CPE 102 Lab 7 - Introduction to Processing
(Demo only : Due 11/14/16)

Goals:

- To learn how to use the Processing Development environment.
- To get a head start designing your Monster for the Project 3 Maze Game.

Orientation:

You are going to design and write code to render a drawing of a Monster object to the screen using the Processing development environment. According to the Processing website, "Processing is an open source programming language and environment for people who want to create images, animations, and interactions." The Processing Environment uses a programming language based on java. We will use Processing environment to give life to our Maze Game program.

The Maze rendering will be as 5 points as extra credit to your projects. If you are not planning to take this extra credit you don't need to worry about drawing the objects in the Maze. Just fallow the direction for Lab7.

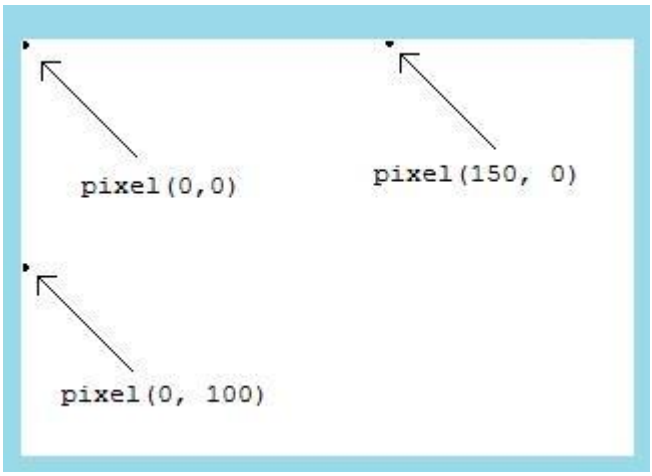
You first need to decide on a theme for your Maze Game, which will eventually need to render an Explorer, Monsters, and Treasures. One theme is a cat Explorer dodging bull dog Monsters in search of cheese Treasures. Here is an example of a Monster:

Because the Monster is intended to be used with the finished Maze Game, you will render it such that it fits into a 50x50 pixel square. To make the finished lab more fun, we will add a mouse interaction to enable your Monster to move about the screen.

Processing Basics:

Using Processing, you will write code to draw shapes on the screen. Processing comes with an extensive reference guide that you can find here.

Drawing in Processing is based on pixels. The origin of the coordinate system is the top left corner of the drawing window. Positive x is to the right, positive y is *down*.



You can draw primitive shapes using some of the following commands.

```
ellipse(x, y, width, height);
line(x1, y1, x2, y2);
quad(x1, y1, x2, y2, x3, y3, x4, y4);
rect(x, y, width, height);
triangle(x1, y1, x2, y2, x3, y3);
```

By default, shapes are drawn with a one pixel outline (stroke) around them. You can turn off this line or change the thickness of it with these commands.

```
noStroke();
strokeWeight(pixels);
```

Shapes have a fill color and a stroke color. The background of your drawing has a color, as well. You can set the color with the following methods, and using a variety of formats. All of the values below specify an int in the range of 0-255. 0 means black for grayscale, or none of the color for RGB. 255 means white for grayscale, or all of the color for RGB. You can also use a hex value for the color. For example, the value #FFA200 is pumpkin orange. A great website for selecting color is here.

```
background(grayscale);
background(red, green, blue);
fill(grayscale);
fill(red, green, blue);
stroke(grayscale);
stroke(red, green, blue);
```

You can even create a color and save it for later use. This makes it easy to switch back and forth between colors when writing your code. For example:

```
color c = color(50, 55, 100); // Create a color for 'c'
fill(c); // Use color variable 'c' as fill color
```

Sample Sketch:

Every Processing sketch needs to have a "main" file that contains a draw() method. *This file must be in a directory with the same name.* The draw() method is called continuously at a rate of 60 times per second. Since the draw() method is called repeatedly, shapes that change position over time will appear to move (animate). You can optionally include a setup() method that is called one time at the start of the sketch. The setup() method is good for doing things like setting the size of the sketch window.

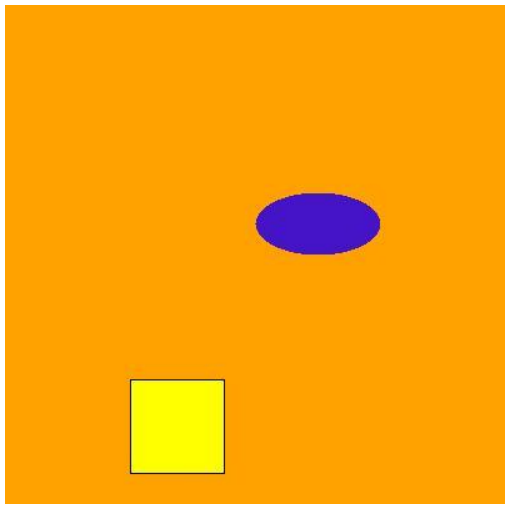
The following is sample code that creates the Processing sketch shown below (note there is no animation in this sketch):

```
void setup()
{
   size(400,400); // set the size of the drawing window
}

void draw()
{
   background(#FFA200); // set the background to orange

   stroke(0); // stroke color black
   fill(255, 255, 0); // yellow
   rect(100, 300, 75, 75);

   noStroke(); // turn off the outline stroke
   fill(70, 20, 200); // set the fill color to dark blue
   ellipse(250, 175, 100, 50);
}
```



Transformations:

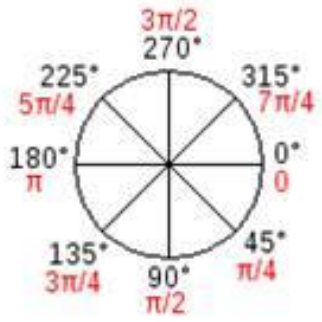
Two important transformations in Processing are **translate** and **rotate**. Translating lets us draw our shapes as if they were centered at the origin but render them elsewhere in the sketch. Often it is easier to compute pixel location based on the (0,0) pixel. Rotating, rotates shapes (or the whole scene). For example, using rotation you can draw an ellipse at a 45 degree angle.

`translate(x,y);`

Moves the origin of the scene to the specified (x,y) location. Shapes drawn before this call will be drawn in their normal location. Shapes drawn after this call will be drawn as if the origin has moved. For example, if translate (200,200) were called and *then* and ellipse was drawn at the (0,0) pixel. The ellipse would be rendered centered at pixel (200,200). This transformation is effective for the remainder of the sketch (including subsequent calls to draw()). See push/pop matrix below to understand how to contain your transformations to certain parts of your sketch.

`rotate(radians);`

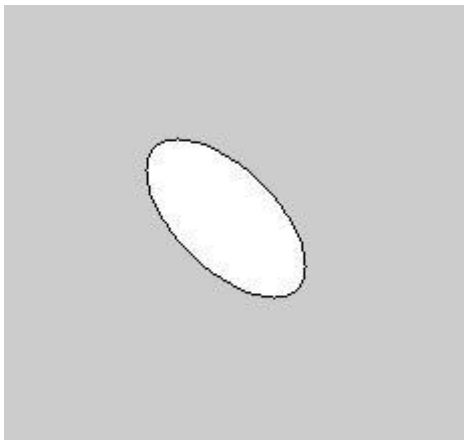
Rotates, the x/y axes so that subsequent shapes are drawn at an angle. You may use the constant PI when specifying the radians, or you may use the `radians()` method that takes degrees and returns the equivalent angle in radians. This transformation is effective for the remainder of the sketch (including subsequent calls to draw()). See push/pop matrix below to understand how to contain your transformations to certain parts of your sketch.



Containing your transformations:

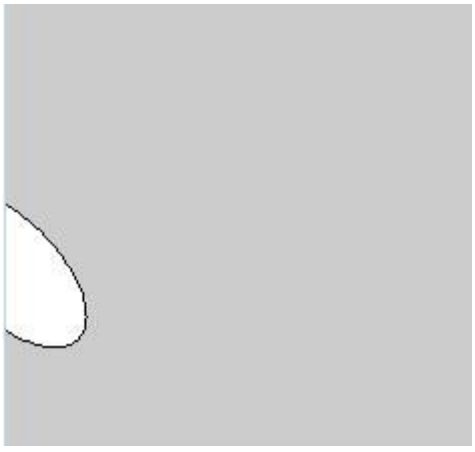
Transformations *build on each other* and remain in effect after they are called. The following three calls would draw an ellipse centered at (200,200) and rotated 45 degrees:

```
translate(200,200);
rotate(PI/4);
ellipse(0, 0, 100, 50);
```



Since these transformations build on each other, order is import. The following three calls look very similar to the code above, however the x/y axes are rotated first. Then when the origin is translated to (200,200) it is translated based on the rotated axes:

```
rotate(PI/4);
translate(200,200);
ellipse(0, 0, 100, 50);
```



Lastly, these transformations remain in effect for the remainder of the sketch. To contain them to a certain portion of the sketch you may use the push/pop matrix methods. First call pushMatrix(). Then apply any transformations you would like. Then draw the shapes that you want transformed. Finally, call popMatrix() to cancel the effect of the transformations for the rest of the sketch.

```
pushMatrix()
translate(200,200);
rotate(PI/4);
ellipse(0, 0, 100, 50);
popMatrix()
```

What to Do:

1. Download Processing if you do not already have it.
2. Read, carefully, the MonsterMash.pde code given to you. This code will control the sketch and call methods of your Monster class to do the actual drawing. For the most part you should not change this code. You may set a different background color if you like.
3. Save the MonsterMash.pde code in a directory named MonsterMash
4. In a file named "Monster.pde", implement the Monster class according to the specification in Monster.html.
    - Notice that the instance variables are shown in the javadocs (Field Summary). Make sure your instance variables are private. They are shown here for your clarification. These are the only instance variables you need.
    - Be sure to read the "Detail" description of all fields and methods in the javadocs. There is important information in there for you.
    - Make sure you save the Monster.pde file in the "MonsterMash" directory along with MonsterMash.pde.
    - I *highly* recommend making a skeleton version of the class first so that your Processing sketch can compile and run. You will want to check your progress as you add shapes to the drawing of your Monster.
    - Start with the constructor and draw methods. Develop your sketch incrementally and run your code often to check how your Monster looks.
    - **After** you get your Monster looking how you want it, finish implementing the rest of the methods: getX, getY, move, and setVector so that your Monster can move around the scene in the direction of any mouse clicks.
5. After you get the lab working to match this specification, feel free to play with MosterMash.pde to change the behavior and add things to your sketch!

Additional Resources:

To help you get started, here is the complete draw() method for the bull dog Monster.

```
public void draw()
{
    pushMatrix();
    translate((int)x, (int)y);

    noStroke();

    // head
    fill(120);
    ellipse(0, -10, 25, 25);

    // eyes
    fill(255);
    ellipse(0, -10, 20, 18);
    fill(120);
    ellipse(0, -17, 20, 10);
    fill(0);
    ellipse(-2, -10, 2, 2);
    ellipse(2, -10, 2, 2);

    // muzzle / jowels
    fill(180);
    ellipse(0, -3, 22, 10);
    fill(120);
    rect(-7, 0, 14, 13);
    ellipse(0, 10, 20, 10);
    fill(180);
    ellipse(-10, 5, 7, 15);
    ellipse(10, 5, 7, 15);

    // nose
    fill(0);
    ellipse(0, -6, 8, 5);

    // teeth
    fill(255);
    triangle(-7, 0, -5, -5, -3, 0);
    triangle(7, 0, 5, -5, 3, 0);

    // ears
    fill(120);
    ellipse(-10, -20, 6, 3);
    ellipse(10, -20, 6, 3);

    popMatrix();
}
```