Processing Notes

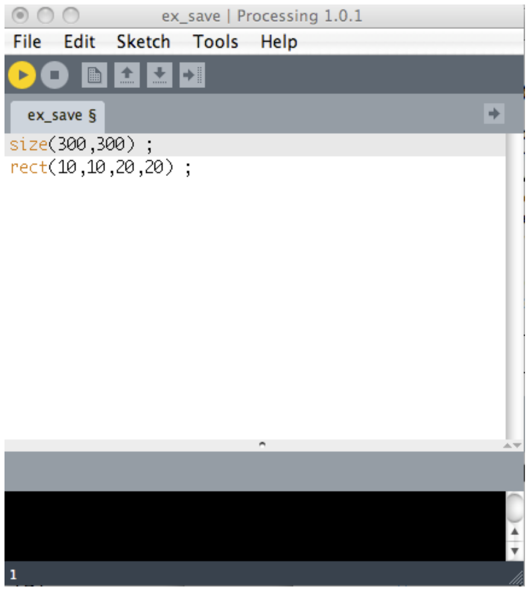
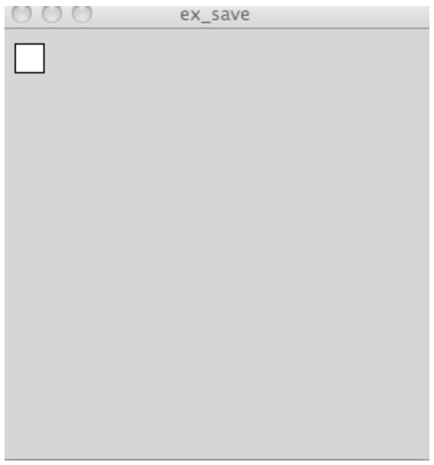
Chapter 1: Starting Out

We will start our journey into Processing with creating static images using commands available in Processing:

rect( )
line ( )
ellipse()
triangle()

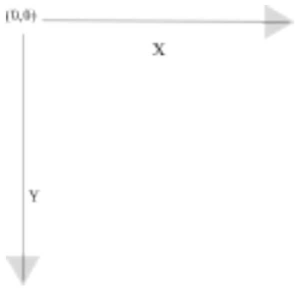
NOTE: to find out more about these, and most things in Processing, select Help -> References from the menu at the top of the Processing Window. Then click on the command you want to look up, for example click on "rect()" to see how the rect() command is used. When you click on it you will see that rect() takes 4 arguments: rect(x,y,width,height), and it will draw a rectangle with one corner at location x,y and then be of size width x height.

Put the following two lines of code in the Processing code window as show in the table below on the left, hit the play button, and you get output as show in the right:

```
size(300,300) ;
rect(10,10,20,20) ;
```

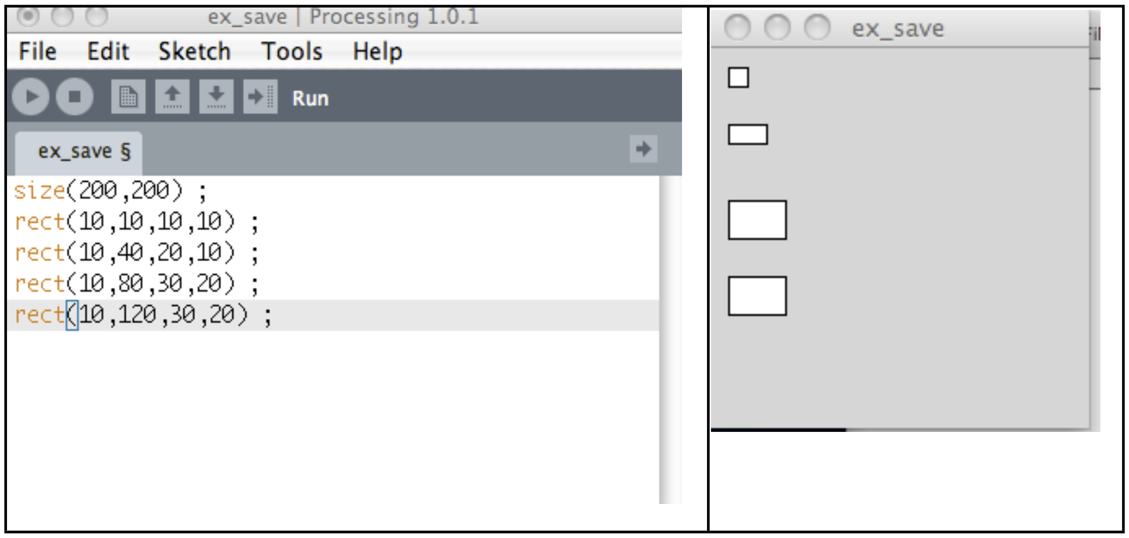
Note, that each line MUST be terminated with a semicolon. The size command will specify the size the window created., in this case 300 pixels wide by 300 pixels high. the command: rect(10,10,20,20) creates a rectangle at location (10,10). You may be wondering why that rectangle is in the upper left corner if it is at location (10,10). The answer is the **coordinate system may be different than you are used to**. In Processing, and in many other graphics languages, the point (0, 0) is in the upper left hand corner. X increase to the right as you are used to, but **Y increases going DOWN**, not going up.



For a more thorough explanation see:

http://processing.org/learning/drawing/

As another example, the following code produces the following output:

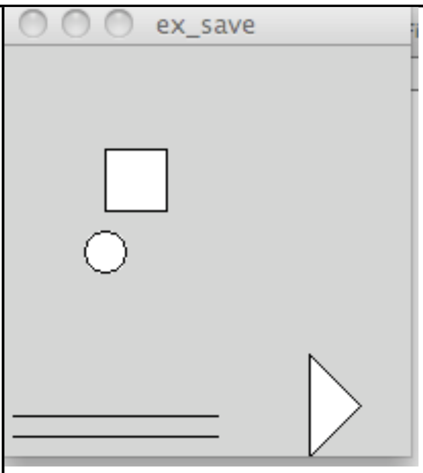


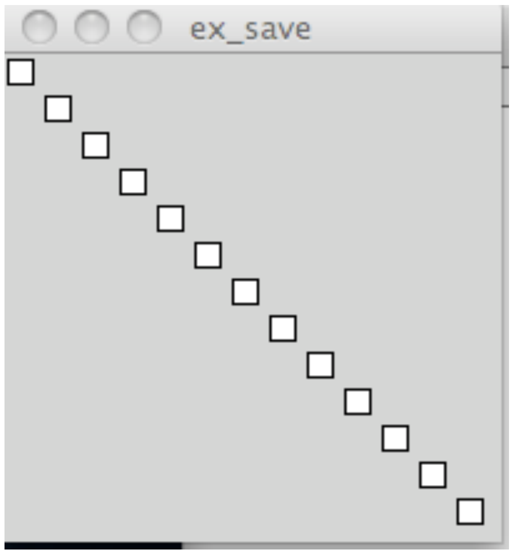
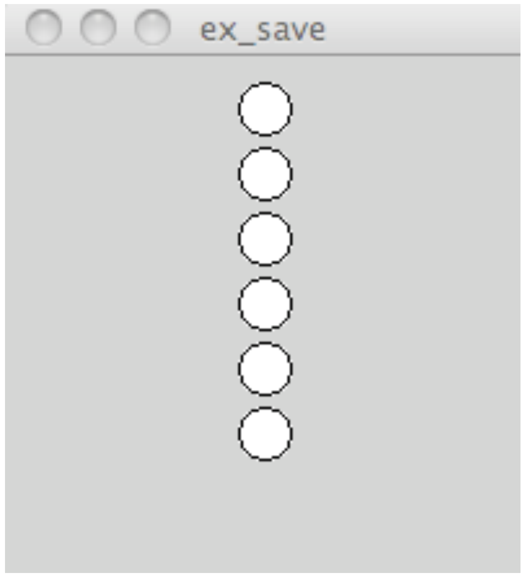
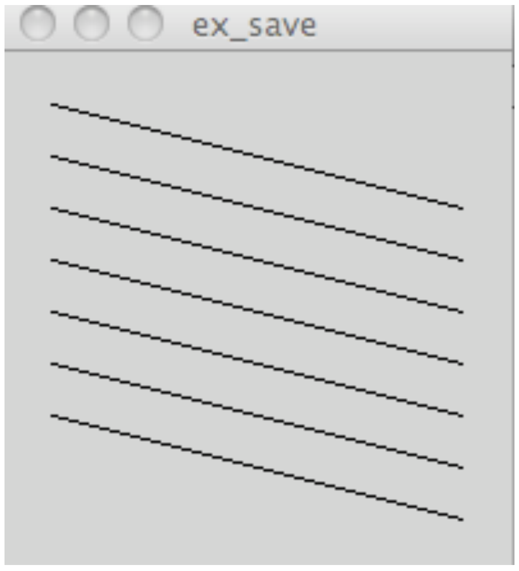
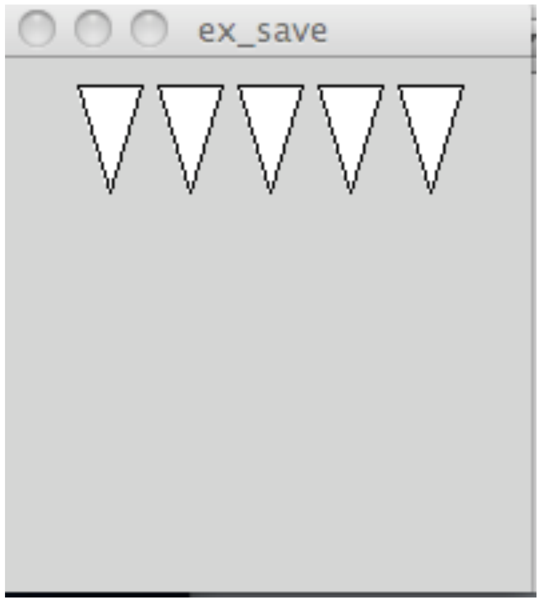
```
size(200,200) ;
rect(10,10,10,10) ;
rect(10,40,20,10) ;
rect(10,80,30,20) ;
rect(10,120,30,20) ;
```

The code shows that 4 rectangles are being created at locations (10,10), (10,40), (10,80), and (10,120). Notice also the size of the rectangles differ, the first is 10x10, the second is 20x10, the third is 30x20, and the last is also 30x20.

Here is an example of using each of the rect, ellipse,triangle, and line commands:

```
size(200,200) ;

rect(50,50,30,30) ;
ellipse(50,100,20,20) ;
triangle(150,150,150,200,175,175) ;
line(5,180,105,180) ;
line(5,190,105,190) ;
```



EXERCISE 1A Write four sets of code to create each of the following images:	
	
	

Color and line types:

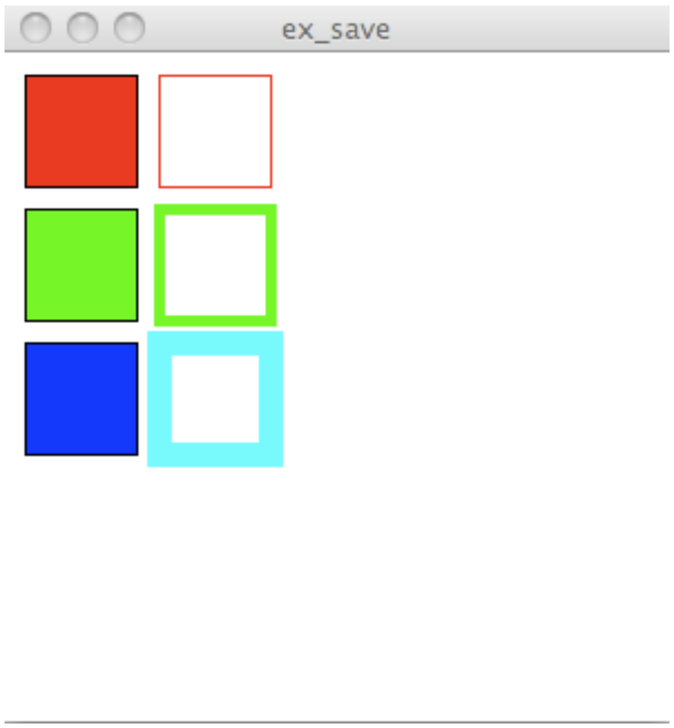
Processing allows you to use color for fills, background, and lines/borders, as well as creating different line types. Colors in Processing, and many other graphic languages, are composed of three components Red, Green, Blue. The intensity of each component is expressed as a number from 0 to 255, with 0 being none and 255 being the maximum. This is different than studio art color theory where colors are composed of the primary colors red, yellow and blue. The difference is that computer screen create color by mixing colored lights whereas in painting you are mixing physical pigments. To change the background color, the default is grey, you use the background command:

```
background(255,255,0) ;
```

This command sets the maximum amount of red, the maximum of green, and the minimum of blue. You will get a strong yellow background. If you do (255,0,0) you will get a strong red background. If you use smaller numbers you get less "light" and hence darker colors. For example if you do (100,0,0) you get a darker shade of red, and (50,0,0) even darker. Note, the value (255,255,255) will give you white. Also, smaller but equal numbers such as (200,200,200) will give you different shades of grey. Further note, if you use just one number you get grey scale: background(255) ; will give you white, background(0) will give you black, and the numbers in between give you different values of grey.

You can also change the color of 2D Primatives (rect, triangle, ellipse, etc) by using the fill() command: fill(255,0,0) will make subsequent 2d primatives filled in red. When you execute the fill command it holds for all subsequent 2d primative calls. You can also modify the thinkness of the border (or of lines) by using the strokeWeight(), the larger the number in the call to strokeWeight() the thicker the border line. You can also change the color of the border line using stroke(). For example, stroke(255,0,0) will make the border color red. If you make the fill the same color as the background, then you can see just the border. The following code produces the following image:

```
size(300,300) ;
// set the background to white
background(255) ;

fill(255,0,0) ;
rect(10,10,50,50) ;
fill(0,255,0) ;
rect(10,70,50,50) ;
fill(0,0,255) ;
rect(10,130,50,50) ;

// Set the fill to white, all following
// rectangles will appear empty
fill(255) ;

stroke(255,0,0) ;
rect(70,10,50,50) ;

stroke(0,255,0) ;
strokeWeight(5) ;
rect(70,70,50,50) ;

stroke(0,255,255) ;
strokeWeight(10) ;
rect(70,130,50,50) ;
```



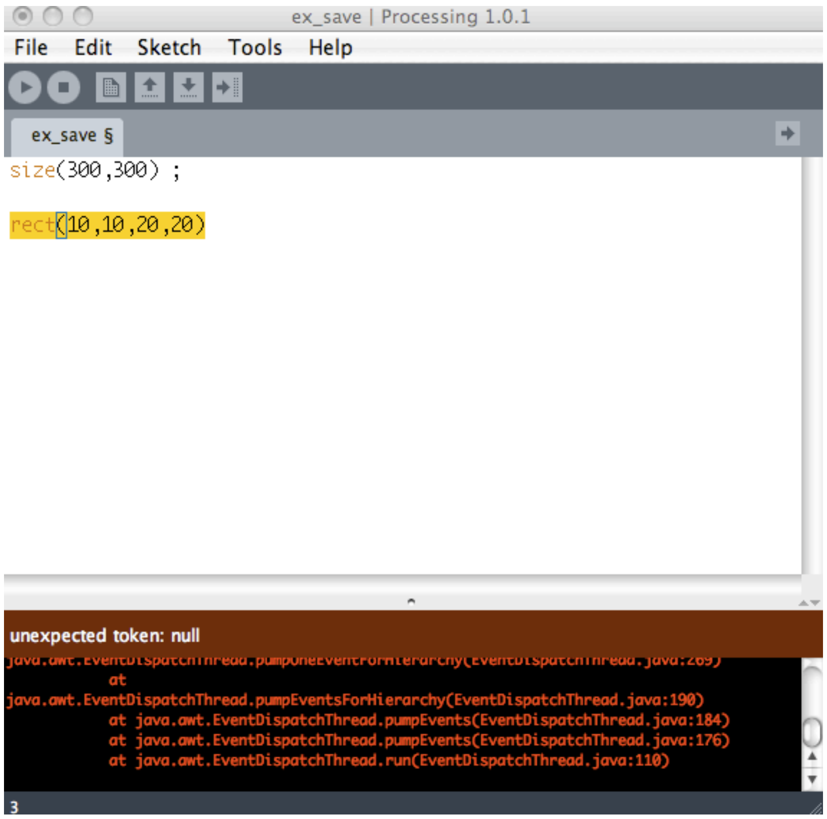
To read more about colors and see how to "mix" colors google "RGB color picker" or simply "RGB". You can find a lot of cool info that will explain this more thoroughly, for those that want to know. There is also an excellent page about color in Procesing by Daniel Shiffman at:

http://processing.org/learning/color/

Comments and Errors

In the code above you will notice some funny looking lines that start with two slashes "//". The double slash signifies the start of a comment. A comment is something the programmer adds to code to help explain the code to others, or to themselves when they look at the code later. In computer science commenting code is very important. In fact, in large software development projects there is often more lines of comments then code!

In any text-based programming language it is easy to make typos that are errors. When there are errors (also called "bugs") the code will not run. The process of removing errors is called "debugging". In Processing error messages appear in the bottom window. For example, consider the following. When you run it it does not work, pops up a whole bunch of gobbly-goob in the bottom window, and a more helpful message just above that says “unexpected token: null”.



```
size(300,300) ;

rect(10,10,20,20)
```

```
unexpected token: null
java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:269)
        at
java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:190)
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:184)
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:176)
        at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
3
```

It also highlights the offending line of code in yellow. The problem here is that the line does not end with a semicolon. If a semicolon is inserted and the code is re-run, then it will work. Consider another error, say I put in the following line of code:

rect(10,10,,20) ;

When I run it I’ll get an error, and it will say “unexpected token: , “ Again, Processing will highlight the offending line in yellow. In this case, there IS a semicolon, but, the rect( ) command is expecting 4 arguments: x, y, width, height, and I have only supplied 3.

In general, if you have an error message, look at it, it might be helpful, but almost always Processing will highlight the line where the problem is in yellow. Note, there may be multiple lines with problems, you need to fix them all. A final word of advice: do not type in 100 lines and then hit run, you will likely have many errors and with multiple errors it become harder to debug. Instead, add your code incrementally, running every now and then to make sure it is going like you expect.

Chapter 2: Variables

Variable are used to hold data during program execution. Variables can hold data of a certain type:

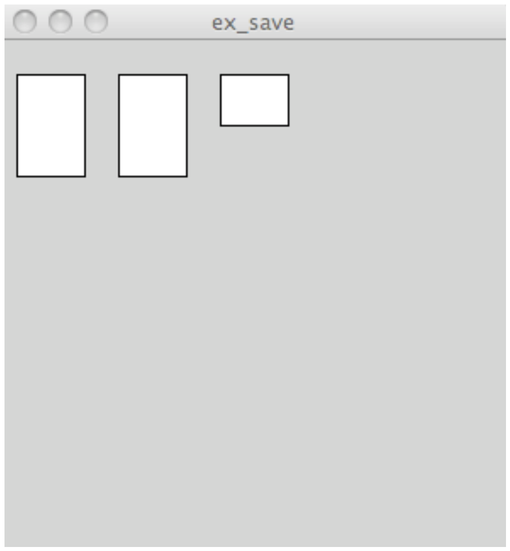
int	holds integers, i.e. whole numbers
float	holds decimal point numbers
char	holds a character

One declares a variable by stating the type of variable, the variable name, and then a semicolon. For example:

```
int numRectangles ;
```

would declare the variable "numRectangles" and it can only hold integers. The following code produces the following output:

```
size(300,300) ;

int x ;
int y ;
int rectWidth ;
int rectHeight ;

x = 10 ;
y = 20 ;
rectWidth = 40 ;
rectHeight = 60 ;

rect(x,y,rectWidth,rectHeight) ;

x = 70 ;
rect(x,y,rectWidth,rectHeight) ;

x = x + 60 ;
rectHeight = rectHeight / 2 ;
rect(x,y,rectWidth,rectHeight) ;
```



In the above code we declare four integer variable: x, y, rectWidth, and rectHeight. We then put values of 10, 20, 40, and 60 respectively into each variable. The statement:

```
x = 10 ;
```

should be read as "assign the value 10 to the variable x". The = sign means "assign to".

The statement:

```
rect(x, y, rectWidth, rectHeight) ;
```

creates a rectangle as normal, but rather then specifying the values for the rectangle directly with numbers, we use the names of the variables. When a variable is used, the value currently inside the variable is used. Hence, the first rectangle is drawn at location (10,20) with a width of 40 and a height of 60. One can change the content of a variable by assigning a new value to it. The statement:

```
x = 70 ;
```

replaces the current value of 10 in x with 70. Thus the statement rect(x,y,rectWidth,rectHeight) now creates a recantly at location (70,20) still with width and height of 40 and 60. The next two statments get the current value of x and rectHeight, manipulate them, and assign the new value from the manipulation back into the variable. In particular:

```
x = x + 60 ;
```

gets the current value of x (which is 70), adds 60 to it to get 130, and then assigns this value of 130 to x, replacing the previous value of 70. The statement:

```
rectHeight = recHeight / 2;
```

getst the current value of rectHeight (which is 60), divides it by 2 to get 30, and then assigns this value of 30 to rectHeight. Thus, the following statement rect(x, y, rectWidth, rectHeight) now draws a rectangle at location (130,10) with a width of 40 and a height of 30.

Note, variable creation and assignment can be done in one step. For example, the statements:

```
int x ;
x = 10 ;
```

can be done in one statement:

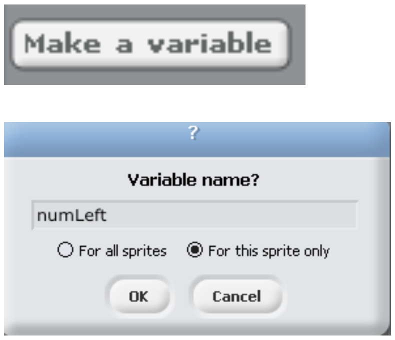
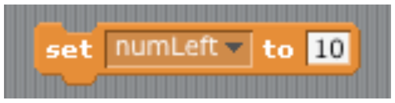
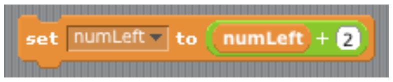
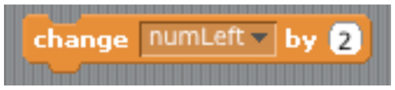
```
int x = 10 ;
```

Thus, the 8 variable declaration and assignment statements above can be changed to:

```
int x = 10 ;
int y = 20 ;
```

```
        rectWidth = 40 ;
        rectHeight = 60 ;
```

In the table below we show how to write code in Processing versus Scratch for those of you who know the Scratch programming language:

Processing	Scratch
int numLeft ;	 Make a variable; Variable name? numLeft; For all sprites [✓ For this sprite only]; OK; Cancel
numLeft = 10 ;	 set numLeft to 10
numLeft = numLeft + 2 ;	 set numLeft to numLeft + 2 or  change numLeft by 2

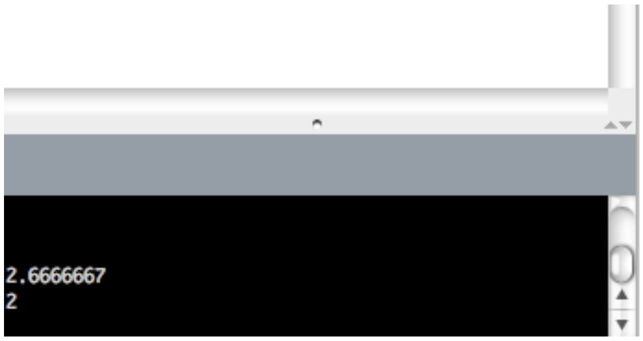
In Procesing if you want to see the current value of a variable you can print it out using the Java command: System.out.print(varName), where varName is the name of the variable. The value of the variable then gets printed out in the black box in the bottom of Procesing. For example:

```
int numLeft = 22 ;
System.out.println( numLeft ) ;
```

will print out the number “22” in the black box at the bottom of the Processing window.

If you use **float** variables you can get values that are not integers. The following code creates following output when run:

```
float fx, fy ;
int ix, iy ;

fx = 8 ;
fy = fx / 3 ;
System.out.println( fy) ;

ix = 8 ;
iy = ix / 3 ;
System.out.println( iy) ;
```



```
2.6666667
2
```

Notice the first number printed out is 2.666667, whereas the second is 2. float variables can hold real numbers, whereas int variables can only hold whole numbers. If you do a calculation on the right hand side that evaluates non-integer value and assign it into an integer variable in Processing it will just round it down to the whole number less than or equal to the floating point number.

Consider the following code:

```
int ix1, ix2 ;
float fx ;

ix1 = 2 ;
ix2 = ix1 ;
fx = ix1 ;

fx = 2.0 ;
ix2 =fx ;
```

The last line will cause an error. The error statement is: “cannot convert from float to int”. What this is saying is you cannot put a floating point value into an integer. You can think of the different types of variables as different shaped storage boxes: int variables can only hold integers, float variables hold real numbers AND they allow integers, but they become floating point values. Now in the example above, it seems you should be able to assign the contents of variable fx to variable ix2. Afterall, the value is 2.0, that is the same as 2, right? Mathematically, yes, but, remember, a float variable can hold any real number. When processing the commands, the compiler (the softwared that translates high level programming languages like Processing and Java into low level machine-executable code) just looks to see what is on the right side of the assignment operator and what is one the left. If the left is an integer, and the right contains a floating point variable or a constant with a decimal point, it flags the assignment as an error.

One can force the contents into an integer variable by doing **<u>type casting</u>**. An example is:

```
ix2 = (int) fx ;
```

this statment says force the contents of variable fx into an integer. It does this by truncating the decimal part.

A note about variable names. Variable names can be anything you want that is not a reserved word. Reserved words are words that are part of the Processing or Java language such as: { rect, ellipse, line, for, int, float, char, while, class, setup, draw, and a whole bunch more}. How do you know if a name is a reserved word? If you type it in Processing while highlight it in orange for you so you can see it is a reserved word, hence, you can not use it as a variable name. Variables can not contain a space, and they can not start with a number. So, the variables "num1" and "num_1" are legal, but "1num" and "num 1" are not. (The last one because it contains a space.)

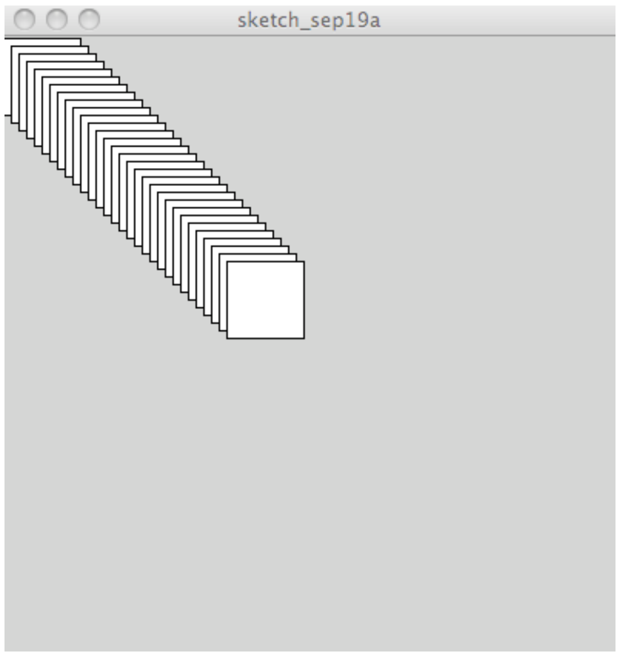
In general, use variable names that are descriptive of the contents. For example, if I am using variables to keep track of the location of a ball, I might use the variables: ballX and ballY, or ball_X and ball_Y.

Chapter 3: Simple Animations: Setup( ) and Draw( )

Up until now we have been creating static images in Processing. We will come back to working with static images, but for now we will see how to interact with the computer mouse and also create simple animations.

Consider the following code. The image on the right is a snapshot of the animation as it unfolds. When you run it there should be one rectangle in the top left corner, then, every 1/2 a second another rectangle should be drawn slightly to right and slightly lower than the previous rectangle.

```
int rx ;
int ry ;

void setup()
{
  size(400,400) ;
  rx = 1 ;
  ry = 1 ;
  frameRate(2) ;
}

void draw()
{
  rect(rx, ry, 50, 50) ;
  rx = rx + 5 ;
  ry = ry + 5 ;
}
```



code	explanation
int rx ; int ry ;	Declare two integer variables. We will use this to keep track of the x,y coordinate of the current rectangle to be drawn

void setup()
{
 	
}
	The setup **FUNCTION.** A function is a chunk of code that gets executed when it is called. The code that belongs to the function is everything between the open curly brace, { , and the close curly brace, } .
	
	The setup function is a special Processing function that is called exactly once at the beginning of a code run.

size(400,400) ;
rx = 1 ;
ry = 1 ;
	Initialize the screen size to be 400x400. Initialize the values of variables rx and ry to be 1.

frameRate(2) ;	This is built in Processing function that specifies how frequently they draw ( ) method is called. If set to 2, as in this example, then the draw( ) method is called twice per second.

code	explanation

void draw()
{
 	
}
	The draw function. This function is called by Processing repeatedly while the code runs. In here we put all the code that we want to do stuff over and over. The frequency of execution is determined by the frameRate( ) call, as describe in the row above, and usually found in the setup( ) function.

rect(rx,ry,50,50) ;	Draw a 50x50 rectangle at the location rx, ry, where rx and ry is the current value of those variables.

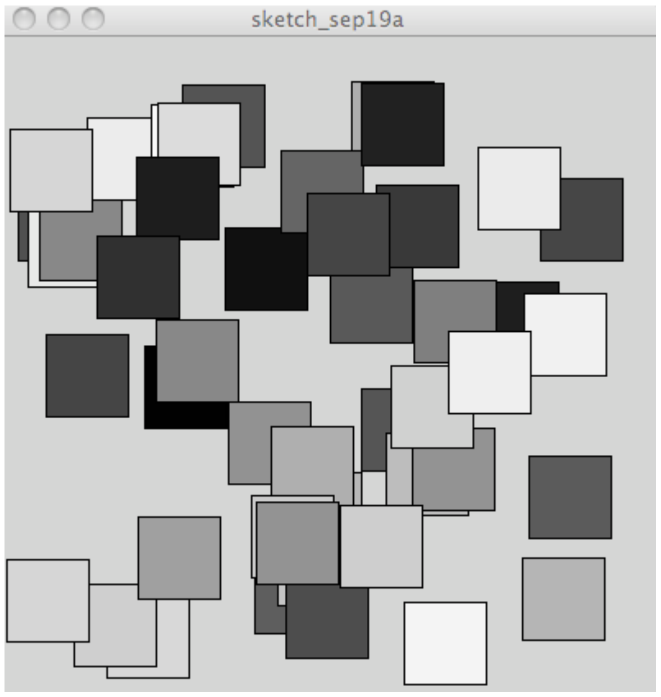
rx = rx + 5 ;
ry = ry + 5 ;
	Change the value of rx to be its current value plus 5. Same with ry.

Thus, the draw method is called twice per second. Each time it is drawn it increases the contents of variables rx and ry by 5, so, each time it is called it draws another rectangle 5 pixels over and 5 pixels down.

You will notice the word **void** before setup and draw. The word before the function is the **return type.** Functions can return values and hence we need to specify the type of value the fuction will return. For example, I can return a function that returns the average age of students in the classroom. such a function would return a float that contains the average, hence, instead of the word void you would say float. Void is use to signify that you do not care about the return type, it will not be uses. If this is confusing now, don't worry, just remember to put void before setup( ) and draw( ), and we will revisit this issue later when we explain functions in more detail.

Lets look at a new example:

```
int rx ;    // rectangle's x-coord
int ry ;    // rectangle's y-coord
int gscaleColor ;  //random color

void setup()
{
  size(400,400) ;
  frameRate(2) ;
}

void draw()
{
  rx = (int) random(1,350) ;
  ry = (int) random(1,350) ;
  gscaleColor = (int) random(255) ;

  fill( gscaleColor ) ;
  rect(rx, ry, 50, 50) ;

}
```



In this example we keep drawing rectangle as before, but this time the rectangle locations and grey-scale color are random. The image on the right show the output after 46 (no reason for choosing 46 other than it looked cool) rectangles have been drawn. We now use the Procesisng function **random( )** to generate random numbers. The line:

    random(1, 350) ;

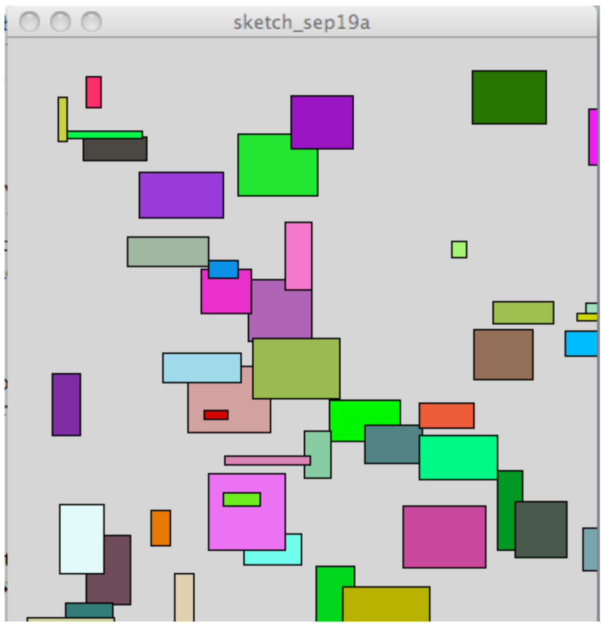
generates a random number between 1 and 350 inclusive. Note, the function actually returns a floating point value, and since we chose to use integer variables for rx and ry we need to cast the floating point value returned into an int using (int) before assigning it to rx and ry. Also note that we can call the random( ) function with a single parameter as in:

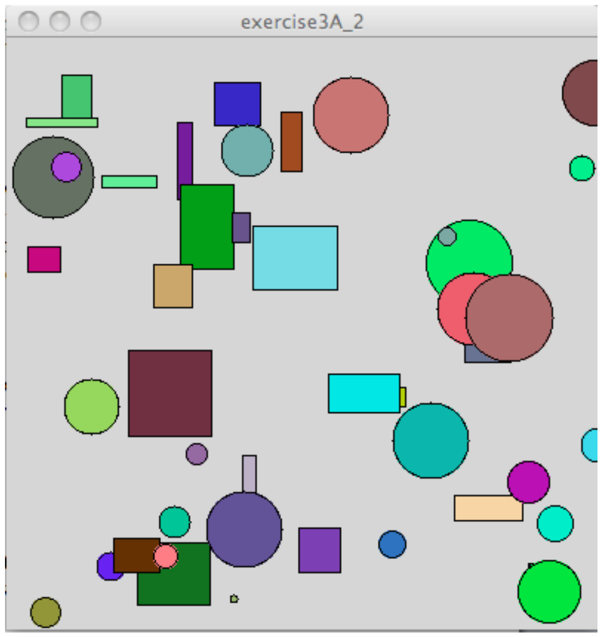
    gscaleColor = (int) random(255) ;

When called with a single parameter of 255 the function returns a random number between 0.0 nad 255.0 inclusive, which is cast into an integer to serve as a parameter to the fill( ) function.

Exercise 3A

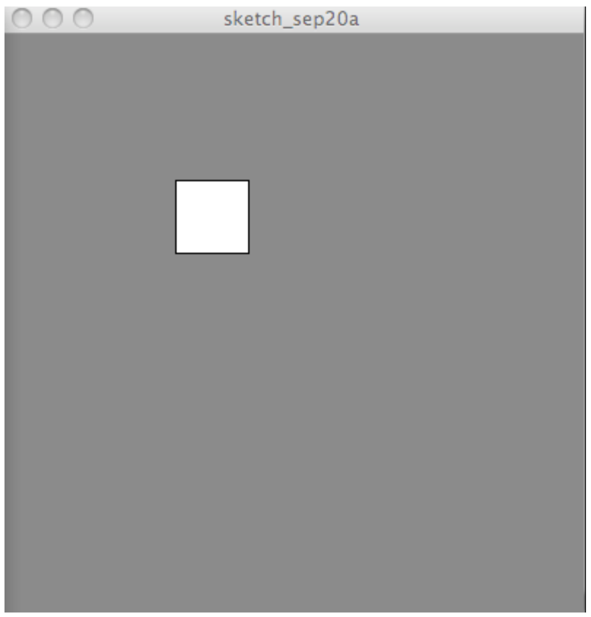
Write code to create something that looks like each of the following images (Note, if you printed this out in black&white, the images are multi-colored where each rectangle or circle has a different random color). Note, your code should draw these figures one polygon at a time using setup( ) and draw( ), the pictures below just show the state after 46 polygons have been drawn. Hint: use variables to hold random numbers for width, size, and R G B colors of each rectangle or circle.





You will notice that in all the examples so far the next rectangle is drawn and the previous one(s) remain. We just keep adding more. If you want to make an object move across the screen you need to wipe out it's previous drawing. This can be done with the **background( )** command. If you call background( ) inside the draw( ) function, then the window contents are erased each time when the new background is drawn. In the example below the background is set to greyscale value 120, you could of course set it to colors. In the example below a white rectangle moves across the window from left to right. The image on the right is after it has moved 120 times:

```
int currentX ;

void setup()
{
  size(400,400) ;
  background(120) ;
  currentX = 0 ;
}

void draw()
{
  background(120) ;
  rect(currentX,100,50,50) ;
  currentX = currentX + 1 ;
}
```



Chapter 4: Mouse Interactivity

Processing makes it very easy to interact with the mouse. Processing keeps track of a number of **System Variables**. System variables are variables for which Processing maintains the contents. Two examples are: **mouseX** and **mouseY.** Type in the following code and hit run. While running move you mouse around inside the output window, fast and slow:

```
void setup()
{
  size(400,400) ;
  frameRate(12) ;
}

void draw()
{
  rect(mouseX,mouseY,50,50) ;
}
```

When you move the mouse around Processing draws a 50x50 rectangle at the current mouse location. Because the frameRate is set to 12, the draw( ) function is called every 1/12 th of second, and hence rectangles are drawn at 1/12 th of a second. If you leave the mouse in the window but don't move it, it is still drawing rectangles.

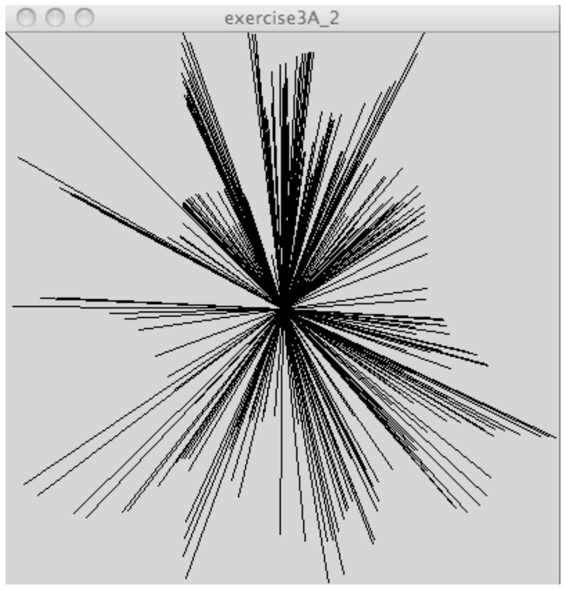
Now, replace the rect(mouseX,mouseY,50,50) above with:

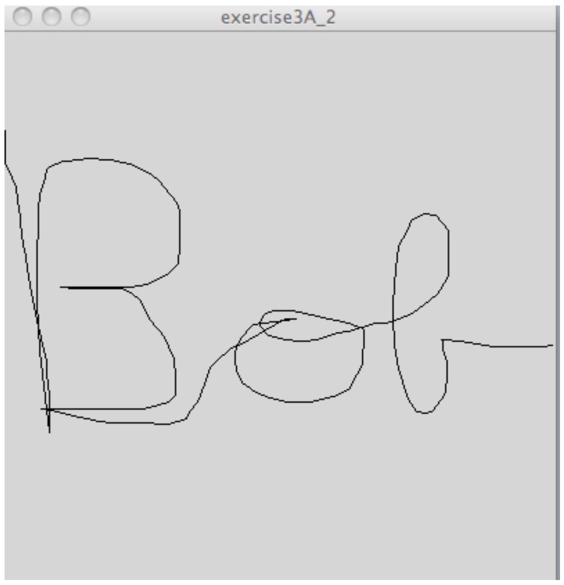
```
line(200,200,mouseX,mouseY) ;
```

After moving the mouse around you should get something that looks like the image below on the left. Replace the line with:

```
line(pmouseX,pmouseY,mouseX,mouseY) ;
```

and you will get something like the image on the right (if you try to move the mouse around to spell "bob":

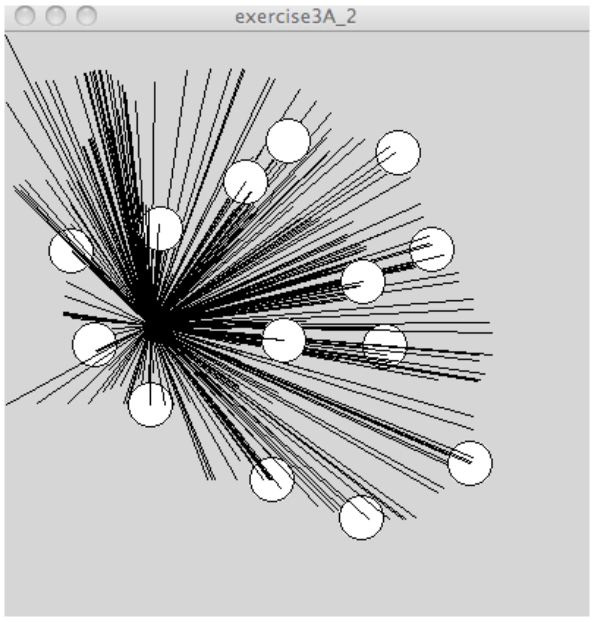




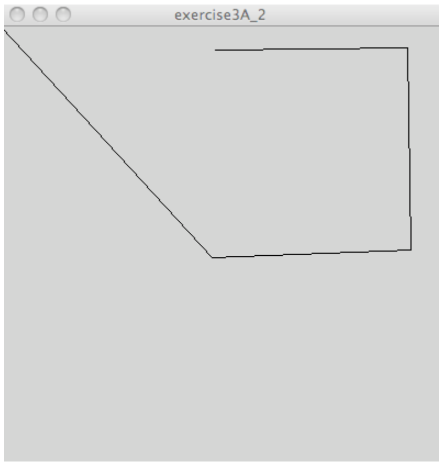
That last one was tricky. We used two more system variables: **pmouseX** and **pmouseY.** These two variables hold the previous value of mouseX and the previous value of mouseY respectively. Precisely: the previous value is the value the last time the draw function was called.

In addition to accessing the system variables mouseX, mouseY, pmouseX, and pmouseY, we can do more with the mouse. If you check out the reference from the drop down menu under "Help", and go about half way down you will see a bunch of mouse stuff. One of the functions available is **mousePressed( ).** You add this fundtion to the code as below on the left. Now, when you run the code, whenever the mouse is pressed that function is called. On the right below is the output from me running it and clicking 14 times:

```
void setup()
{
  size(400,400) ;
  frameRate(24) ;
}

void draw()
{
  line( 100,200, mouseX, mouseY) ;
}

void mousePressed()
{
  ellipse(mouseX,mouseY,30,30) ;
}
```

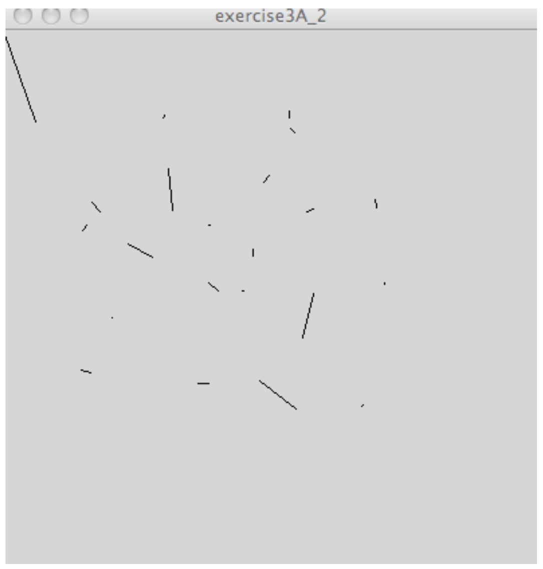


Now lets create a line drawing tool. Our tool should draw a line from the location of the last mouse click to the location of the current mouse click. For example, if I click 4 times in a row at locations: middle of screen, middle of the right side, top right corner, and then middle of the top, it should look like:



 Lets try first with the following code as shown below on the left. Unfortunately when we run that we get something that looks like the output on the right below. This output was created by many clicks and moves.

```
void setup()
{
  size(400,400) ;
  frameRate(12) ;
}

void draw()
{
  // do nothing
}

void mousePressed()
{
   line(pmouseX,pmouseY,mouseX,mouseY) ;
}
```



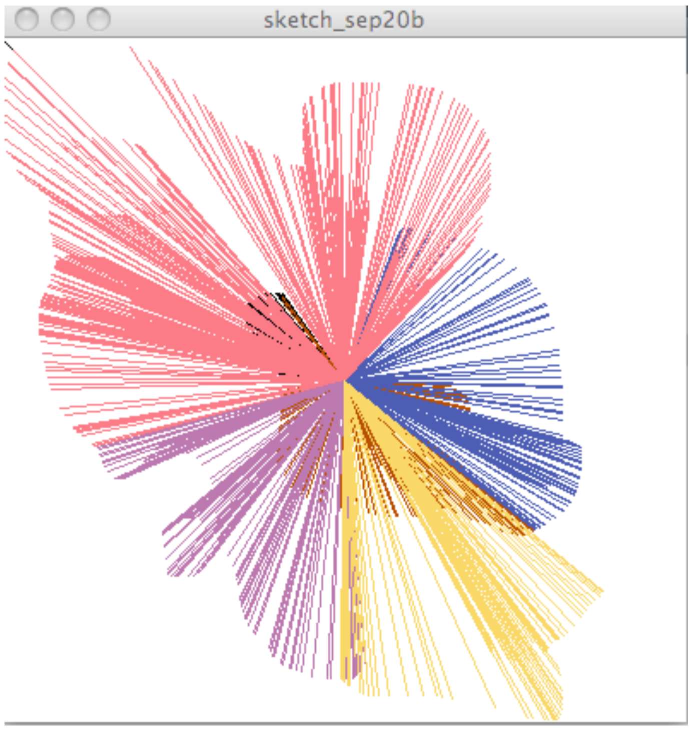
The problem here is that pmouseX and pmouseY are changed to the mouse location every time the draw function is called. And since it is being called 12 times per second, when you move your mouse to the next location for clickly pmouseX and pmouseY are being updated when you don't want them to. Instead, we need to keep track of the location of the mouse for the last time there was a mouse click. We can do this ourselves with variables as follows:

```
float lastX ;
float lastY ;

void setup()
{
  size(400,400) ;
  frameRate(12) ;
  lastX = 0 ;
  lastY = 0 ;
}

void draw()
{
  // do nothing
}

void mousePressed()
{
  line(lastX,lastY,mouseX,mouseY) ;
  lastX = mouseX ;
  lastY = mouseY ;
}
```

Here we create two variables that we name lastX and lastY. The need to be declared outside any of the functions so that all the functions can access them. This is called a **global variable** because you can access the variable from any of the functions, i.e. globally. In the setup( ) functin we intialize the variables to hold 0,0. Thus, the first line is going to start at location 0,0. Inside the mousePressed( ) function with draw a line from the last mouse location (which we know is stored in the variables lastX and lastY) to the current one. Then, we update the values in lastX and lastY to be the current mouseX and mouseY so that the next time mousePressed( ) is called the correct variables are in there!

EXERCISE 4A

Write code to use mouse movement and mouse clicks to create something pretty. Make it so each time you click the mouse the color of changes. Experiment. Example:


sketch_sep20b