Chapter 1: Getting Started

We will start our journey into Processing with creating static images using the following Processing commands:

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

To find out more about these commands, 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 that it will draw a rectangle with one corner at location x,y and then be of size width x height. As you go through this text, if you want more information, start first with Help -> References.

Now, 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:
Note, that each line must be terminated with a semicolon. The size command specifies the size of the window to be created, in this case 300 pixels wide by 300 pixels high. The command: `rect(50,10,100,50)` creates a rectangle at location (50,10) with a width 100 and height 50. The location of the rectangle, (50,10), is the upper left corner of the rectangle. You may be wondering why the rectangle is in the upper left corner if it is at location (50,10) instead of near the bottom of the window. The answer is the coordinate system is 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 about coordinates in Processing see:

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

As another example, consider the following code and the output it produces:

```plaintext
size(300,300) ;
rect(10,10,25,50) ;
rect(10,100,100,10) ;
rect(10,150,100,50) ;
rect(10,240,200,10) ;
```
The code shows that 4 rectangles are being created at locations (10,10), (10,100), (10,150), and (10,240). Notice also the size of the rectangles differ, the first is 25x50, the second is 100x10, the third is 100x50, and the last is also 200x10.

In addition to rectangles, Processing makes it easy to draw triangles, ellipses, lines, and quadrilaterals. Look under Help->Reference->2D Primitives for more details. Below is an example of each:

```java
size(300,300) ;
rect(10,10,200,10) ;
line(20,30,200,50) ;
triangle(10,60,200,60,150,90) ;
quad(10,100, 200,120, 250,150, 50,150) ;
ellipse(50,200, 25, 25) ;
ellipse(200,200,50,25) ;
```

Below is the first exercise. Learning to program requires you actually write code. Just reading a book will not do it for you. If you want to really learn, YOU MUST DO THE EXERCISES!!!! Programming is not a spectator sport.
**EXERCISE 1A**

Write code to create each of the following images.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image 1" /></td>
<td><img src="image2.png" alt="Image 2" /></td>
</tr>
<tr>
<td><img src="image3.png" alt="Image 3" /></td>
<td><img src="image4.png" alt="Image 4" /></td>
</tr>
</tbody>
</table>
**Color and line types:**

Processing allows you to use color for fills, background, and lines. It also allows you to create different line types, i.e. smooth versus dashed. 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 screens create color by mixing colored lights whereas in painting one mixes physical pigments. In Processing the default background color is grey. To change the background color use the background command:

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

By specifying 255, 255, and 0 as the three inputs to the command, the 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, background(100,0,0) results in a darker shade of red, and background (50,0,0) results in an even darker color. Note, the value (255,255,255) is the maximum of all colors, which will give you white. Smaller but equal numbers such as (200,200,200) will give you different shades of grey. The background command can be used with three input values as above, or, just one such as background(255). If you use just one value you get greyscale values between background(255), which will give you white, and background(0), which will give you black, and the numbers in between resulting in different values of grey.
You can also change the color of 2D Primitives (rect, triangle, ellipse, etc) by using the fill() command, for example: fill(255,0,0) will set the fill to red. When you execute the fill command it holds for all subsequent 2d primitive calls until the fill is changed. You can also modify the thinkness of the border (or of lines) by using the strokeWeight() command. The larger the number in the call 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. The first six lines create the red, green, and blue filled rectangles vertically descending along the left side. The next line sets the fill to white so that subsequent calls to rect will have a white interior. The stroke() command specifies the color of lines that are drawn. Specifically, the command stroke(255,0,0) sets the lines to be red. Thus, the next rectangle drawn has a red border, as determined by the previous stroke command, and a white fill. The strokeWeight() command specifies how think lines should be. Thus the next two lines change the border of subsequent rectangles to have a thickness of 5 and a color of green. Finally, the last stroke() and strokeWeight() commands cause the last rectangle to have a wide blue border.
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 Processing by Daniel Shiffman at:

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

**Comments and Errors**

In the last example above you will notice a line that start with two slashes “//”. The double slash signifies the start of a comment. A comment is something a programmer adds to the code to help explain the code to
others or to themselves when they look at the code later. In computer programming commenting code is very important. In fact, in large software development projects there are often more lines of comments than code! You will make your life easier if you comment your code to explain tricky parts so that when you come back to look at it a few days or months later you remember what you did.

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 or possibly run with unexpected results. 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 gobbledy-gob in the bottom window, and a more helpful message just above that says “unexpected token: null”. 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 values: x, y,
width, height, and I have only supplied 3.

In general, if you have an error message, look at the message, it might be
helpful, sometimes not. Regardless, Processing will always highlight the
line near or 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 is 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

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

- `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” to be a variable that can hold integers. The following code produces the following output:
In the above code the first four lines declare four integer variables: x, y, rectWidth, and rectHeight. In the next four lines of code we put initial values of 10, 20, 40, and 140 respectively into each variable. Putting initial values into variables is called **initialization**. The statement:

\[ x = 10; \]

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

We can now use the variable names in places that we have used numbers before. The current value in the variable is used for the value in the statement. Thus, the statement:

\[ \text{rect}(x, y, \text{rectWidth}, \text{rectHeight}); \]
creates a rectangle as normal, using the values inside variables x, y, rectWidth, and rectHeight. Hence, the first rectangle is drawn at location (10,20) with a width of 40 and a height of 140. One can change the content of a variable by assigning a new value to it. In the next line of code, the statement:

\[ x = 70; \]

replaces the current value of 10 in x with 70. Now the statement rect (x,y,rectWidth,rectHeight) creates a rectangle at location (70,20) still with width and height of 40 and 140.

Before the third call to rect(), we change the values of both variables x and rectHeight. In the statement:

\[ x = x + 60; \]

We assign to variable x the current value of x plus 60. Thus, after this statement x now contains the value 130.

In the statement:

\[ \text{rectHeight} = \text{rectHeight}/2; \]

we assign to variable rectHeight the current value of rectHeight divided by 2. Thus, after this statement variable rectHeight now contains the value 70.

After these statements, x, y, rectWidth, and rectHeight contain the values 130, 20, 40, and 70 respectively, hence the statement rect(x, y, rectWidth, rectHeight) draws a rectangle at location (130,20) with a width of 40 and a height of 70.
Note, variable creation and assignment can be done in one step. For example, the statements:

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

can be done in one statement:

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

Thus, the same effect of the 8 variable declaration and assignment statements above can be achieved with the following four statements:

```java
int x = 10 ;
int y = 20 ;
rectWidth = 40 ;
rectHeight = 60 ;
```

If you have programmed in Scratch, as suggested in the prefaces and explained in Appendix 1, then it helps to compare Scratch variables with Processing. In the table below we show how to write code using variables in both Processing and Scratch. If you did not take the advice and have not learned Scratch, ignore this table.
In Processing if you want to see the current value of a variable you can print it out using the `println` command:

    println( 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 Processing. For example:
int numLeft = 22;
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;
println(fy);
ix = 8;
iy = ix / 3;
println(iy);

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 a non-integer value and then assigns 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, even if the value seems like a whole number such as 2.0. 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. A float variable will allow you to assign an integer to it, but it automatically converts it into a float, i.e. 2 becomes 2.0. 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 software 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 type casting. An example is:

ix2 = (int) fx ;

or:
ix2 = int(fx) ;

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

**Naming Variables**

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 a reserved word in Processing the word will be highlighted in orange so you can see it is a reserved word, hence, you can not use it as a variable name. Variables cannot contain a space, and they can not start with a number. So, the variables “num1” and “num_1” are legal, but “num 1” and “1num” are not.

In general, use variable names that are descriptive of the contents. For example, if using variables to keep track of the location of a ball then it might be wise to use variable names: ballX and ballY, or ball_X and ball_Y.
Chapter 3: Simple Animations Using 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 code in Example 3.1. 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. Because this is dynamic make sure you actually type the code into Processing and run it yourself. The code for all numbered examples is available at the book website as clear text files to save you typing, but, for now the examples are short and getting in the habit of typing your own code is better than just cut-n-pasting other people’s code.

After running the code, let’s break down what part of the code does what, step by step:
EXAMPLE 3.1

```java
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 ;
}
```

<table>
<thead>
<tr>
<th>code</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int rx ;</td>
<td>Declare two integer variables. We will use this to keep track of the x,y coordinate of the current rectangle to be drawn</td>
</tr>
<tr>
<td>int ry ;</td>
<td></td>
</tr>
</tbody>
</table>

Processing: Pixels To Games, A Gentle Introduction
<table>
<thead>
<tr>
<th>code</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setup()</td>
<td>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 automatically called exactly once at the beginning of a code run.</td>
</tr>
<tr>
<td>size(400,400) ; rx = 1 ; ry = 1 ;</td>
<td>Initialize the screen size to be 400x400. Initialize the values of variables rx and ry to be 1.</td>
</tr>
<tr>
<td>frameRate(2) ;</td>
<td>This is a built in Processing function that specifies how frequently the draw() method is called. If set to 2, as in this example, then the draw() method is called twice per second.</td>
</tr>
<tr>
<td>void draw()</td>
<td>The draw function. This function is called by Processing repeatedly while the code runs. In here we put the code that we want to execute 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.</td>
</tr>
<tr>
<td>rect(rx,ry,50,50) ;</td>
<td>Draw a 50x50 rectangle at the location rx, ry, where rx and ry is the current value of those variables.</td>
</tr>
<tr>
<td>rx = rx + 5 ; ry = ry + 5 ;</td>
<td>Change the value of rx to be its current value plus 5. Same with ry.</td>
</tr>
</tbody>
</table>
The overall effect of this code is to draw a new rectangle 5 pixels to the right and 5 pixels down from the previously drawn rectangle twice per second.

You will notice the word `void` before the setup and draw functions. This is important and cannot be omitted. The word before a function is the return type. Functions can return values and hence we need to specify the type of value the function will return. For example, I can create a function that returns the average age of students in the classroom. Such a function would return a float value that contains the average value, hence, instead of the word `void` you would say `float`. `Void` is used to signify that you do not care about the return type, it will not be used. 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.

Let’s look at a new example:
In this example we keep drawing rectangles as before, but this time the rectangle locations and grey-scale color are random. The image on the right show the output after 46 rectangles have been drawn. Note, there is no reason for choosing 46 other than it looked cool to the author. Something new here is the use of the Processing function random() to generate random numbers. The command:

```plaintext
random(1, 350)
```

generates a random number between 1 and 350 inclusive.
You will notice that in all the examples so far the next rectangle is drawn and the previous one(s) remain. If you want to make a single object appear to move across the screen you need to wipe out its previous drawing and replace it with the new drawing shifted over slightly. Wiping out the previous drawing 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 a white rectangle moves across the window from left to right. The image on the right is after it has moved 120 times:

```
EXAMPLE 3.3

int currentX ;

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

void draw()
{
  background(120) ;
  rect(currentX,100,50,50) ;
  currentX = currentX + 1 ;
}
```
Exercise 3A

Write your own code to create something that looks like each of the following images. 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 the R, G, and B color values for each rectangle or circle.

Exercise 3B

Write code to make a blue circle start in the top right corner of the screen and move to the bottom left while changing color from blue to red. The website video titled “Exercise 3B” shows an example of what is desired.
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:

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

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

While running the code move your mouse around inside the window. 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:

```java
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”:

![Image of mouse movement and resulting text]

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 many functions related to the mouse. One of the functions available is mousePressed(). Go ahead and add this function 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:

```cpp
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.

```java
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 clicking 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 by keeping track of the last locations in variables as follows:

```c
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 variables 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. For now think of it this way: functions are greedy! If one declares a variable inside of a function it can not be shared by other functions. If on the other hand one declares a variable outside of any function, then all of the functions can access that variable. In computer programming this concept is called **scope** and will be explained in more detail later.

In the setup() function we intialize variables lastX and lastY to hold 0 and 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 because we have stored in variables lastX and lastY, to the current one. After drawing the line with the line() command 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.

<table>
<thead>
<tr>
<th>EXERCISE 4A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>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. Below is an example:</strong></td>
</tr>
</tbody>
</table>

[Image of a sketch with colorful lines]