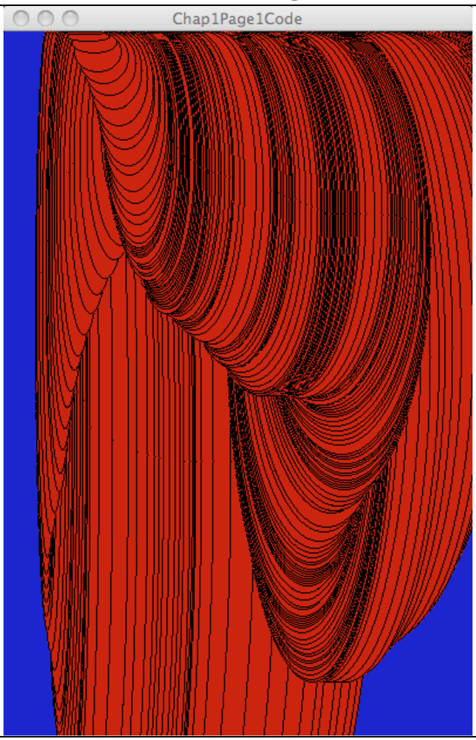
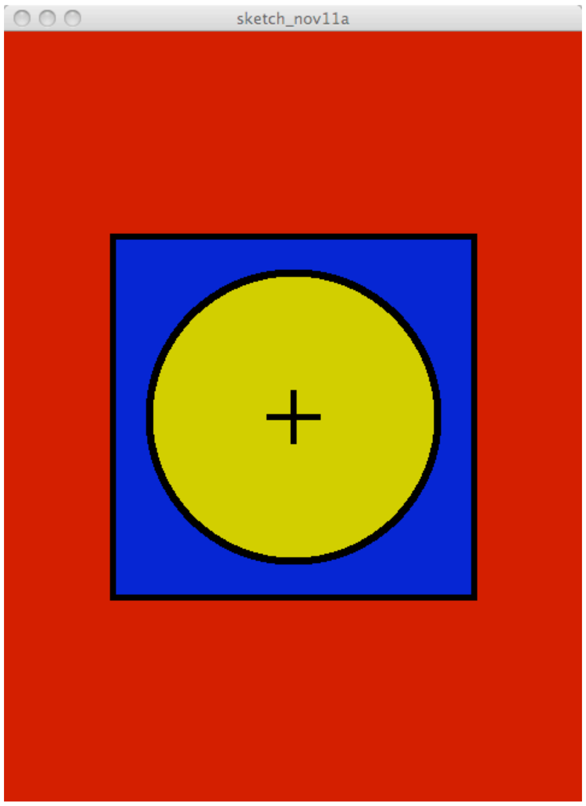
The Junior Woodchuck Manual of Processing Programming for Mobile Devices

The Image	*The Code*

 Chap1Page1Code	
	```
	void setup( )
	{
	  size( 400, 600 );
	  background( 0, 0 , 200 );  // blue
	  fill( 200, 0, 0 );                //red
	}

	void draw( )
	{
	   ellipse( mouseX,   mouseY,
	                mouseX/2, mouseY*2 );
	}
	```

Chapter 4
Finding our Way Home

The Image	*The Code*
	see listing below

```
float x, y, w, h;

void setup( )
{
  size( 480, 640 );
  x = width/2;
  y = height/2;
  w = 300;
  h = 300;

  rectMode( CENTER );
      // shade of red
  background( 200, 0, 0 );
}

void draw( )
{
    // shade of blue
  fill( 0, 0, 200 );
  rect( x, y, w, h );

        // shade of yellow
  fill( 200, 200, 0 );
      // shade of blue
  ellipse( x, y, w*0.8, h*0.8 );

      // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );
}
```

First, A Brief Review…

We have covered a lot of new stuff in a very short time. So before we start some new stuff, let's review what you went over last week but with new code. See if you can match what you learned last time with the code below.

The code below is the code that is on page 1 of this exciting chapter. Let's walk through it for the review.

Code	*Its Purpose*
`float x, y, w, h;`	*These are "our" variables. They are sometimes called "global" variables because they can be used throughout our code. This line of code tells Processing that we want to reserve four pieces of memory to store four decimal values and that we want to name them x, y, w, h.*

`void setup( )`
`{`
	This is our setup( ) function that is somewhat special because we write it BUT Processing uses it. We must NEVER use this function. If we do, the results are unpredictable. Who knows, your computer might melt… just kidding.

`size( 480, 640 );`	*Here we call the size( ) function and tell Processing to make the window 480 pixels wide and 640 pixels high. Why 480 and 640? Who knows – it just popped into your teacher's mind like magic.*

`x = width/2;`
`y = height/2;`
`w = 300;`
`h = 300;`
	This code tells processing what values to give our variables. Without this, their values are zero. The value of x is half of the width that is 480 (look a the arguments for the size function) and the value of y is half of the height that is 640.

`rectMode( CENTER );`	*This changes the anchor point of any rectangle we draw from the upper left corner to the center of the rectangle.*

`// shade of red`
`background( 200, 0, 0 );`
	The two "//" characters make this a comment.
	This sets the background color to a shade of red.

`}`	*This is the end of our setup( ) function.*

`void draw( )`
`{`
	- This is our draw( ) function. Like our setup( ) function, our draw( ) function is also somewhat special because we write it and Processing uses it. We must NEVER use this function. If we do, the results are unpredictable.

`// shade of blue`
`fill( 0, 0, 200 );`
	Another comment
	This calls the fill() function to set the fill color to a shade of blue.

`rect( x, y, w, h );`	*This calls the rect() function to draw a rectangle with the **center** of the rectangle x pixels from the left edge, y pixels from the top. The rectangle will be w pixels wide and h pixels high.*

`// shade of yellow`
`fill( 200, 200, 0 );`
	Another comment
	This calls fill() to set the fill color to a shade of yellow.

`ellipse( x, y, w*0.8, h*0.8 );`	*This calls the ellipse function to draw an ellipse that has its center x pixels from the left edge and its height y pixels from the top edge. The ellipse will have a computed width that is 80% of the value of the variable w and 80% of the value of the variable h.*

`// black`
`stroke( 0 );`
	Another comment
	This calls the stroke( ) function to set the color of any future line to black.

`strokeWeight( 5 );`	*This calls the strokeWeight() function to set the width of the lines to 5 pixels.*

`line( x- 20, y, x+20, y);`	*This calls the line() function. The arguments are the (x,y) coordinates of both ends of the line. The first pair of coordinates:* **line(x- 20, y, x+20, y);** *place one end of the line:* - *220 pixels from the left edge which is computed by subtracting 20 from the variable x which has a value of 240.* - *320 pixels from the top edge which is the value of the variable y.* *The second pair of coordinates:* **line(x- 20, y, x+20, y);** *determines the other endpoint of the line. Using the logic in the explanation above, see if you can compute this point's location:* *Answer is in the foot note:*[1]
`line( x, y-20, x, y+20 );`	*See if you can compute the endpoints for this line. Answers are in the footnote.*[2]
`}`	*This is the end of the draw( ) function.*

If you made it through all of that, it should help you reduce your errors and you can get things to look the way you want them to faster.

Next, A New Kind of Function…

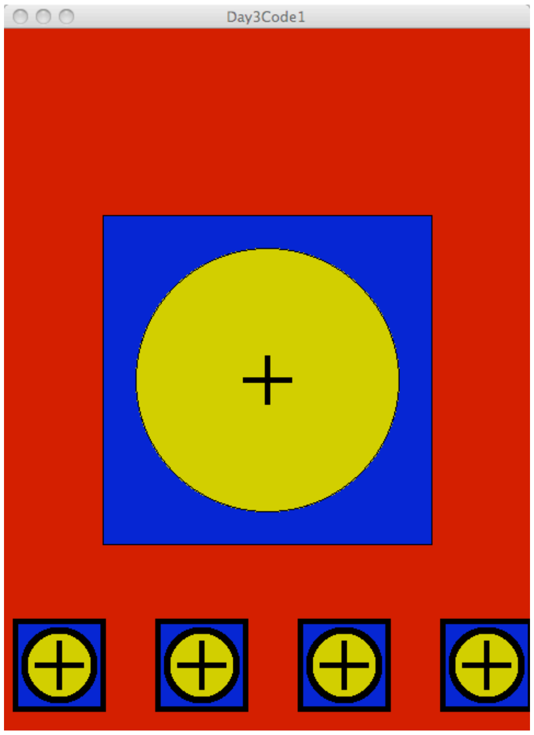
We have seen two kinds of functions:

- One kind is the set of functions that Processing writes and we use such as rect( ) and fill( ).
- A second kind contains the functions we write and Processing uses. We have seen two of these: setup( ) and draw( ).
- A third and new kind are functions that we write and we use.

We will use this third kind today. Next week we will add a fourth kind of function.

[1] line(220, 320, 260, 320);
[2] line(240, 300, 240, 340);

This third kind of functions as we said are functions that we write and we use. You may be thinking, "why do we need these?". If we write functions that we use, our programs will be easier to plan, code, test, debug, and possibly modify. Without these functions, our code will get very difficult to work with. Want to see why? Read on...
Suppose we want to modify the code on page 1 of this chapter to draw five figures like this:



If we do this in the draw( ) function, we have to copy the code that draws the figure and paste it back into the program four more times - once for each of the four figures at the bottom of the screen.

Next we have to figure out where to put code that change the values of the variables so draw( ) can display the four new figures.

The resulting code would look like this:

```
void draw( )
{
  // Big Figure
     // shade of blue
  fill( 0, 0, 200 );
  rect( x, y, w, h );
         // shade of yellow
  fill( 200, 200, 0 );
       // shade of blue
  ellipse( x, y, w*0.8, h*0.8 );
       // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );

  // Preparation for small figures
  w = 80;
  h = 80;
  y =  580;

  // Small Figure - farthest left
  x = 50;
       // shade of blue
  fill( 0, 0, 200 );
  rect( x, y, w, h );
         // shade of yellow
  fill( 200, 200, 0 );
  ellipse( x, y, w*0.8, h*0.8 );
       // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );

   // Small Figure second from left
  x = 180;
        // shade of blue
  fill( 0, 0, 200 );
  rect( x, y, w, h );
       // shade of yellow
  fill( 200, 200, 0 );
  ellipse( x, y, w*0.8, h*0.8 );
       // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );

    // Small Figure third from left
  x = 310;
  fill( 0, 0, 200 );
  rect( x, y, w, h );
        // shade of yellow
  fill( 200, 200, 0 );
       // shade of blue
  ellipse( x, y, w*0.8, h*0.8 );
       // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );

   // Small Figure fourth from left
  x = 440;
        // shade of blue
  fill( 0, 0, 200 );
  rect( x, y, w, h );
         // shade of yellow
  fill( 200, 200, 0 );
  ellipse( x, y, w*0.8, h*0.8 );
       // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );
}
```

In order to draw five figures, our new code in the draw( ) function grows to four times the original size. This can get very confusing. If we do not add comments, it can become impossible to work with. And what if we want to draw four more figures on top and then four down each side????

There is a different and better solution. We can write our own drawFigure( ) function (similar to Processing's functions) to draw the figure. If we do this, our code in the draw( ) function changes to this:

```
void draw( )
{
 // Big Figure
 drawFigure( );

 // Preparation for small figures
 w = 80;
 h = 80;
 y =  580;

 // Small Figure - fartherest left
 x = 50;
 drawFigure( );

  // Small Figure second from left
 x = 180;
drawFigure( );

   // Small Figure third from left
 x = 310;
drawFigure( );

   // Small Figure fourth from left
 x = 440;
drawFigure( );
}
```

We have to do one more thing – we have to write the function figure( ) so Processing knows how to draw the figure. The code looks like this:

```
void drawFigure( )
{
     // shade of blue
  fill( 0, 0, 200 );
  rect( x, y, w, h );
         // shade of yellow
  fill( 200, 200, 0 );
  ellipse( x, y, w*0.8, h*0.8 );
       // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );
}
```

Physically, this function looks like our setup( ) and draw( ) function. The main difference is that Processing uses the setup( ) and draw( ) functions. BUT it will not use our drawFigure( ) function unless we tell it to do so.

You may be wondering about the first word, void. Since we only have about six hours total to work on this stuff, let's just say that putting this word here is a rule we have to follow. Talk to us in week 6 and ask us about it and we wil explain it then. If you leave it out, Processing will not compile and run your code.

The drawFigure( ) function must have the four variables we declared or it will not work properly.

We usually write our drawFigure( ) function after or below the draw( ) function but that is not a rule.

For the rest of this course we are going to insist that you write your own functions like this. It will make it much easier for you in the coming classes.

The next page shows the entire program so you can see how it is organized.

```
float x, y, w, h;

void setup( )
{
  size( 480, 640 );
  x = width/2;
  y = height/2;
  w = 300;
  h = 300;

  rectMode( CENTER );
       // shade of red
  background( 200, 0, 0 );
}

void draw( )
{
  // Big Figure
  drawFigure( );

  // Preparation for small figures
  w = 80;
  h = 80;
  y =  580;

  // Small Figure - fartherest left
  x = 50;
  drawFigure( );

   // Small Figure second from left
  x = 180;
 drawFigure( );

     // Small Figure third from left
  x = 310;
 drawFigure( );

     // Small Figure fourth from left
  x = 440;
 drawFigure( );
}

void drawFigure( )
{
     // shade of blue
  fill( 0, 0, 200 );
  rect( x, y, w, h );

         // shade of yellow
  fill( 200, 200, 0 );
  ellipse( x, y, w*0.8, h*0.8 );

       // black
  stroke( 0 );
  strokeWeight( 5 );
  line( x- 20, y, x+20, y);
  line( x, y-20, x, y+20 );
}
```

And Now… Adding Animation to Our Code

(Yes, we did this last week but we are going over it again…)

You may not remember this from last week but there is something special about the draw( ) function.

The draw( ) function is being executed or run by Processing about sixty times each second. And the window is redrawn or refreshed after each execution of the draw( ) function. Each execution of the draw function produces what we call a frame. This frame is displayed in the window that we see when we run our programs. Unless we color the background of the frame or draw a large rectangle that covers the entire frame, the frame is transparent. We can see the previous frames "through" the new frame.

The reason we did not see this is initially is because the draw( ) function was drawing the exact same thing over and over sixty times each second.

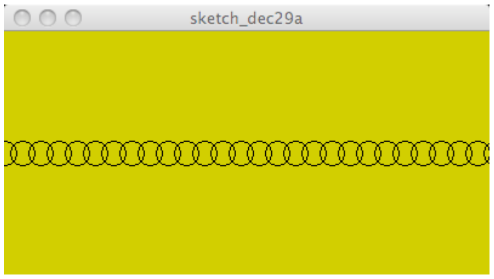
Let's revisit the demonstration. Here is some code.

```
float x;

void setup( )
{
  size( 400, 200 );
   noFill( );
   x = 0;
   background( 200, 200, 0 );  // yellow
}

void draw( )
{
  ellipse( x, 100,  20, 20 );
   x = x + 15;
}
```

which eventually draws this on the window:



This is the result of the ellipse or circle being drawn over and over again. However, each circle has a different value for x.

Look at the code in draw( ) again – it is shown below:

```
void draw( )
{
  ellipse( x, 100,  20, 20 );
  x = x + 15;
}
```

Look at the purple code. This is NOT an algebraic statement as you may have learned in math class. This is a programming statement that changes the value of x.

This code is read in English as:

The ***new*** value of x is the ***current*** value plus 15.

Because Processing is repeating the draw( ) function over and over again, it draws the ellipses fifteen pixels apart.

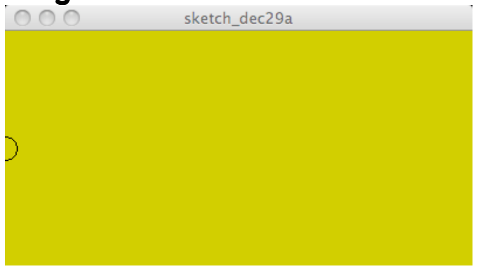
We will write other code similar to this in class but it does not work too well in the printed page like this so you will have to wait a bit. . . Sorry. . .

Moving on…

Each execution or iteration of the draw( ) function results in a new picture or image that is called a frame – just like a frame of a movie.

Unless we tell Processing to do something different, the new frame is draw on top of the old frame so we can see any part of the old frame or frames that the new frame does not cover.

The first frame Processing shows us looks like this:



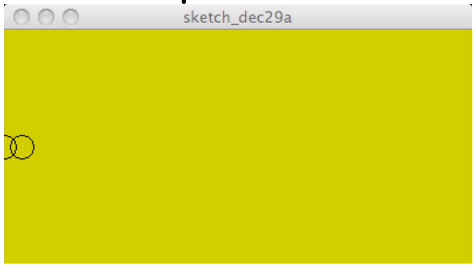
We see only one circle. The value of x is zero so the circle is centered zero pixels from the left edge.

As soon as the ellipse is drawn, the value of x is changed to 15 by this line of code:

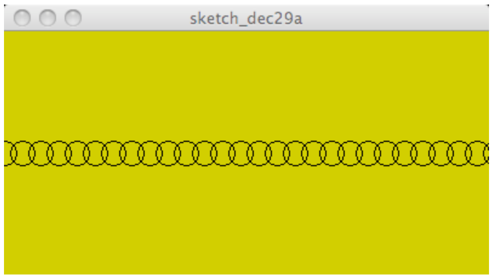
```
  x = x + 15;
```

That is the end of the first execution of the draw( ) function and Processing displays this frame.

Then it starts the execution of the draw( ) function a second time. It starts a new, transparent frame and draws the circle 15 pixels from the left edge. Since we have not told Processing to do anything different, the new frame is transparent. Once the second ellipse is drawn, the value of x is changed to 30 by the next line of code. This finishes the execution of the draw( ) function so Processing displays the new, transparent second frame over top of the first frame and we see:



This process repeats itself about sixty times each second so we eventually see this:



Again, there are other interesting things we can do with this but it is useless to try to show them here so we will do that in class.

Finally… Controlling Our Animation

What happens when the circles reaches the right side of the window? Processing does not stop drawing it. It just draws it off screen to the right.

Processing does not let us scroll the window so we cannot see this.

However, what if we do not want to "lose" the ellipse when it reaches the right side of the window? What if we want to return it back to the left side? Can we do that?

The answer is, "yes" or we would not have asked it.

The code that moves the circle from the left edge of the window to the right edge is this:

```
  x = x + 15;
```

The variable, x begins with the value zero, Every execution of the draw() function adds 15 to its value. The window is 400 pixels wide because of this code:

```
  size( 400, 200 );
```

Processing stores the value of the first argument, which is 400 in its variable named width. It stores the value of the second argument (200) in its variable name height. We can use these but we must never change them.

So when the value of X becomes larger than 400, the circle will be drawn too far to the right to be seen.

What we need to do is change the value of x when its value is greater than 400.

And that is not very difficult to do.

BUT FIRST A WORD FROM OUR SPONSOR…
THE RELATIONAL OPERATORS

In the last chapter, you were introduced to Processing's version of addition, subtraction, multiplication, and division and their operators:

```
   +   -   *   /
```

These are great and very useful.

Processing has a similar set of operators that we can use to make our animations more interesting. These are the relational operators. You may or may not have studied them in your math class. If you have not, that is ok. We will explain it to you.

Programmers, programming, and Processing use the relational operators to compare the values of variables like X to other values. You have done this for a long time. You have compared your age to others. We compare the number of wins the Steelers have to the number of wins Baltimore has. What we need to do is to compare the value of x to the value of the right side of the window.

Here are two of the six relational operators – these are all we will need:

<	less than	or	is smaller than
>	greater than	or	is larger than

There are four more relational operators but that is for another day…

OK -- how do we use these???

We use these is in code that asks a question about the variables.
What we need to ask is,
"Is the value of the variable x greater than the width of the window?"

Processing has a way to ask this question. It is called a "control structure". The control structure is the if. The if looks like this:

```
if (  )
{

}
```

Note that there are NO semicolons in this code . . . yet . . .

The relational operators are used inside the parentheses like this:

```
if ( x > width  )     // remember that Processing stores the width of
{                     // the window in its variable, width

}
```

We read this code in English like this:
"If x is greater than the width of the window?"

Programming languages like Processing answer this type of question with only one of two possible answers:

true
false

The way the if works is this way:

- if the answer is true, Processing does whatever code we put inside the { }.
- if the answer is false, Processing skips whatever code we put inside the { }.

The { } are called different things by different programmers. Some programmers call them "curly brackets" and others call them "curley braces". Just braces is probably the best name.

So what do we put inside the braces?

We put what we want Processing to do when the answer is true.

This whole thing was started to keep the circle on the screen and not let it move off the right side. The window is 400 pixels wide. As long as the value of x is less than 400, we will see the circle. It is only when the value of x is greater than 400 that we need to change the value of x back to zero. So we could write the code like this:

```
if ( x > width  )
{
   x = 0;
}
```

As long as x is less than width, the code inside the braces will be skipped and our circle will move from the left side to the right side of the window.

The first time x has a value greater than width, x will have its value changed back to zero.

This type of motion is called wrapping. The circle "wraps" from the right side to the left side.

Notice that we can do anything we want to inside the braces:
- we could change the fill color
- we could change the size of the circle
- we could change the y value of the circle to move it down or up

We can do anything we want to do as long was we are happy with the results.

Question: If we alter the y value, the circle eventually disappears.

How can we prevent this?

Give it a try…

Next time we will figure out how to bounce the circle off the edges of the window. . .

Fun stuff from last time:

Putting a picture in your program.

Follow these steps exactly as they are listed:

1. Save the file you are writing.
2. Fine the folder that has the .pde file in it – it has the same name as the file
3. Using the right click or control/click, make a new folder.
4. Name the folder, data -- the name must be all lower case
5. Put the picture file into the data folder. Processing can display pictures with these extensions: **gif**, **.jpg**, **.tga**, and **.png**
6. At the top of your program create a variable like this:
   `Pimage   picture1;`

7. In the setup( ) function after the call of the size( ) function add this line of code:
   `picture1 = loadImage( "exactNameOfYourPictureFile.extension");`

   The name of your picture file in the data folder goes inside the quotation marks. The name must be exactl the same as the file name including the case of the letters. Check to see if the extension is jpeg or jpg – it is important.

8. To display the picture, add this line of code in the draw() function:
   `image( picture1, 100, 100, 60, 70 );`

9. Go to the API and look up PImage and image to find out what the arguments stand for.