These notes are for the Theta figure drawn in class today – they are matched to the folder named Theta. The first three demos have not associated notes. Refer to the comments in the code and the API for explanations about the functions being used.

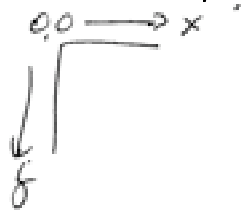
First Part of Class Brief discussion:

Was there a "learning curve" – did the last initial take less time than the first? If this was true for you, they you were learning something about programming.

"Rules learned" while coding the program – those things you have to do to get your code to compile such as:
semicolon at the end of a line
parenthesis after each function call
multiple arguments must be separated by commas
spelling and case are very important
stuff done last appears on top covering stuff done first
default value exist such as window size is 100x 100, stroke is black and 1 pixel wide, fill is white, background is some shade of grey.

Today's Discussion:
-why magic numbers are bad – can not easily move or resize the initials
the type int (integer or whole numbers) and float (decimal numbers)
-declaring and initializing variables of type int and float
-arithmetic expressions and evaluation for + - * / and the % operator was must mentioned
-anchor points and relative locations were discussed as a way to make moving and resizing the initials easy.

The coordinate system in the drawing window:



The unit of measure is the pixel – pixel is a contraction of the two words picture element -- do not ask me where the "x" comes from…

A pixel is the smallest dot you can draw on the screen. If the strokeWeight has width of 1, then a point is drawn with the dimension of 1 pixel by 1 pixel.

A line of code like this:

```
  line(50, 55, 60, 60);
```

is a function call. You are telling Processing to call or execute the function line using the four int values as parameters. For most of you., each line of code in your program was probably a function call. Not all function calls have parameters.

```
  noStroke(   );
```

However, the parentheses are required.

Function calls must end with a semicolon.
If you have more than one parameter, they must be separated by commas.
Spelling AND the case of the letters in the function name are important.

```
 NoStroke(   );
 or
 nostroke(   );
```

are not the same as

```
 noStroke(   );
```

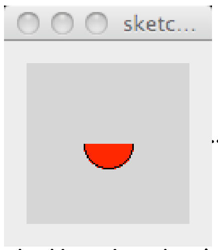
and only one will work!

Using arcs:
To use the arc function, you need to know where zero degrees is and which way is positive and which way is negative rotation.
Positive rotation is clockwise. You must draw arcs with a clockwise rotation.
This code:

```
  arc( 50, 50, 30, 30, 0, PI);
```

draws this arc:

PI radians is here ---->



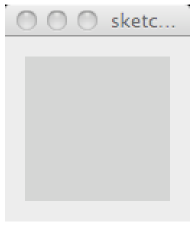
<------ Zero radians is here

Rotation is clockwise.

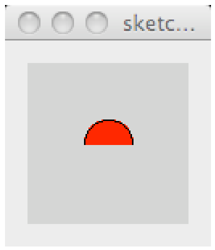
The 0 point is on the right side of the semi-circle. The point PI is on the left side. PI is a constant value defined somewhere deep in Processing.

This code:

```
arc( 50, 50, 30, 30, PI, 0);
```
draws nothing:



because the direction of rotation is seen by Processing as being negative or counterclockwise. To draw the semicircle on the "other side" you have to use this code:
```
arc( 50, 50, 30, 30, PI, TWO_PI);
```
which draws this arc:



One more thing – you have to use radian values for the last two parameters of the call of the arc function. If you are not comfortable to thinking in radians, you can use a Processing function to compute them. The function is named radians( ). The parameter is the number of degrees you need converted to a radian value. This function will return the radian value for the parameter.

So instead of coding a call of the arc( ) function like this:
```
arc(50, 55, 60, 60, PI/2, PI);
```

You can do this:
```
arc(50, 55, 60, 60, radians( 90 ), radians( 180 ) );
```
Processing will execute the function radians( ) with the parameter 90 and substitute the value it returns as the fifth parameter. It does the same thin for the second call of the radians function but it uses the value of 180 as the sixth parameter.

Jim's ramblings about homework #2

The code you wrote for hw1 can be described as "hardwired". Every coordinate point and sizes was an actual honest number. Hardwired code produces the same output every time. This is fine if we never want to alter what we see on screen.

BUT, what if we want to move your initials up or to the right or make them a bit larger? You have to recode almost every line of code with new numbers. Your code is hardwired or fixed to one output. This is not good.

The parameters you used – those actual honest numbers are called magic numbers because they sorta' appear like magic. To another reader, it is not obvious why you used them. And they do not allow you to easily move or resize your initials.

In general, magic numbers are almost always bad. Certain numbers like 0 and 1 are not magic. But 42 – at least in this universe – is magic.

So – magic numbers and hardwired code will be considered very bad in 257. Let's see how we can avoid them and, at the same time code our initials to allow us to move them and/or resize them easier.

BEFORE we do that, here is some minutiae …
Processing uses two types of number data:
  int which is used for whole numbers or integers
  float which is used for decimal or fractional numbers.
There are other types of number data in Processing but these two are the ones will use for most of what we do.
    1 is an int.   1.0 is a float.
These operators work with numeric data. Most of these you are very familiar with:
+ for add
- for subtract
* for multiplication (this is the asterisk)
/ for division.
There is another very important division operator (%) that we will discuss later.

The rules for using these are very simple.
int + int ➔ int
int - int ➔ int

int * int ➔ int
int / int ➔ int

float + float ➔ float
float - float ➔ float
float * float ➔ float
float / float ➔ float

You can mix your arithmetic by adding, multiplying, subtracting and dividing int and float values but what is the result?
int + float ➔ float
float - int ➔ float
float * int ➔ float
int / float ➔ float

If either part of the operation involves a decimal value, the result is a decimal. This evaluation avoids "loss of data" or "loss of precision" problems in our code.

Think about this:
1 + 2.3 ➔ ???
What is the best answer? 3 or 3.3? The designers of the Processing and many other languages hate to lose data so the result is a decimal number instead of an integer.

For HW2 we will use the float type of data.

Back to the original problem – avoiding magic numbers.

Here are two pieces of code that do the same thing:

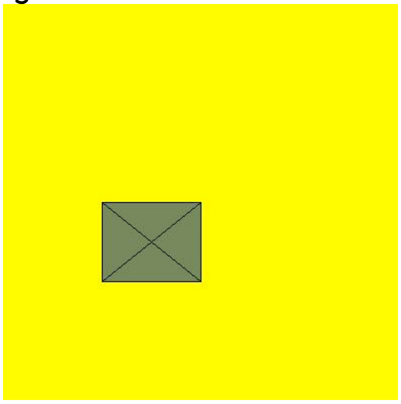
```
fill( 100, 120, 75 );
rect( 100, 200, 100, 80);
line( 100, 200, 200, 180);
line( 100, 180, 200, 100);
```

```
float x, y, wd, ht;
x = 100;
y = 200;
wd = 100;
ht = 80;

fill( 100, 120, 75 );
rect( x, y, wd, ht);
line( x, y, x+wd, y+ht);
line( x+wd, y, x, y+ht);
```

The code on the left is what you wrote for hw1. The code on the left uses four variables that store float values and makes the same drawing as the code on the right.

The advantage of the code on the right is that we can move the drawing to the right by just changing the value of x.
Here is the original drawing:

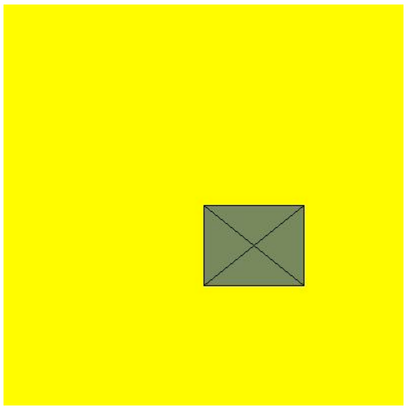


Now if we alter the value of the variable x as shown below

```
float x, y, wd, ht;
x = 200;
y = 200;
wd = 100;
ht = 80

fill( 100, 20, 75 );
rect( x, y, wd, ht;
line( x, y, x+wd, y+ht);
line( x+wd, y, x, y+ht);
```

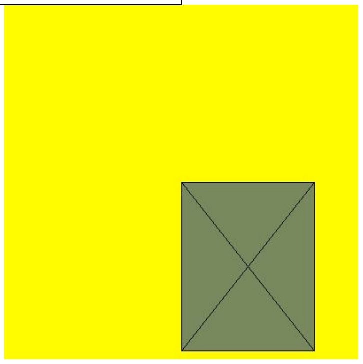
we get this result:



We can make the rect smaller or bigger and keep the lines connected to the corner by just changing the wd and ht values:

```
float x, y, wd, ht;
x = 200;
y = 200;
wd = 150;
ht = 190

fill( 100, 20, 75 );
rect( x, y, wd, ht;
line( x, y, x+wd, y+ht);
line( x+wd, y, x, y+ht);
```



By spending a bit more time thinking about our code and using variables, we can reduce the time needed to modify the output in the future.

The function calls of rect( ) and line( ) do not use magic numbers.
What about the values of fill( ) – aren't they magic??? Not really because the community of programmers using Processing understand that these values range between 0 and 255. So they really are not magic…[1]

The idea of a variable in a program is probably new to you. We use variables to store values that either:

- we want to change from time to time
- will change as our program is executed

The first bullet is true for hw2. We will encounter the second bullet a bit later. Let's walk though a simple thought process for "thinking about variables."

Step 1: realize that you need a variable – we need them because it is in the specification.

More on 0 to 255 later.

Step 2: figure out what data will or might change.

Step 3: figure out what is the type of data you need to store -
you ***will*** store float values.

Step 4: type in the name of the type

```
float
```

Step 5: choose a good name that is meaningful such a sx, y, wd, ht. The code we began in step 4 expands to be:

```
float x, y, wd, ht;
```

Step 6: assign the variables their values

```
x = 50;
y = 70;
wd = 100;
ht = 110;
```

Step 7: replace the magic numbers in the code with the variables or with expressions that use these variables.

Steps 4 and 5 can be described as "declaring the variable" or a "variable declaration":

```
float x, y, wd, ht;
```

We are telling Processing that we want four variables to store float (decimal) values and we want them named x, y, wd, and ht.

Step 6 is called the variable initialization the first time. It is also called the variable assignment.

```
x = 50;
```

The = sign is not used to check for equality.
In Processing the = sign is the "assignment operator".
This line is read in English as:
x is assigned the value 100 or
x gets 100

Step 7: replace the magic numbers with the appropriate variable name:

```
rect( x, y, wd, ht );
```

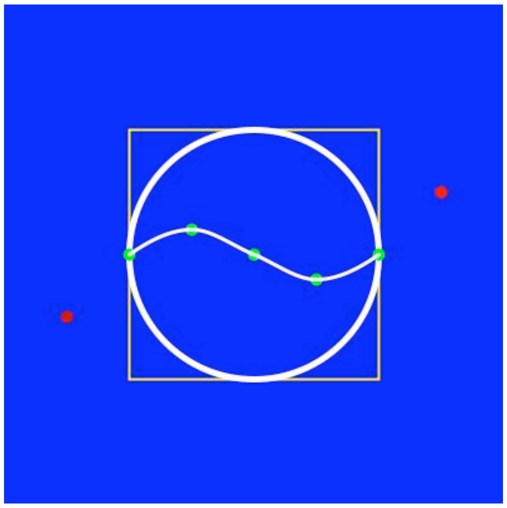
Using this for HW2

You need to do some planning for this. First, you need to explore the four functions beginShape( ), vertex( ), curveVertex( ), and endShape( ). Look at Jim's class code for today. Run the code, modify the code, and get a feel for what happens.

Then return to this.

To draw useful figures (ones that we can easily move and resize) we start with the idea of an anchor point. A specific (x, y) point in the figure from which all other points in the figure can be computed using arithmetic expressions composed with the +, -, * and / operators and variables.

In the class code, Jim draws the letter theta.



The red and green dots and the yellow rectangle are for illustrative purposes. In the final version of this, they will not be drawn.

His strategy is to establish an imaginary rectangle within which the letter will be drawn. The yellow rectangle is the imaginary rectangle. To specify this rectangle, Jim uses two variables (x and y) to locate the upper left corner.

He also declares two variables to store the values of the width and height of the rectangle:

```
//  This declares the four variables of type float.
float x, y, wd, ht;

//  This assigns values to the four variables.
x = 100;
y = 75;
```

```
wd = 120;
ht = 200;
```

He uses these four variables to locate the points and width and height for the ellipse that make up part of the letter theta:

```
        ellipse( x + wd * .5,     y + ht * .5,    wd,    ht );
```

Look very carefully at the arithmetic that represents the x and y coordinate position. The anchor point of the circle in this code is the center of the ellipse[2].

Jim's code computes the x value of the center of the ellipse as x + wd * .5 which is (x pixels) plus (one half of the width of the ellipse pixels) from the left edge of the frame.

The code computes the y value of the center of the ellipse in a similar fashion.

Be sure you understand what this code does.

Next he uses the variables and the + and/or the * operator to locate the points of the curve that makes up the center line of the letter theta. This code does the work. This code tells Processing that a shape is being drawn and that the points (vertices) between the beginShape( ) and endShape( ) define the location of the shape.

If we use the vertex( ) function, the shape will a polygon which has straight lines for the edges. Here the curveVertex( ) function is being used and this tells Processing that we are defining a curved line.

```
beginShape( );
   curveVertex( x - ( .25 * wd   ),    y + ( .75 * ht)    );
   curveVertex( x                ,    y + (  .5 * ht)    );
   curveVertex( x + ( .25 * wd)  ,    y + ( .25 * ht)    );
   curveVertex( x + (  .5 * wd)  ,    y + (  .5 * ht)    );
```

[2] This is the default position of the anchor point. We can alter this to the upper left corner of the rectangle enclosing the ellipse using the `rectMode(CORNER)` function and argument. Note that all ellipses drawn after this function is called use the upper left corner for the anchor point. We can reset the anchor point back to the original (or default) value with this function call: `rectMode(CORNER)`.

```
    curveVertex( x + ( .75 * wd)   ,     y + ( .75 * ht)     );
    curveVertex( x + wd            ,     y + (  .5 * ht)     );
    curveVertex( x + (1.25 * wd)   ,     y + ( .25 * ht)     );
endShape( );
```

The spacing used above is just to make it easier for you to read. Let's look at the first curveVertex( ) function call – specifically the first parameter:

```
      curveVertex( x - ( .25 * wd   ),    y + ( .75 * ht)    );
```

Processing evaluates the expression that is used as the parameter. It looks up the values of the variables x and wd and converts the expression to:

```
     curveVertex( 100 - ( .25 * 120   ),    y + ( .75 * ht)
```

This evaluates further to:

```
    curveVertex( 100 - ( 30  ),    y + ( .75 * ht);
```

and then to:

```
   curveVertex( 70,    y + ( .75 * ht);
```

It then evaluates the second parameter in exactly the same maker resulting in:

```
    curveVertex( 70,    225);
```

The other function calls of curveVertex( ) are executed in the same manner.

The use of the variables and the arithmetic operators in the parameter list means that we can move and resize the letter theta with only a few key strokes.

Two more additional topics:

For the curveVertex( ) function there is no line drawn between the first two points and the last two points. The first point and the last point are "reference points" or "control points" that we can use to adjust the overall shape of the curve. *If you build a curve with only three calls to curveVertex( ), noting will be drawn.*

Polygons work very similarly to curves. The vertex( ) function must be used within the beginShape( ) – endShape( ) functions. There is more so you will need to look up vertex( ) in the API. Given this fantastic mumbling about curves, we leave polygons for you to figure out…

IMPORTANT!!

The rect and ellipse shapes have what Processing calls a "mode'. The mode value tells processing where the (x, y) anchor point is for the figure.
The default rectMode allows you do specify the rect's location using the upper left corner of the rectangle.

The default ellipseMode allows you to the ellipse's location using the center of the ellipse.

If it makes your work easier, you can change the modes for both. Read the Processing API entries for the functions ellipseMode( ) and rectMode( ). They are in the Attributes subset in the center column. You may find that changing the mode makes your arithmetic easier.

Final Words - important.

In the class code for today, Jim used the upper right corner of a rectangle as the anchor point and the width and height of the rectangle to compute the location of the points of the curve.

Homework #2 MANDATES that you use the center of a circle as the anchor point and the diameter of the circle for computing the location of the points to determine the curves that form your initials.

The process you must follow for your code in homework #2 is parallel to the process Jim used in his code for today. If you understand what he is doing in the class code for today, you should be able to write the code for homework #2.

If you do not understand the arithmetic and how it was developed in the class code, you must get some help before you start coding your solution to homework #2.