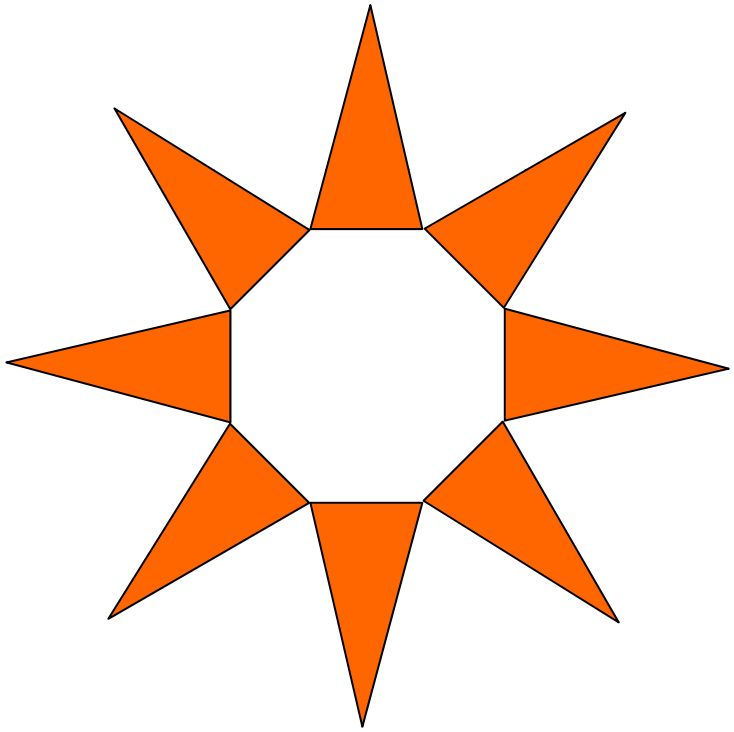
- translate(200,200);
  ellipse(0, 0, 50, 50);
- pushMatrix(); //save(x~y)
  translate(-75,-75);
  rotate(-PI/4);
  rect(0,0,30,50);
  popMatrix(); //restore(x~y)

- pushMatrix(); //save(x~y)
  translate(75,-75);
  rotate(PI/4);
  rect(0,0,30,50);
  popMatrix(); //restore(x~y)



Think about how to…

- Draw the figure and make it rotate over time around the center of the window
  - A nice rule of thumb: sandwich translation and rotation for each of the shapes between pushMatrix & popMatix, so that they won't interfere with each other

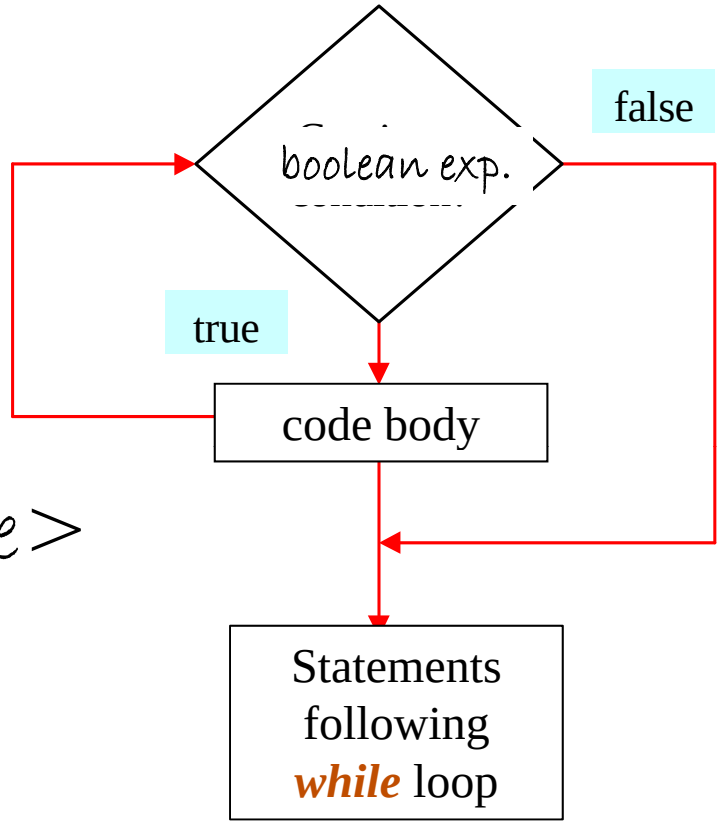


Loops

- Sometimes you want to execute code multiple times
  - E.g. draw() is being called in a loop – an infinite loop
- Java provides some other looping mechanisms: loops based on certain condition
  - They all test some *boolean* expression (just like an *if* statement does) and continue to execute code while the expression is *true*
  - Loops will stop when the expression becomes *false*

while loops

```
while( <boolean exp.> )
{
   <code to execute each time>
}
```



- Repeatedly executes the code body while the *boolean expression* is true

Example of *while* loops

- Calculate sum of number 1~99:

```
int i = 0;
int sum = 0;
while(i < 100){
  sum += i;      //sum = sum +i;
  i++;           //i = i + 1;
}
println("sum = " + sum);
```

Using Compound Assignment Operators

- **+=** is one of the compound assignment operators
  - ***sum+=i*** ; //the same as ***sum= sum + i;***

- Compound assignment operations combine an arithmetic operation with an assignment operation

Compound Assignment Operators

TABLE 4.1 COMPOUND ASSIGNMENT OPERATORS

Operator	Description	Example	Same Result as
+=	Increment assignment operator	`x += 5;`	`x = x + 5;`
-=	Decrement assignment operator	`x -= 5;`	`x = x - 5;`
*=	Multiplication assignment operator	`x *= 5;`	`x = x * 5;`
/=	Division assignment operator	`x /= 5;`	`x = x / 5;`
%=	Modulus assignment operator	`x %= 5;`	`x = x % 5;`

++ and -- operators

- **++**: a shorthand for adding one to a ***numeric variable***:

```
x++; // the same as x = x + 1;
```

- **--**: a shorthand for deducting one from a ***numeric variable***:

```
x--;   // the same as x = x - 1;
```

for loops

```
for( <init. statement> ; <boolean expression> ;
                                    <final statement> )
{
   <code to execute each time in loop>
}
```

- First execute *init. Statement* (to initialize a counter)
- Second test *boolean expression* (counter vs. control value) – if true, execute the code *once*
- Then *repeat* the following:
  - execute *final statement* – change the counter
  - test *boolean expression* → execute *code* if true

Example of *for* loops

- Calculate the sum of number 1~99:

```
int sum=0;
for(int i = 0; i < 100; i++){
  sum += i;
}
println("sum = " + sum);
```

Converting *for* to *while*

- Seeing how *for* loops can be converted to *while* loops helps you understand *for* loops

```
for(<init. stmt> ; <boolean exp> ; <final stmt> ) {
   <code>
}

// is the same as

<init. stmt> ;
while( <boolean exp> ) {
   <code>
   <final stmt> ;
}
```

Counting backward with *for* loops

- Sometimes you may find that you need loops that count backward

- How should you implement such a backward counting *for*-loop?
  - Initialize the ***counter*** to a value **higher** than the ***control value***
  - Then ***decrement*** it with each iteration until the *control* value is reached

Example of Counting Backward loop

```
/*
 * CountDown: Demonstrates how to make a for loop
 * count backward by using the decrement (--) operator
 */
println("Countdown:");
for (int t=10; t > 0; t--){
  print(t + " ");
}
print("\nBLASTOFF!");
```

The *break* Statement

- The ***break*** statement can be used to break out of a loop explicitly

- Examples of ***break*** to break an infinite loop:

```
int count =0;
 while (true) {
   println(count + " true");
   if (count==10){
      break;
   }
   count++;
 }
 println("Out of Loop");
```

Summary

- More transformations
  - Rotate around different axes
  - Scale
  - Handle multi-transformations with pushMatrix/popMatrix
- Java control structures and arrays
  - Loops (*while*-loop, *for*-loop)