

Introductory Programming With Processing

Scott Leutenegger
University of Denver

Table of Contents

Preface	3
Chapter 1: Getting Started	5
Color and line types:	9
Comments and Errors.....	11
Chapter 2: Variables	14
Naming Variables.....	18
Chapter 3: Simple Animations	19
Chapter 4: Mouse Interactivity	24
Chapter 5: Conditionals and Booleans	29
Booleans:	35
Chapter 6: For-Loops	38
Chapter 7: Doubly Nested For-Loops	45
Chapter 8: PImage: Image Manipulation	52
Chapter 9: Simple Functions	64
Chapter 10: Functions Returning a Value	74
Chapter 11: Using Functions With Colors	77

Preface

This book is intended for the beginning programmer. This particular version is intended as a text for the University of Denver course COMP 1100, a general education class that satisfies the foundational educational requirement in Math and Computing. Perhaps you have written a little bit of code in some language but do not yet fully understand concepts such as arrays, looping, lists, and list iteration. This book is to serve as a companion on your way to learning programming via Processing. After more than 25 years of teaching computer science it is the view of the author that most students benefit from a guide, something to help the learner to follow a path that allows rapid understanding of concepts yet travels through truly beautiful and engaging ideas. A book that is short, concise, and not expensive. Computer science is filled with beauty and the first steps of learning to program merely enable the learner to be ready to engage in the beauty, but why not see some beauty while learning the basics?

The **Processing** language provides an opportunity to explore beauty via visual image creation and manipulation while at the same time learning how to program. There are many excellent choices for learning to program each with various advantageous and disadvantages. We chose to focus on Processing and Java since Java is one of the most common “industrial grade” programming languages in use today yet at the same time provides a relatively easy learning curve. Processing is simply Java with a library of commands that make media programming easier and arguably more fun. Processing also uses a simple and elegant Integrated Development Environment (IDE). Many CS educators will scoff at the meager choice of tools available within the basic Processing IDE. We agree it is limited, but for a first programming environment Processing provides an excellent trade off between simplicity and power and we are taken with its elegant design. Its predecessor, design by numbers, is an even more beautiful example of simple design, but somehow most students don’t become excited by the idea of 100x100 grayscale pixel images. We encourage the learner to later move into more industrial strength IDEs such as Eclipse, Xcode, or Visual Studio.

If you already know how to program and just want to learn processing this

book is arguably not for you. For a non-nonsense logical introduction I would recommend “Learning Processing” by Daniel Shiffman. Shiffman’s book is also excellent for beginners although I have personally found good success with the slightly different order and examples found in this book. Two absolutely beautiful and inspiring books are “Processing: Creative Coding and Computation Art” by Ira Greenberg and “Processing: A Programming Handbook for Visual Designers and Artists” by Casey Reas and Ben Fry. These last two do not provide as logical of a path for the novice to follow, but the breadth of beauty found in them is nothing short of mind blowing. I strongly urge the reader to move on to these books either in tandem with this book or as a follow up.

To get the most out of this book, I recommend:

- 1) Read this book and **DO ALL THE EXERCISES!** One does not learn programming by reading alone. Reading a book, and reading code examples are excellent, but you must practice **WRITING** your own code to learn.
- 2) When you get curious and ask, “How could I change this so that instead of doing the example, it does something different”, **TRY IT.** Explore. Invent. When you get stuck ask a fellow student, teaching assistant, or your instructor if you are lucky enough to have on.

Don’t get hung up on getting everything 100% “correct” unless that is your personality, have fun. Learning is so much easier when you are having fun. Enjoy, and good luck on your journey.

Chapter 1: Getting Started

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

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

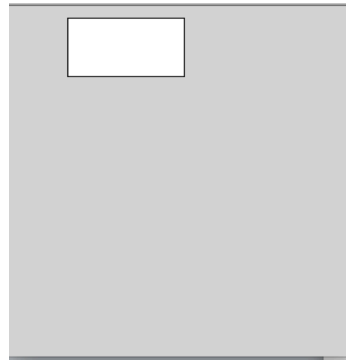
To find out more about these commands, and most things in Processing, select Help -> References from the menu at the top of the Processing Window. Then click on the command you want to look up, for example click on “rect()” to see how the rect() command is used. When you click on it you will see that rect() takes 4 arguments:

```
rect(x,y,width,height),
```

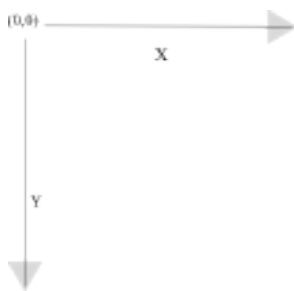
and that it will draw a rectangle with one corner at location x,y and then be of size width x height. As you go through this text, if you want more information, start first with Help -> References.

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:

```
File Edit Sketch Tools Help
sketch_dec03a §
size(300,300) ;
rect(50,10,100,50) ;
```



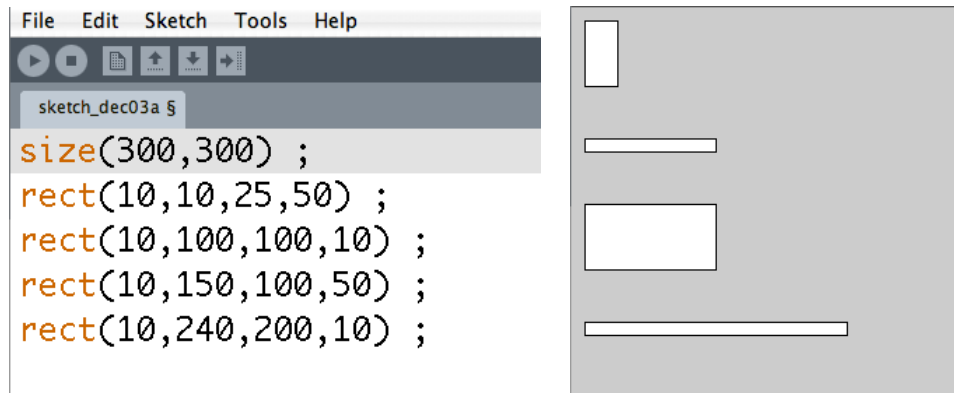
Note, that each line must be terminated with a semicolon. The size command specifies the size of the window to be created, in this case 300 pixels wide by 300 pixels high. The command: `rect(50,10,100,200)` creates a rectangle at location (50,10) with a width 200 and height 50. The location of the rectangle, (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 (10,10). This is because the coordinate system may be different than you are used to. Unlike the coordinate system you used in high school algebra, 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/>

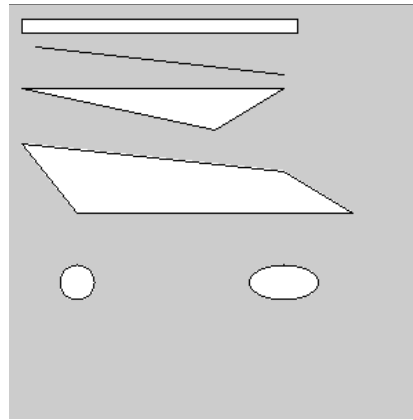
As another example, consider the following code and the output it produces:



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:

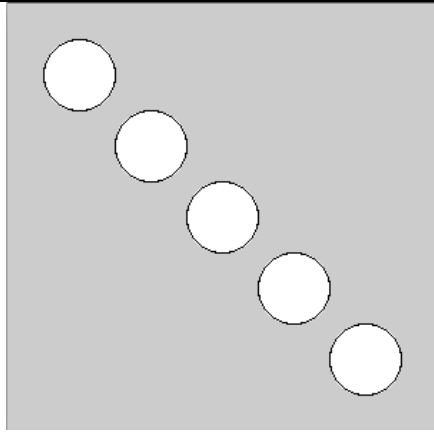
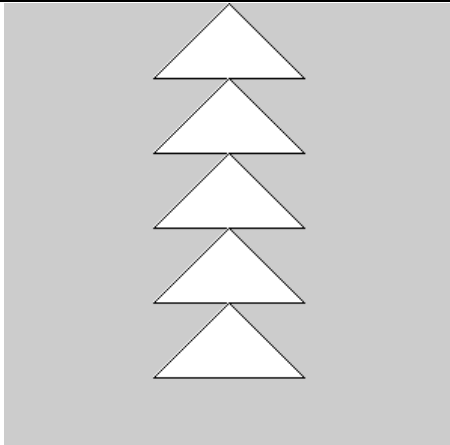
```
size(300,300) ;  
rect(10,10,200,10) ;  
line(20,30,200,50) ;  
triangle(10,60,200,60,150,90) ;  
quad(10,100, 200,120, 250,150,  
50,150) ;  
ellipse(50,200, 25, 25) ;  
ellipse(200,200,50,25) ;
```

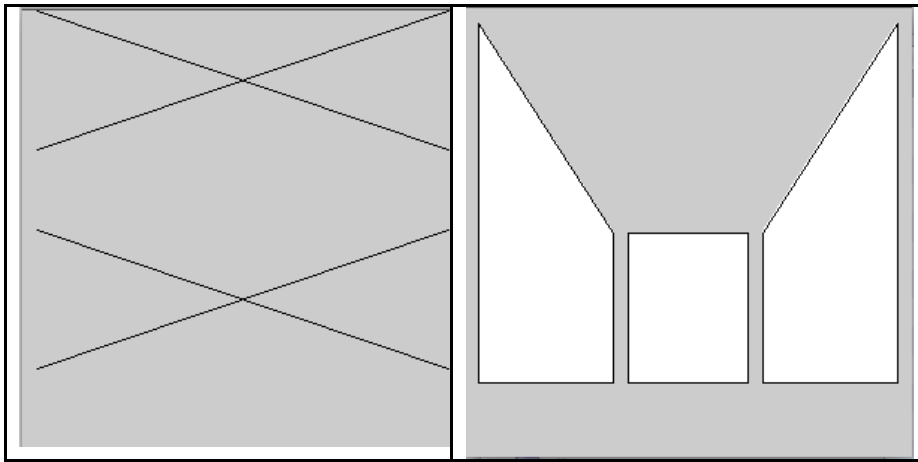


Below is the first exercise. Learning to program requires you actually write code. Just reading a book will not do it for you. If you want to really learn, **YOU MUST DO THE EXERCISES!!!!** Programming is not a spectator sport.

EXERCISE 1A

Write code to create each of the following images.

	
---	--



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 you are mixing physical pigments. In Processing the default background color is grey. To change the background color use the background command:

```
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: `fill(255,0,0)` will make subsequent 2d primitives filled in red. When you execute the `fill` command it holds for all subsequent 2d primitive calls. You can also modify the thickness 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 thick 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.

```

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) ;

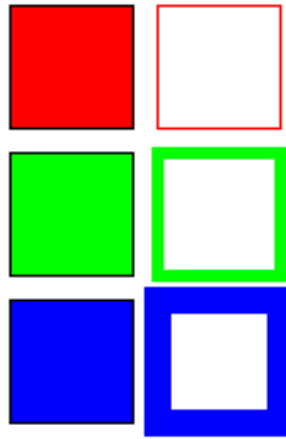
// set fill to white
fill(255) ;

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

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

stroke(0,0,255) ;
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 Processing by Daniel Shiffman at:

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

Comments and Errors

In the last example you will notice a line starting 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 programming commenting code is very important. In fact, in large software development projects there is 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.

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 gobbly-goob 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 it, 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:

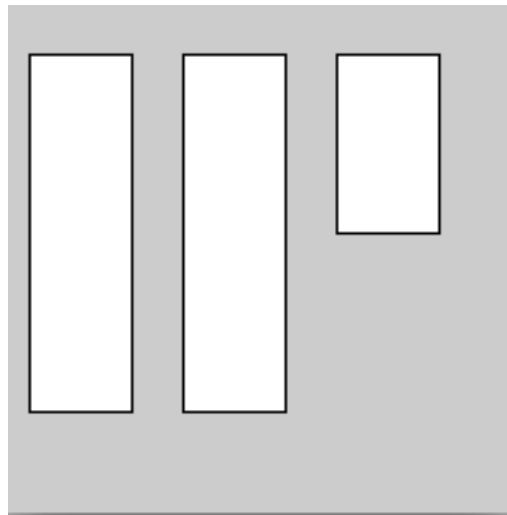
int holds integers, i.e. whole numbers
float holds decimal point numbers
char holds a character

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:

```
int x ;  
int y ;  
int rectWidth ;  
int rectHeight ;  
  
x = 10 ;  
y = 20 ;  
rectWidth = 40 ;  
rectHeight = 140 ;  
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 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:

```
rect(x, y, rectWidth, 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 recHeight. 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:

```
rectHeight = 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:

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

can be done in one statement:

```
int x = 10 ;
```

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. After all, 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 on 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.

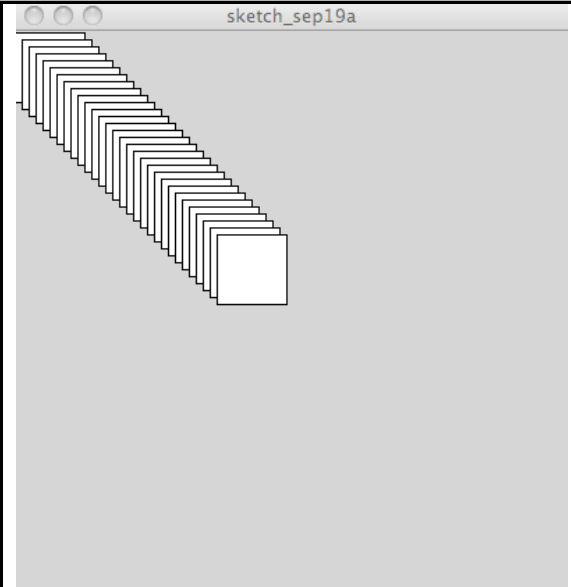
Naming Variables

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 it in Processing will 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 cannot 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 descriptive variable names. For example, if using variables to track that location of a ball logical names might be: **ball_X** and **ball_y**.

Chapter 3: Simple Animations

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 the 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.**

EXAMPLE 3.1	
<pre>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 ; }</pre>	

After running the code, lets break down what part of the code does what, step by step:

code	explanation
<pre>int rx ; int ry ;</pre>	Declare two integer variables. We will use this to keep track of the x,y coordinate of the current rectangle to be drawn
<pre>void setup() { } </pre>	<p>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, } .</p> <p>The setup function is a special Processing function that is called exactly once at the beginning of a code run.</p>
<pre>size(400,400) ; rx = 1 ; ry = 1 ;</pre>	Initialize the screen size to be 400x400. Initialize the values of variables rx and ry to be 1.
<pre>frameRate(2) ;</pre>	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.
<pre>void draw() { } </pre>	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.
<pre>rect(rx,ry,50,50) ;</pre>	Draw a 50x50 rectangle at the location rx, ry, where rx and ry is the current value of those variables.
<pre>rx = rx + 5 ; ry = ry + 5 ;</pre>	Change the value of rx to be its current value plus 5. Same with ry.

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

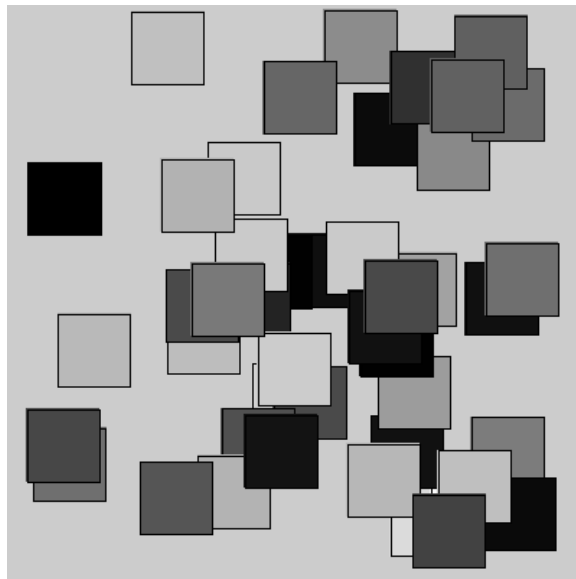
per second.

You will notice the word void before the setup and draw functions. This is important and can not 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, hence, instead of the word void you would say float. Void is use to signify that you do not care about the return type, i.e. 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.

Lets look at a new example:

EXAMPLE 3.2

```
float rx ; // rectangle's x-  
coord  
float ry ; // rectangle's y-  
coord  
float gscaleColor ; //random  
color  
  
void setup()  
{  
  size(400,400) ;  
  frameRate(2) ;  
}  
  
void draw()  
{  
  rx = random(1,350) ;  
  ry = random(1,350) ;  
  gscaleColor = random(255) ;  
  fill( gscaleColor ) ;  
  rect(rx, ry, 50, 50) ;  
}
```



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:

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

generates a random number between 1 and 350 inclusive.

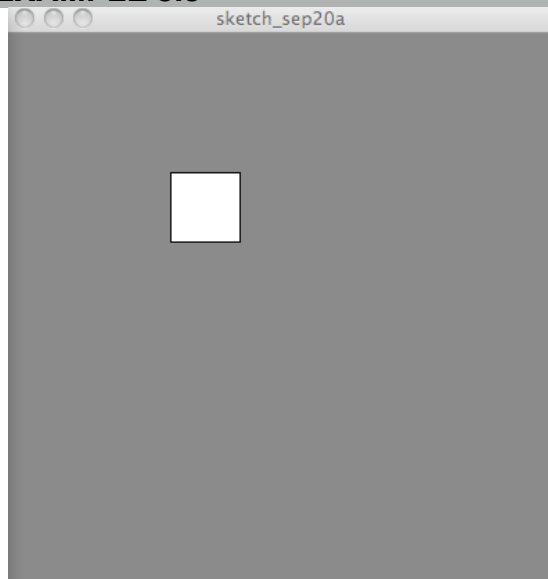
You will notice that in all the examples so far the next rectangle is drawn and the previous one(s) remain. If you want to make a single object appear to move across the screen you need to wipe out it's previous drawing and replace it with the new drawing shifted over slightly. Wiping out the previous 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 ;

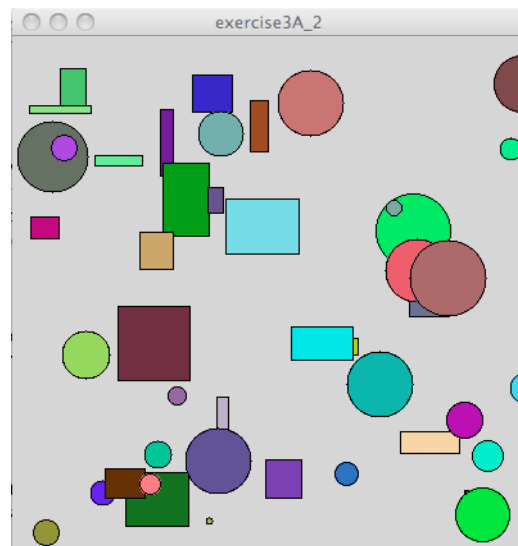
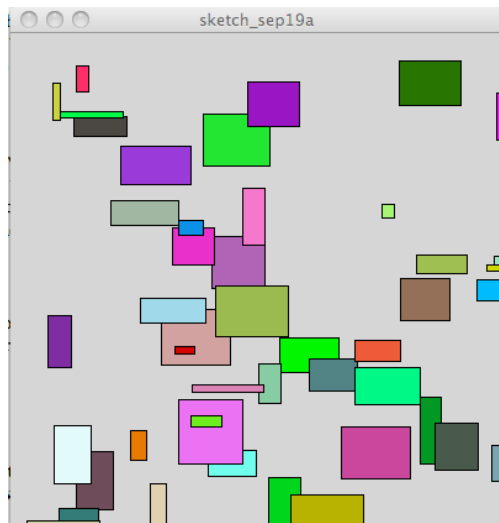
void setup()
{
  size(400,400) ;
  background(120) ;
  currentX = 0 ;
}

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 red circle start in the top right corner of the screen and move to the bottom left while changing color from red to blue. 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:

```
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.

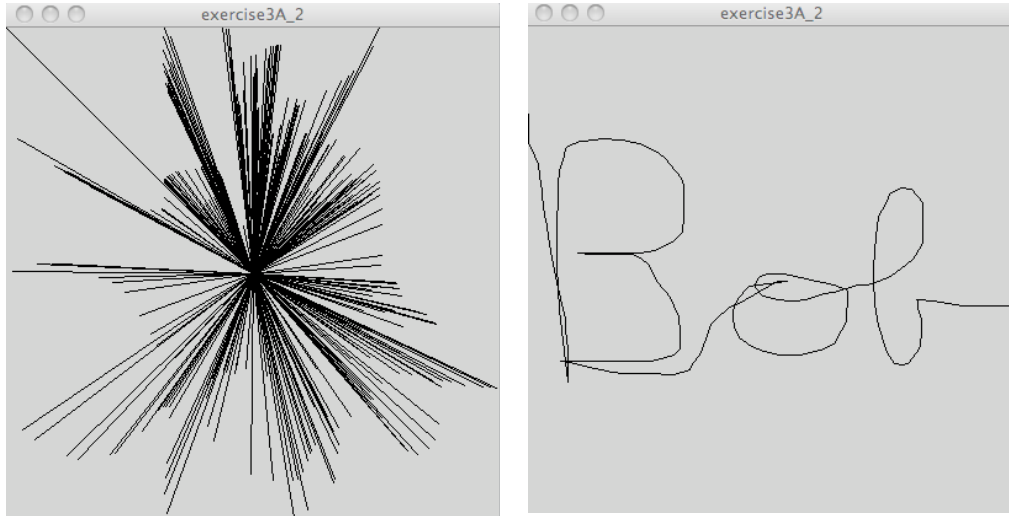
Now, replace the `rect(mouseX,mouseY,50,50)` above with:

```
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 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:

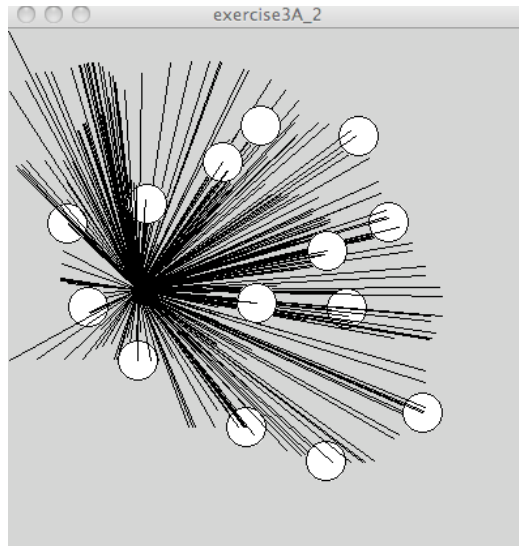
```

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

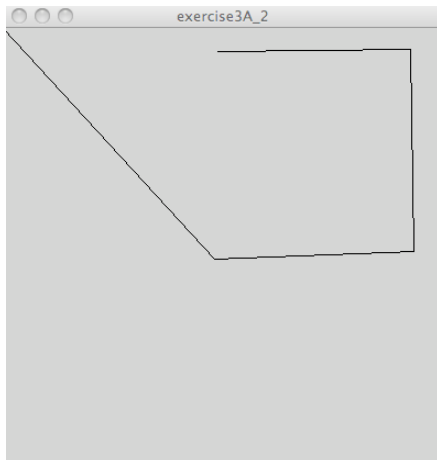
void draw()
{
  line( 100,200, mouseX,
mouseY) ;
}

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(mouseX, mouseY, mouseX,
        mouseY) ;
}
```



The problem here is that mouseX and mouseY 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 mouseX and mouseY 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 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 initialize 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.

Chapter 5: Conditionals and Booleans

We often use conditional statements such as

```
if a certain condition is true
    do one set of commands
else
    do another set of commands.
```

The general format in Processing is as follows:

```
if ( )
{
}
else
{
}

if ( )
{
}
```

The format on the left is used when you want something to happen if the condition in parenthesis is true, and you want something else to happen if the condition is false. The format on the right is used when you want something to happen if the condition inside the parenthesis is true but do not have any action in mind if the condition is false.

Consider the following if/else example. When the user clicks on the right side of the screen, i.e. ($\text{mouseX} > 199$), small red circles are drawn, otherwise larger green rectangles are drawn. The image on the right was captured after clicking a bunch of times:

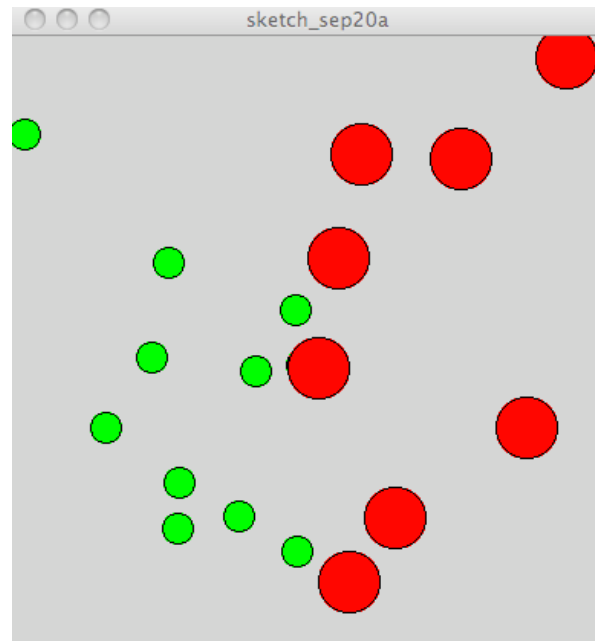
```

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

void draw()
{
  // do nothing
}

void mousePressed()
{
  if (mouseX > 199)
  {
    fill(255,0,0) ;
    ellipse(mouseX,mouseY,40,40) ;
  }
  else
  {
    fill(0,255,0) ;
    ellipse(mouseX,mouseY,20,20) ;
  }
}

```



It is important that every open brace, { , have a matching close brace, } . If you do not have a matching open/close brace, you will get an error. Further, your code will be much more readable, and you will be less prone to creating errors, if you vertically align the open/close braces. Processing will do this indentation for you automatically if you let it. You yourself have to force bad alignment.

Now lets make a ball move back and forth across the screen. The code is in the box below. We will do this by using three variables that we name: velocityX, currentX, and currentY. In the setup() function we initialize these three to 1, 0, and (height / 2) respectively. Note, **height** and **width** are system variables that hold the height and width of the window. Since we set size to 300x300 height will contain 300. We will use velocityX to keep track of in what direction and how fast the rectangle is move.

In the draw() function we first set the background to wipe out previous circles, then draw a circle at location (currentX,currentY). We then need to update the currentX location so that the next time draw() is called the ball moves over. Variable velocityX contains the current velocity. If it contains a positive number we want the ball to move to the right, if it contains a negative number we want to move to the left. We can do this simply by saying:

```
currentX = currentX + velocityX ;
```

Now, currentX will contain the next x-coordinate for the ball so it moves the next time draw() is called. We need to do one more thing, we need to check that the ball has not gone off the screen. If currentX is greater than the contents of system variable width, then it has gone off the screen. We can bring it back by negating, i.e. multiplying by -1, the contents of velocityX. Likewise, we need to make sure the ball has not gone off the screen by going too far to the left, i.e. currentX < 0. Again, if currentX < 0 we just negate our variable velocityX.

```
int velocityX ;
int currentX, currentY ;

void setup()
{
  size(300,300) ;
  background(150) ;
  velocityX = 1 ;
  currentX = 0 ;
  currentY = height / 2 ; // middle of screen
}

void draw()
{
  background(150) ; // wipe out previous drawings

  // draw the ball
```

```

ellipse(currentX, currentY, 20,20) ;

// update currentX for next call to draw( )
currentX = currentX + velocityX ;

// negate velocityX if ball going off screen
if (currentX > width)
{
    velocityX = velocityX * -1 ;
}
if (currentX < 0)
{
    velocityX = velocityX * -1 ;
}
}

```

The code above works fine but can be made more concise and neat. The logic for updating velocityX is currently:

```

if (currentX > width)
{
    velocityX = velocityX * -1 ;
}
if (currentX < 0)
{
    velocityX = velocityX * -1 ;
}

```

This can be rewritten as:

```

if ( (currentX > width) || (currentX < 0) )
{
    velocityX = velocityX * -1 ;
}

```


In this example you see a new funny expression: `||`

The `||` means “or”. This is saying if either of the expressions (`currentX > width`) OR (`currentX < 0`) is true, then the whole if condition evaluates to true. Notice you need a total of three open parenthesis and three close parenthesis. It is more common, and easier to read, to write your code the second more concise way.

In addition to OR there is the AND operator. In Processing/Java the AND operator is two ampersands: `&&`. Hence, I could write:

```
if ( (currentX > width) && (currentX < 0) )
```

but this would always evaluate as false unless width was a negative number, which, in Processing it must always be a non negative number.

Exercise 5A

Make a ball bounce around the screen in both dimensions. Hint: you will need to update `currentY` also, and we suggest you create and use a `velocityY` variable.

Modify your ball so that when it is on the left half of the screen the ball is red, and when it is on the right half of the screen the ball is green.

So far we have said if/else statements have to follow the form:

```
if ( )
{
  // if statements
}
else
{
  // else statements
}
```

```
}
```

Actually, this is not quite right, the general form is:

```
if ( )  
    // if statement block  
else  
    // else statement block
```

A statement block is one or more statements. If it is more than one statement the block needs to be enclosed by curly braces. If it is only one statement there is no need for the curly braces. Consider the table of expression of equivalent code below. In the first case both the if and else statement blocks are single lines of code, hence we can omit the curly braces. In the second case both are two or more lines, hence the curly braces are mandatory. In the third case, the if statement block has only one line, hence we can remove the curly braces there.

Original code	More concise code
<pre> if (age < 75) { System.out.println("not old") ; } else { System.out.println("old") ; } </pre>	<pre> if (age < 75) System.out.println("not old") ; else System.out.println("old") ; </pre>
<pre> if (age < 75) { System.out.println("not old") ; age = age + 1 ; } else { System.out.println("old") ; age = age + 2 ; // rapid aging } </pre>	<p>no smaller expression, both statement blocks have two or more statements, hence, the braces are necessary.</p>
<pre> if (age < 75) { System.out.println("not old") ; } else { System.out.println("old") ; age = age + 2 ; // rapid aging } </pre>	<pre> if (age < 75) System.out.println("not old") ; else { System.out.println("old") ; age = age + 2 ; // rapid aging } </pre>

Booleans:

The expressions inside the parenthesis of an if statement are called **Booleans**. Boolean is just a fancy name for an expression that evaluates to true or false. Lets say Assume a few variables are created as follows:

```
int age ;
```

```
int weight ;  
int gpa ;
```

In the following table I show some some Boolean expressions. Lets assume I first run the code in Row 1 to initialize the variables, then we will look at the following rows to evaluate the as true or false.

Row	Processing/Java	
1	age = 19 ; weight = 115 ; gpa = 3.72 ;	
2	(age < 20)	TRUE
3	(weight == 115)	TRUE
4	(gpa > 3.0) && (age < 20)	TRUE
5	(gpa > 3.8) (weight < 100)	FALSE
6	(age >= 19)	TRUE
7	(age > 21) && (weight < 125)	FALSE
8	(age != 19)	FALSE

Row 1 contains the code for assigning values to the variables. Row 2 evaluates to true since 19 is < 20. Note in row 3 that in Processing/Java “==” is used for equals, see the important note below. This evaluates to true as weight contains the value 115. For row 4 to be true BOTH expressions need to be true, and, they are. In row 5 we have and OR operator. Or will evaluate to true if either or both of the expressions on each side of the OR are true. In this case, neither is true, hence the expression evaluates to false. Note, if the right hand expression were replaced with (weight > 100), then the entire expression would evaluate to true because one of the two sides would be true. In row 6 the expression evaluates to true. The operator “>=” means greater or equal. Since it is equal the expression evaluates to true. Scratch does not have a <= or >= operator. In row 7 we have an AND operator. For AND to be true both sides must be true. If either or both of the sides are false the entire expression evaluates as false. In this case, the left hand side is false,

hence, the entire expression is false. In row 9 the expression is (AGE NOT EQUAL TO 19). In Processing/Java we write this as (age != 19). Since age contains 19 this expression evaluates to false.

IMPORTANT: In row 3 note that we use “==”, i.e. two equal signs, to represent = in Processing/Java. A single = sign means assign the right hand expression to the left hand variable. Thus, we can not use a single = to mean “equals”. We could do it in Scratch because we had a different syntax for assignment, namely “set varName to value”. Here is a place that java differs from Processing. Lets say you mistakenly type:

```
if (weight = 115)
```

In Java the code would run, the value of 115 would be assigned to variable weight and the expression would evaluate as “TRUE”. You need to be careful in Java. Processing is more friendly as Processing will flag this as an error and not run the code.

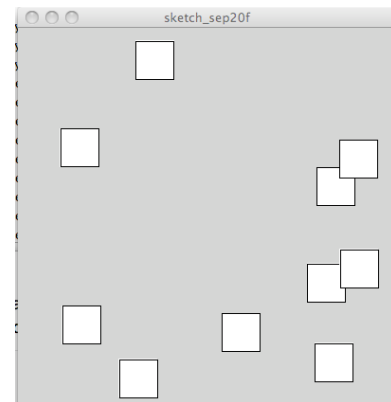
EXERCISE 5B	
int A = 10 ; int B = 20 ; int C = 30 ;	Answer True or False
(A < 15)	
(B == 20) && (C < 10)	
(B == 20) (C < 10)	
(A < 15) && (B <= 20) && (C > 0)	
(A < 15) ((B == 10) && (C > 10))	
(A != 10)	
((A < 0) (B > 0)) && ((A==10) (C == 10))	

Chapter 6: For-Loops

Often in programming you want to run a chunk of code multiple times. For example, let's say you have written a grading program that calculates the grade for each student in the course by adding up their exam and assignment grades, weighting them appropriately, and then printing out the final numeric grade. You would want this code to run multiple times, once for each student.

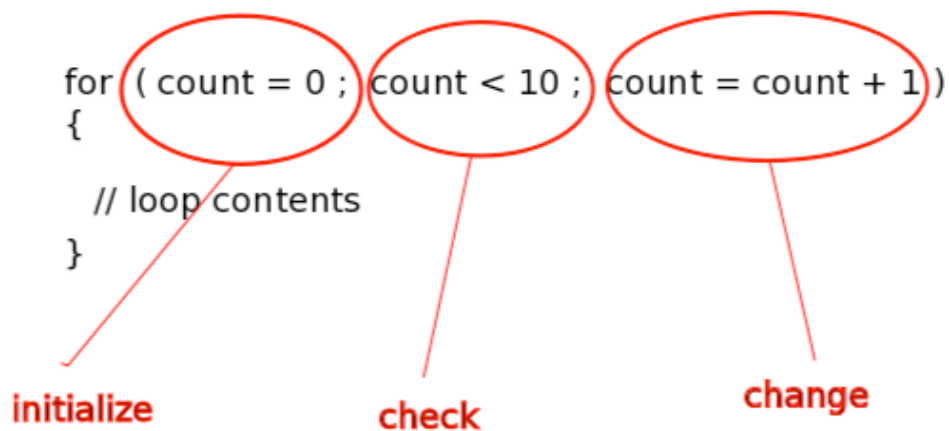
Let's start with creating multiple rectangles in a static Processing program. By static we mean there is not draw() function that is called multiple times, hence, the code is just run once and creates a single image. The following code creates the following image:

```
size(400,400) ;  
int count ;  
float rx ;  
float ry ;  
  
for (count = 0 ; count < 10 ; count = count + 1)  
{  
  rx = random(width) ;  
  ry = random(height) ;  
  rect(rx,ry,40,40) ;  
}
```



The loop, i.e. the code between the curly braces, gets executed ten times. Hence, 10 rectangles are drawn at 10 random locations. If you change the 10 to a 20 the loop is executed 20 times and 20 rectangles are drawn. It is important to note this is very different than in the previous chapters where we used the draw() method to keep drawing rectangles. In this case the code is run once and the image is created and displayed, whereas when using draw() the contents of draw() are run until the program is stopped.

A for-loop can be explained as having three parts as show here:



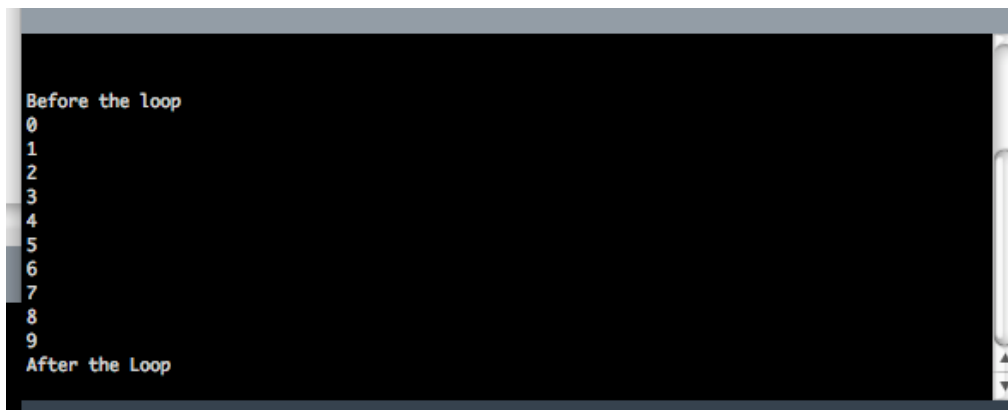
The initialize part gets executed exactly once before the loop starts. Then we check the condition in the middle. If it is true we run the contents of the loop. Then we execute the change part. Then we repeat: check, run, change. Check, run, change. Check, run, change. And so on, until the check fails. In this specific case:

- 1) initialize count to the value 0
- 2) check if count is < 10. YES. Run the loop. Change count to 1.
- 3) check if count is < 10. YES. Run the loop. Change count to 2.
- 4) check if count is < 10. YES. Run the loop. Change count to 3.
- 5) check if count is < 10. YES. Run the loop. Change count to 4.
- 6) check if count is < 10. YES. Run the loop. Change count to 5.
- 7) check if count is < 10. YES. Run the loop. Change count to 6.
- 8) check if count is < 10. YES. Run the loop. Change count to 7.
- 9) check if count is < 10. YES. Run the loop. Change count to 8.
- 10) check if count is < 10. YES. Run the loop. Change count to 9.
- 11) check if count is < 10. YES. Run the loop. Change count to 10.
- 12) check if count is < 10. NO => So execution of the loop stops and the program execution then moves on to the next command after the loop closing brace.

To see the values of the variable count change as the code runs we can print them out as in the following code:

```
int count ;
for (count = 0 ; count < 10 ; count = count + 1)
{
    System.out.println( count ) ;
}
```

You can see in the output window the following:



```
Before the loop
0
1
2
3
4
5
6
7
8
9
After the Loop
```

Thus, the variable count takes on the values 0 .. 9, but does not enter the loop once count is equal to 10.

The expression “count = count + 1 “ is so common in programming that most programming languages, including Processing/Java, give you a shorthand notation:

```
count ++ ;
```

Likewise, varName = varName - 1 is very common and hence is:

```
varName -- ;
```

Thus, the loop above can also be written:

```
for ( count = 0 ; count < 10 ; count++)
{
}
}
```


Another common convention is to use the variable name “i” for a loop counter as follows:

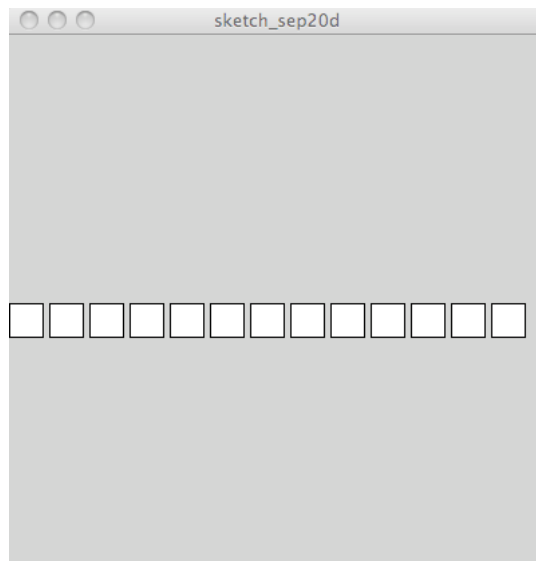
```
int i ;  
for (i = 0 ; i < 10 ; i++)  
{  
  // loop contents  
}
```

Finally, often the loop counting variable is declared inside the loop in the initialization clause as follows:

```
for ( int i = 0 ; i < 10 ; i++)  
{  
}
```

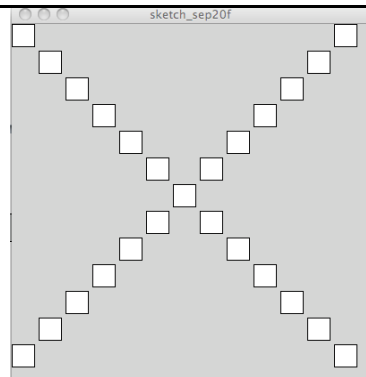
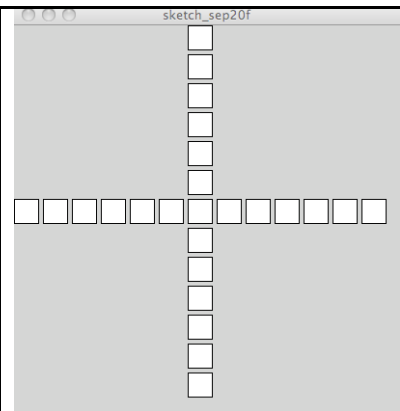
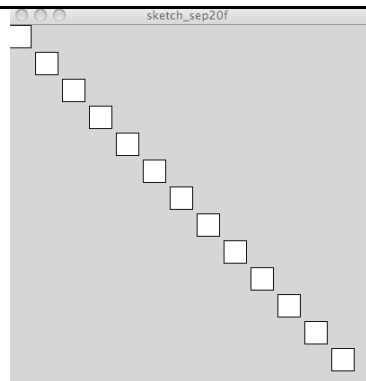
Sometimes the value of the loop counter is used inside the loop as well as being used to determine how many times the loop executes. An example is below. Notice how the x-coordinate of each rectangle is generated from the current value of the loop variable “i”. The rectangle’s x-coordinates are $i*30$, hence, they are located at {0,30,60,90,120,150,180,210,240, and 270}:

```
size(400,400) ;  
for (int i = 0 ; i < 10 ; i++)  
{  
  rect( i*30, 200, 25, 25) ;  
}
```



EXERCISE 6A

Use one or two for loops to create the following images (or something close to them). Do NOT write a separate `rect(x,y,w,h)` command for each rectangle, rather, put the command inside a for loop.



If you make small circles and create lots of of them you can get more interesting pictures such as in the following example. This example, and the exercise following, are motivated by some examples in chapter 6 of the book “Processing. Create Coding and Computational Art.” by Ira Greenberg. Notice that the first loop is executed 133 times (why?), the second loop 200 times (again, why?), and the last loop 400 times. Thus, the first line of dots contains 133 dots, the second contains 200, and the third contains 400.

```

size(400,400) ;
background(100) ;
fill(255) ;
stroke(100) ;

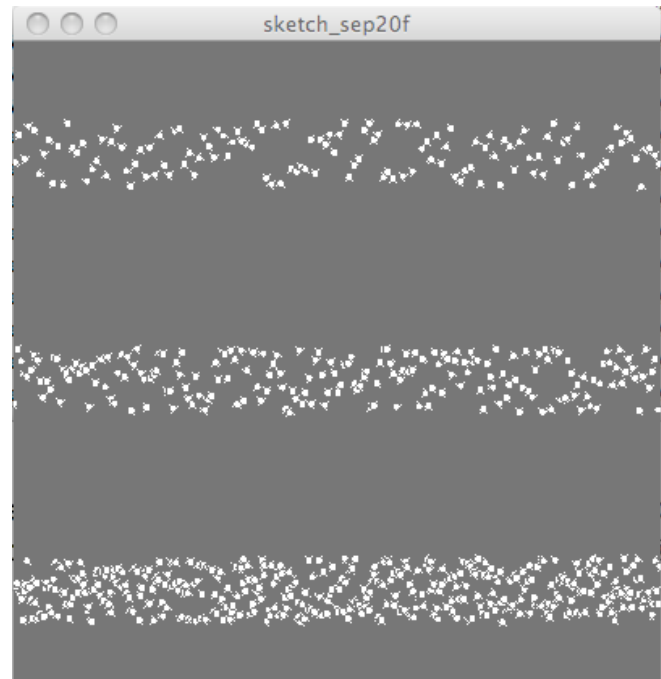
float rx ;
float ry ;

for (int i = 0 ; i < 400 ; i = i + 3)
{
  rx = i + 0.2 * random(-1,1) ;
  ry = random(50,90) ;
  ellipse(rx,ry,5,5) ;
}

for (int i = 0 ; i < 400 ; i = i + 2)
{
  rx = i + 0.2 * random(-1,1) ;
  ry = random(190,230) ;
  ellipse(rx,ry,5,5) ;
}

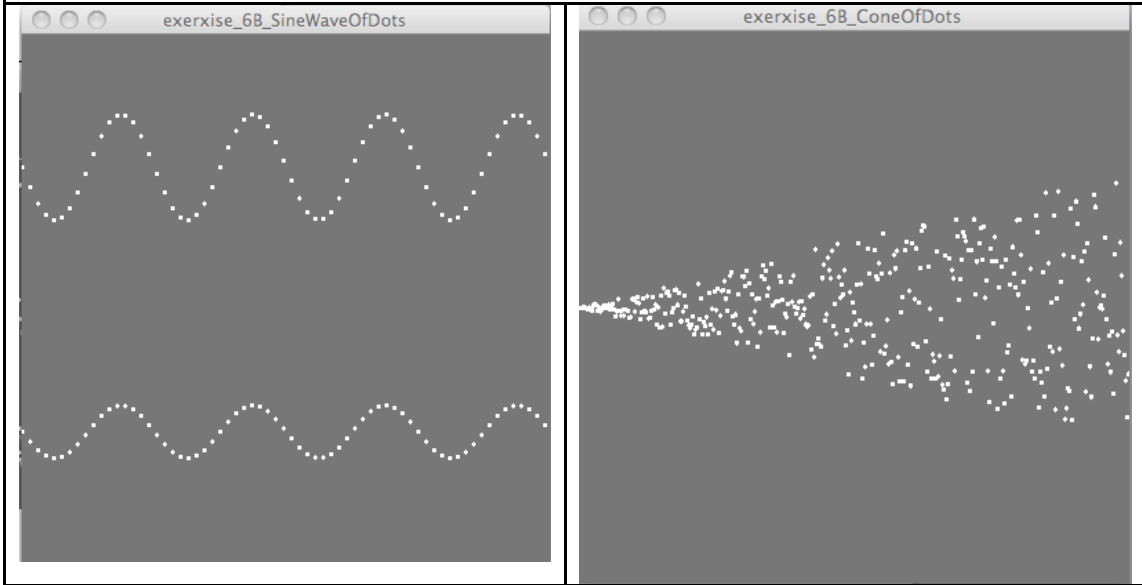
for (int i = 0 ; i < 400 ; i++)
{
  rx = i + 0.2 * random(-1,1) ;
  ry = random(320,360) ;
  ellipse(rx,ry,5,5) ;
}

```



EXERCISE 6b

Write code to create the following images. Hint: for the left hand one use the `sin()` function, i.e. $y = \sin(\text{float } i / 400 * 2 * \text{PI})$, where `PI` is a system variable that holds the mathematical value of `PI`, i.e. 3.14159....



Chapter 7: Doubly Nested For-Loops

A for loop runs the commands nested inside the loop braces for some number of iterations. One can put any valid command inside a for loop including another for loop. Consider the following code and output:

```
for (int i = 0 ; i < 4 ; i++)      0 0
{                                  0 1
  for (int j = 0 ; j < 4 ; j++)    0 2
  {                                  0 3
    System.out.println(i + " " + j); Hi
  }                                  1 0
  System.out.println("Hi");        1 1
}                                    1 2
System.out.println("Bye");         1 3
                                   Hi
                                   2 0
                                   2 1
                                   2 2
                                   2 3
                                   Hi
                                   3 0
                                   3 1
                                   3 2
                                   3 3
                                   Hi
                                   Bye
```

We call the two loops the outer loop (the one with “i” as the loop variable) and the inner loop (the one with “j” as the loop variable.) When the outer loop starts it initializes “i” to hold 0. It then checks and enters the loop. Now the inner loop command is started. The variable j is initialized to 0. Now:

The inner loop is run and prints out "0 0".

The inner loop finishes its first iteration.

j is changed to 1.

The inner loop is run again and prints out "0 1" because i still equals zero.

The inner loop finishes its second iteration.

j is changed to 2.

The inner loop is run again and prints out "0 2" because i still equals zero.

The inner loop finishes its third iteration.

j is changed to 3

The inner loop is run again and prints out "0 3" because i still equals zero

j is changed to 4

=> the inner loop check condition is not true now so the inner loop finishes.

The outer-loop command just after the inner loop is now run. Thus, "Hi" is printed out. That has completed one iteration of the outer loop. So, now variable i is changed to 1. The outer loop code is run with i now equal to 1. The code gets to the inner loop, initializes j to 0. Now:

The inner loop is run and prints out "1 0".

The inner loop finishes its first iteration.

j is changed to 1.

The inner loop is run again and prints out "1 1" because i still equals 1.

The inner loop finishes its second iteration.

j is changed to 2.

The inner loop is run again and prints out "1 2" because i still equals 1.

The inner loop finishes its third iteration.

j is changed to 3

The inner loop is run again and prints out "1 3" because i still equals 1

j is changed to 4

=> the inner loop check condition is not true now so the inner loop finishes.

The outer-loop command just after the inner loop is run, thus "Hi" is now printed. That has completed two iterations of the outer loop. So, now variable i is changed to 2. Two more iterations of the outer loop happen, each one causing four iterations of the inner loop, to print out:

```
2 0
2 1
2 2
2 3
Hi
3 0
3 1
3 2
3 3
Hi
```

At this point, variable i is changed to 4, the check condition of the outer loop is not satisfied, and the program goes on to the next command after the loop, thus printing out :

```
Bye
```

Now consider the following example:

```
for (int i = 0 ; i < 4 ; i++)      0 0
{                                  0 1
  for (int j = i ; j < 4 ; j++)    0 2
  {                                  0 3
    System.out.println(i + " " + j); Hi
  }                                  1 1
  System.out.println("Hi") ;      1 2
}                                  1 3
System.out.println("Bye") ;      Hi
                                  2 2
                                  2 3
                                  Hi
                                  3 3
```

Hi
Bye

Notice that in the inner loop the initialization condition is:

```
int j = i ;
```

Thus, when i equals 0, j starts at 0. When i equals 1, j starts at 1, when i equals 2, j starts at 2, and when i equals 3, j starts at 3.

EXERCISE 7A	
For each of these problem write out what gets printed. DO NOT just type in the code and run it, figure it out by hand. Hint: the code on the left prints out 10 lines of output, and the code on the right prints out 8 lines of output.	
<pre>for (int i = 0 ; i < 5 ; i++) { for (int j = 3 ; j < 5 ; j++) { System.out.println(i + " " + j) } }</pre>	<pre>for (int i = 3; i < 5 ; i++) { for (int j = 1 ; j < 5 ; j++) { System.out.println(i + " " + j) ; } }</pre>

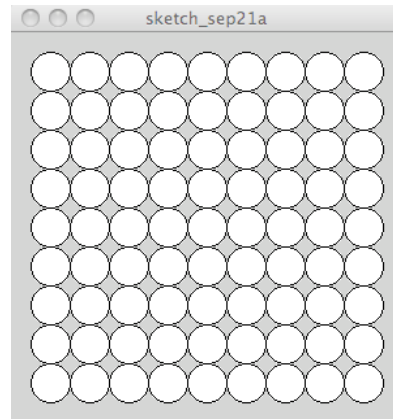
Granted, the above examples are cool to think about, but they are rather silly in that they don't actually "do" anything. Lets use doubly nested for loops to create some images in Processing. For example:


```

size(300,300) ;

for (int i = 1 ; i <= 9 ; i++)
{
  for (int j = 1 ; j <= 9 ; j++)
  {
    ellipse( i*30, j*30, 30, 30) ;
  }
}

```

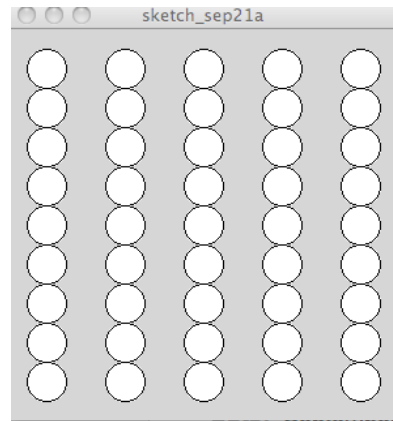


```

size(300,300) ;

for (int i = 1 ; i <= 9 ; i = i + 2)
{
  for (int j = 1 ; j <= 9 ; j++)
  {
    ellipse( i*30, j*30, 30, 30) ;
  }
}

```

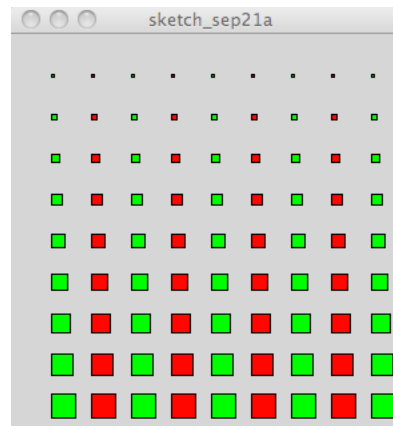


```

size(300,300) ;

int w ;
for (int i = 1 ; i <= 9 ; i = i + 1)
{
  for (int j = 1 ; j <= 9 ; j++)
  {
    if ((i%2) == 0)
      fill(255,0,0) ;
    else
      fill(0,255,0) ;
    w = j * 2 ;
    rect( i*30, j*30, w, w) ;
  }
}

```



Note in the example above the statement:

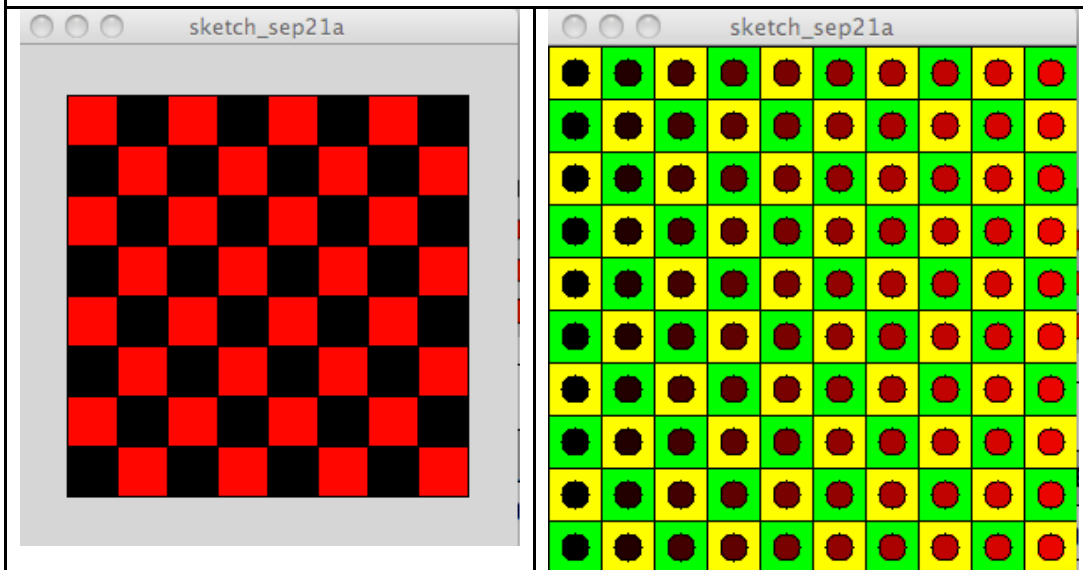
```
if ( (i%2) == 0)
```

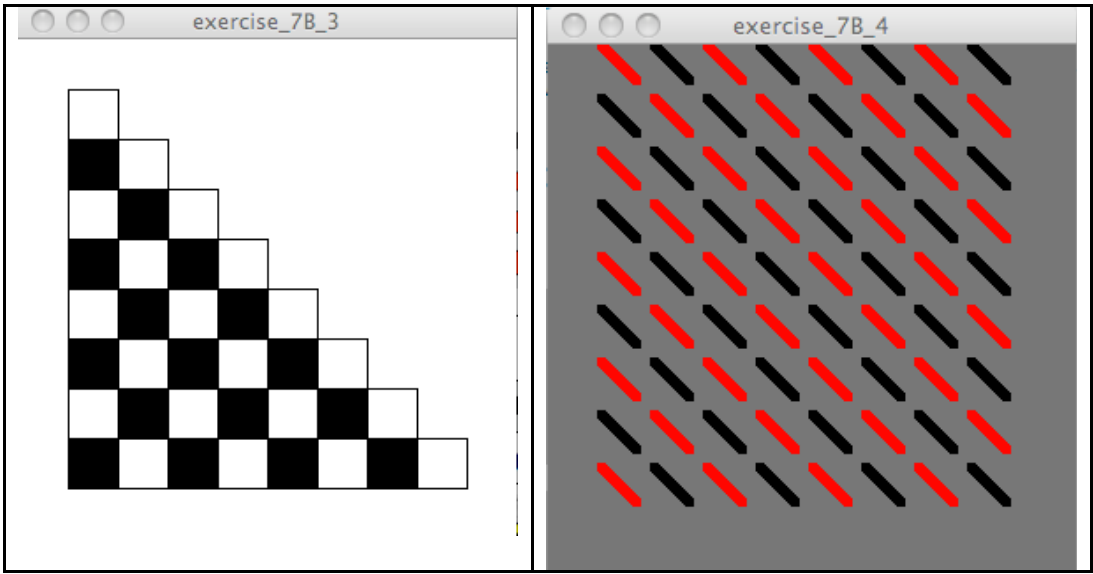
The percent sign is the “modulo” operator. It returns the remainder of dividing the left operand by the right operand. For example: $10 \% 3$ would equal 1 because the remainder is 1, $11 \% 3$ is equal to 2, and $12 \% 3$ is equal to 0. Note, another variation of this would be to get a “checker-board” effect, i.e. the first rectangle in row 1 is red, in row two is green, in row 3 is red, and so on. This can be achieved with using:

```
if ( ((i+j) % 2) == 0)
```

EXERCISE 7B

Write code to create each of the following (or something very close to it). Hint on the last one: try using lines where you modify the stroke weight and color such as: `strokeWeight(5)` and `stroke(255,0,0) // color`



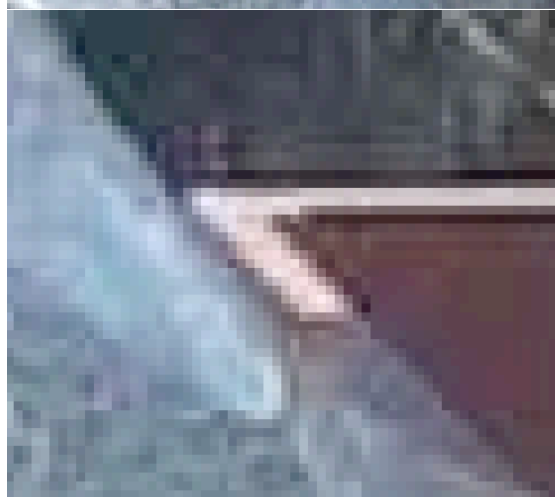
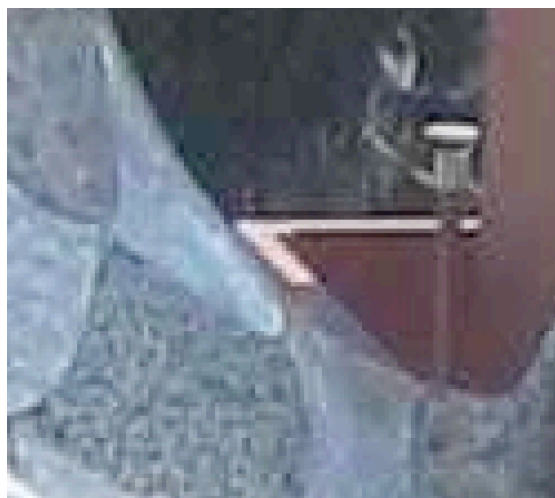


Chapter 8: PImage: Image Manipulation

Digital images in the simplest form, a bitmap file, can be thought of as a two-dimensional “box” of pixels. Consider the photo below. In the top left you see the original photo (well, actually, I already decreased the resolution significantly for this demonstration). In the middle-left you see the photo zoomed in x2. In the bottom-left you see the photo zoomed in another x2. In the top-right another zoom in x2. In the middle-right another x2. And in the bottom-right another x2.

As you zoom in you get to see the individual pixels. In the version on the bottom-right you can see the photo is just a 2D matrix of different colored cells. Each pixel is a color represented by a RGB triplet (0..255, 0..255, 0.255). In digital photos with modern cameras and modern computers the pixels are so small your eye does not see them. Even the top-left picture looks pretty good, but by today’s standards it is very low-resolution.

Processing provides the PImage class for storing, manipulating, and drawing images. From inside Processing click on Help->Reference and then find PImage for a complete description of available methods. Because an image is just a 2D matrix of pixels, we can use doubly-nested for loops iterate over every row and column of the image.



Run the following to start with. Go ahead and use my image if you want, it is on the web site listed as ParkGuell, or use any smaller jpg file. First, lets us PImage to load the file and display it:

```
size(800,800) ;

PImage theImage ;
theImage = loadImage("parkguell.jpg") ;
System.out.println("width = " + theImage.width) ;
System.out.println("height = " + theImage.height) ;

// show the unmodified image in the top left
image(theImage,0,0) ;

// also show the image in the top right, put
// a space of 10 pixels between the two
image(theImage, theImage.width + 10, 0) ;
```

The above code create a variable name “theImage” of type PImage. Actually, technically, this is an object named “theImage” of type class PImage. More on that soon when we move on to classes and objects. We then use the “loadImage()” method to load the image name “parkguell.jpg”. The PImage class gives us properties to access width and height, which we then print out. In the case of this image, the width and height are 244 and 364 pixels respectively. We can use the command image(theImage,0,0) to display the image inside the Processing output window setting its top left corner at location 0,0. The image displayed is the one contained in the object “theImage). In the last line of code we display the image a second time, but this time the top left corner is set to be (theImage.width + 10, 0), i.e. (254,0).

The following code loads an image and then using draw makes it move diagonally across the screen. Why this is particurly relevant: it means you can move any jpg/gif/png image around your screen in Processing: for games, simulation, interactive stories, whatever. You just create an image you want using some tool (a photo, a drawing from illustrator/inkscape/photoshop) and then load it into a PImage object.

```

PImage theImage ;
int imx = 0 ;
int imy = 0 ;

void setup()
{
  size(800,800) ;
  background(100) ;
  theImage = loadImage("parkguell.jpg") ;
}

void draw()
{
  background(100) ;
  image(theImage,imx,imy) ;
  imx += 2 ;
  imy += 2 ;
}

```

Certainly displaying and moving images is good, but manipulating them can be even more fun. Once an image is loaded into a PImage object, it can be accessed pixel by pixel. The Park Guell image once loaded is a 244x364 pixel matrix. Thus, we can use doubly-nested for loops to access every pixel if we want! The PImage class gives us get(x,y) and set(x,y) methods to get the pixel at a certain (x,y) or set the pixel at a certain (x,y). Consider the following code and its output. The output is below the code:

```

size(800,800) ;
background(100) ;
PImage theImage ;
theImage = loadImage("parkguell.jpg") ;

// display the original image in the top left corner

```

```

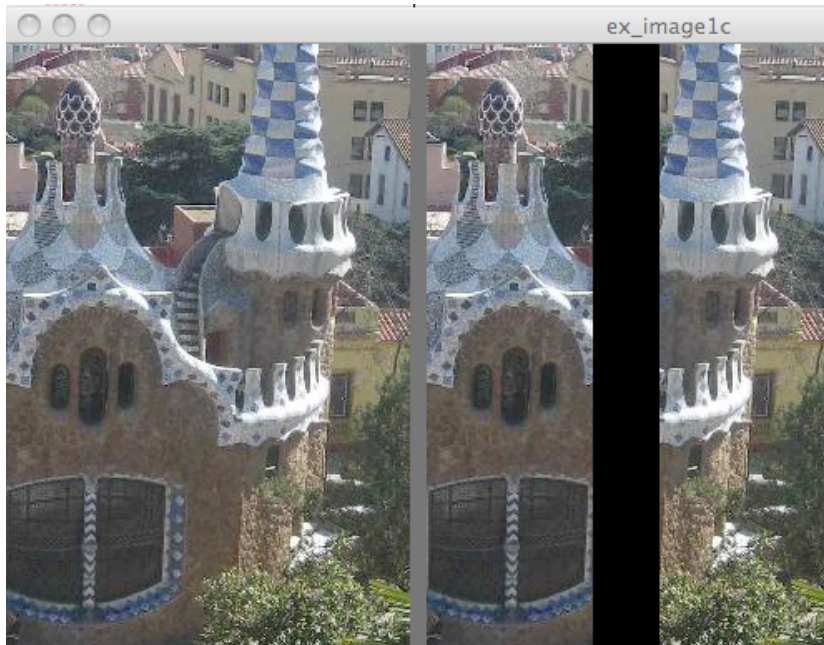
image(theImage,0,0) ;

color color1 = color(0,0,0) ; // black

for (int i = 100 ; i < 140 ; i++)
  for (int j = 0 ; j < theImage.height ; j++)
    theImage.set(i,j,color1) ;

// display the modified image to the right
image(theImage,theImage.width + 10 ,0) ;

```



As we can see, there is a thick black bar in the right side image. The command “theImage.set(i,j,color1)” will replace the contents of the pixel at location (i,j) with the color specified in color1. We can see that we have made “color1” a variable of type color and set it to black. If we change the nested for loops to the following, we would get a black vertical line 100 pixels from the left instead of a 40-pixel wide bar:

```

for (int i = 100 ; i < 101 ; i++)
  for (int j = 0 ; j < theImage.height ; j++)
    theImage.set(i,j,color1) ;

```


Consider the next example. The output is below the code and it displays three images. On the left the original, in the middle one with 10,000 randomly selected pixels set to black, and one with 100,000 randomly selected pixels set to black:

```
size(800,800) ;
PImage im1, im2, im3 ;
im1 = loadImage("parkguell.jpg") ;
im2 = loadImage("parkguell.jpg") ;
im3 = loadImage("parkguell.jpg") ;

color color1 = color(0,0,0) ; // black

int rx, ry ;

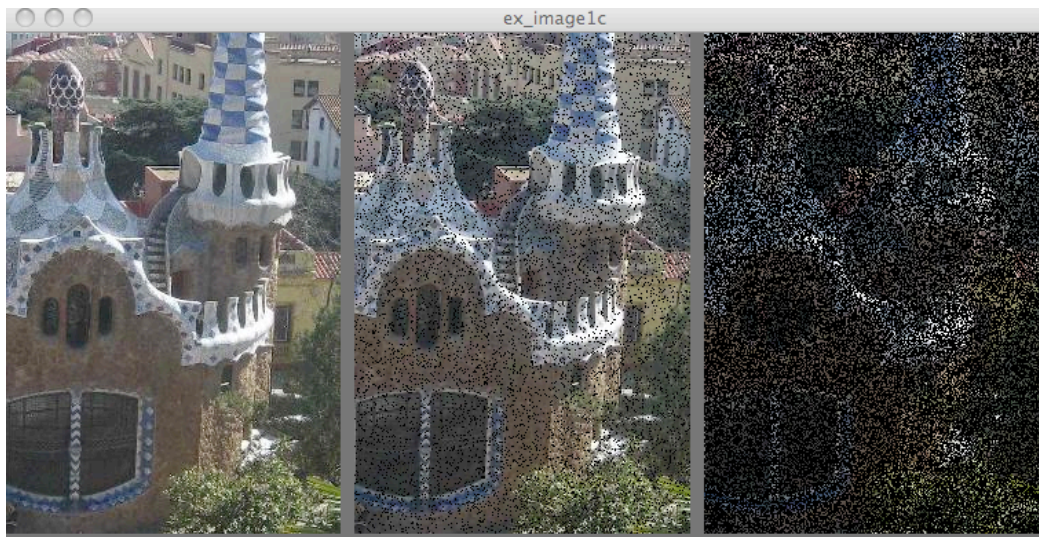
for (int i = 0 ; i < 10000 ; i++)
{
  // cast the float return from random( ) into and integer
  rx = (int) random(im1.width) ;
  ry = (int) random(im1.height) ;
  im2.set(rx,ry,color1) ;
}

for (int i = 0 ; i < 100000 ; i++)
{
  rx = (int) random(im1.width) ;
  ry = (int) random(im1.height) ;
  im3.set(rx,ry,color1) ;
}

// display the original image to the left
image(im1, 0 ,0) ;

// display the one with 10,000 black pixels in the middle
image(im2, im1.width + 10 ,0) ;

// display the image with 100,000 black pixels on the right
image(im3, im1.width *2 + 20 ,0) ;
```



Note, since the image is 244 x 364 pixels, that means there are 88,816 pixels. So, the question is why is not the third image completely black given we set 100,000 pixels black? Answer: when we called random there were lots of duplicate pixels that get set black many times, leaving enough pixels with the original color to still show through.

Not only can we set() the pixels, we can also get() them. The following code looks at every pixel and calculates the average RED, GREEN, and BLUE value over all pixels in the image. In the following code we load the same image into to PImage variables. Then we loop through every pixel using a doubly-nested loop. For each pixel we get the color using the PImage get() method. We then use the red(), green(), and blue() functions to get the R G B values for the color at that pixel. In particular:

```
r = (int) red(tempColor) ;
```

will return a number between 0..255 that is the red component of the color contained in the color variable “tempColor”. The next two lines do the same for green and blue. We then set the color variable “newColor” to have maximum red, the same blue, and the same green, the effect is to max-out the red:

```
newColor = color( 255, g, b) ;
```

Finally, we change the color in the image in PImage variable "im2" at location (i, j) to be this new color:

```
im2.set(i,j,newColor) ;
```

We do this for every pixel in the image by using the doubly nested for-loop. Finally, we draw the original unmodified image in the top left corner and then the modified red-maxed-out version to the right of it as shown below.

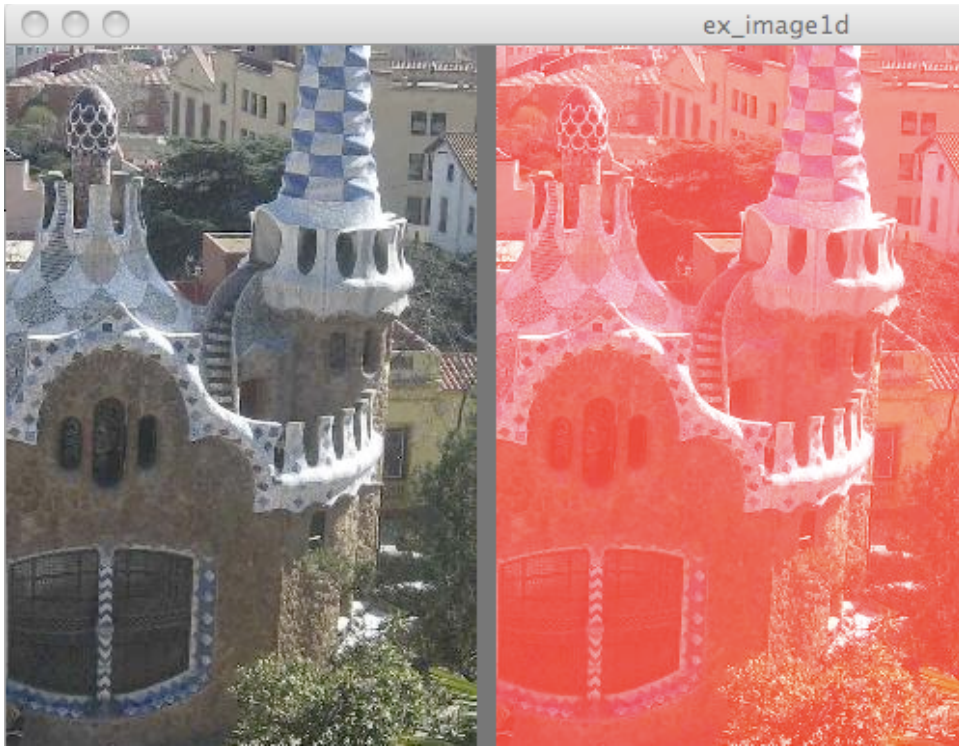
```
size(800,800) ;
background(100) ;
PImage im1, im2, im3 ;
im1 = loadImage("parkguell.jpg") ;
im2 = loadImage("parkguell.jpg") ;

// vars to hold the red, green, and blue components of each pixel
float r, g, b ;

// two color variables
color tempColor, newColor ;

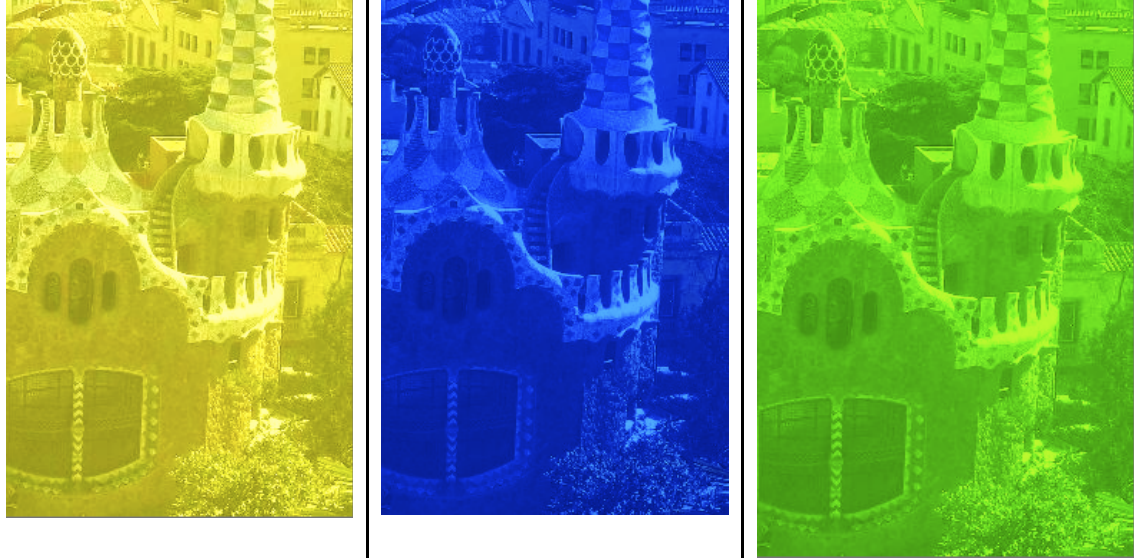
for (int i = 0 ; i < im1.width ; i++)
  for (int j = 0 ; j < im1.height ; j++)
  {
    tempColor = im1.get(i,j) ; // get the color at i,j
    r = red(tempColor) ; // the red value
    g = green(tempColor) ; // the green
    b = blue(tempColor) ; // the blue
    newColor = color( 255, g, b) ; // max out the red
    // replace existing color with new more-red color
    im2.set(i,j,newColor) ;
  }

image(im1,0,0) ;
image(im2,im1.width + 10, 0) ;
```



EXERCISE 8A

Starting with the Park Guell image, created each of the following three images. Note, you may need to reduce the some color components and increase others, try using arithmetic such as: $\text{newColor} = \text{color}(r + 50, g - 100, b + 50)$. That statement will give you a nice purple image.



The `get()` method can also be used to get a “chunk” of pixels instead of just one. Consider the following code and image. In this example we use `get()` to grab a 80x80 group of pixels with the top left corner located at (5,20). The 6,400 pixels (80x80) that are grabbed are the following image:



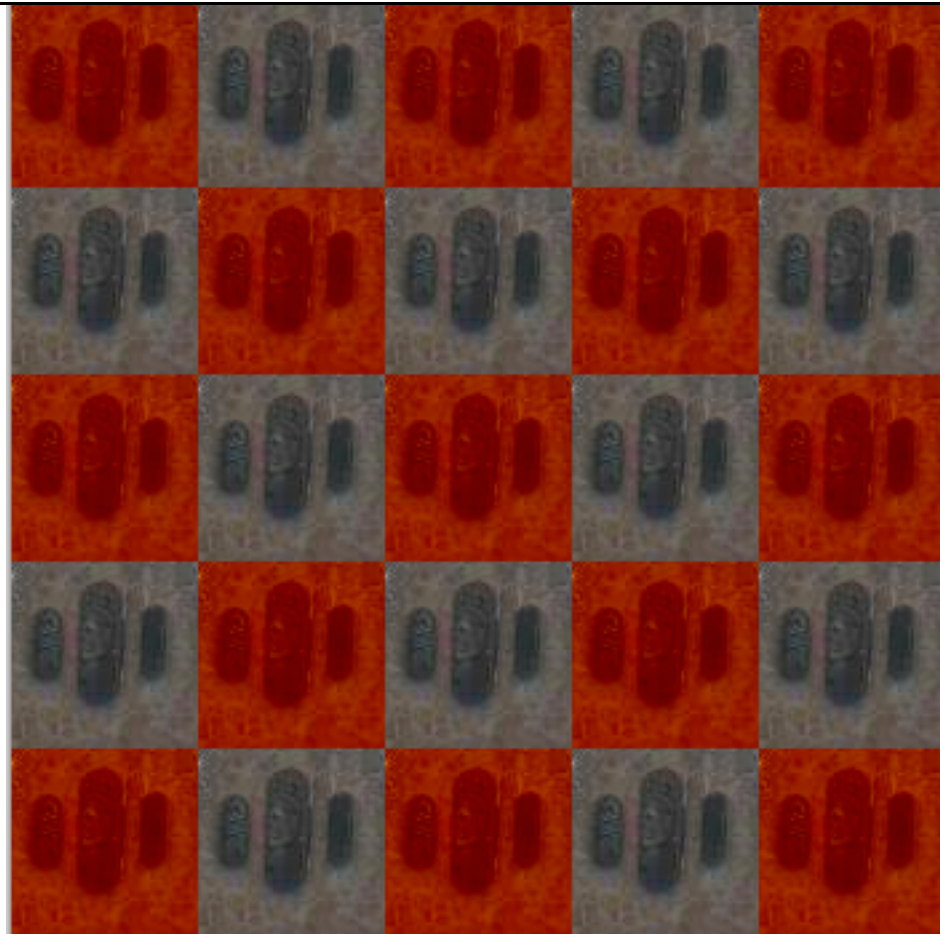
If you look at the original image, starting from (0, 0), i.e. the top left corner, move over 5 pixels to the right and 20 pixels down. That is the starting point. We then grab a sub-image 80 pixels wide and 80 pixels high starting at (5,20). This gives us the crown of the part of the house on the left. We stick `image(im2, i*80, j*80)` inside a doubly-nested for-loop to get the repeated pattern below.

```
size(800,800) ;  
background(100) ;  
PImage im1, im2 ;  
im1 =  
loadImage("parkguell.jpg") ;  
  
im2 = im1.get(5,20,80,80) ;  
  
for (int i = 0 ; i < 5 ; i++)  
  for (int j = 0 ; j < 5 ; j++)  
    image(im2, i*80, j*80) ;
```



EXERCISE 8B

Use `get()` to get a subregion of the image and manipulate the colors to create a tiling with alternating colors that is appealing to you. As an example, below I selected the triple windows above the main windows and alternated colors of original versus red-tinted. I think this would make a great rug pattern in wool!



Chapter 9: Simple Functions

Often in programming we want to execute a set of instructions multiple times. Lets say this set of instructions includes 10 statements. We could just put these 10 statements every place we use them, but if we use them many times, say 10, we end up writing lots of identical code for no reason. All that redundancy makes it hard to make changes, i.e. if you decide to change one of the statements you have to find that statement in 10 different places, and makes it easy to introduce errors, i.e. you might make a typo in some of the places and get it right in others, so the code become inconsistent.

Functions provide a clean/elegant way to reuse code without these problems. Functions should have a single logical task. Examples might include: “move the ball to the left”, “draw a figure on the screen at location x,y”, “enroll the student in class number COMP 1671”, “calculate the student gpa”, “give all employees a raise”, or “calculate the mean temperature over the past year at a given longitude/latitude”.

In fact, you have already been using functions: `setup()`, `draw()`, and `mousePressed()` are all functions. These are special functions that are part of the Processing model, but they are still just Java functions. A function has the general format:

```
returnType  functionName( parameter-list )
{
    function code
}
```

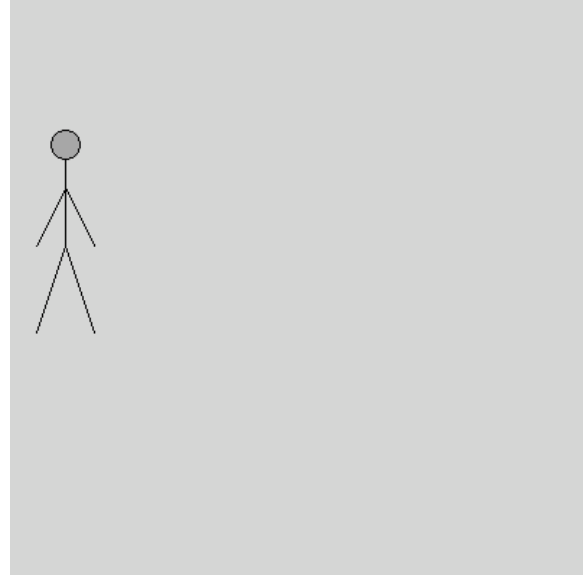
where `returnType` is the type of value returned by the function. If no value is returned that is specified with “void” as the return type. Note `setup()`, `draw()`, and `mousePressed()` all have a “void” return type, i.e. they do not return a value. “functionName” is the name of the function, for example “setup”, “draw” and “mousePressed” are all function names.

The parameter list is the list of values, and their types, passed into a function. The functions `setup()`, `draw()`, and `mousePressed()` do not have any parameters, hence the parameter list is empty. We will give examples of functions with non-empty parameter lists below. The “function code” is all the statements that belong to the function.

Consider the following code that uses a function we named `drawPerson()` :

```
void drawPerson()
{
  line( 40,100, 40,170) ; // body
  line( 40,170, 20,230) ; // left leg
  line( 40,170, 60,230) ; // right leg
  line( 40,130, 20,170) ; // left arm
  line( 40,130, 60,170) ; // right arm
  fill(150) ; // grey for head fill
  ellipse(40,100,20,20) ; // head
}

void setup()
{
  size(400,400) ;
  drawPerson() ;
}
```



In this example the function `drawPerson()` draw a stick figure person. The parameter list is empty as there are no parameters. The return type is `void` because there is no value returned. The function is CALLED in the `setup()` function:

```
drawPerson() ;
```

Like all Processing sketches the code in `setup()` is run once. Hence, `drawPerson()` is called once. Lets say we want to draw the figure more to the right. We need to modify all the code. It would be much nicer if we

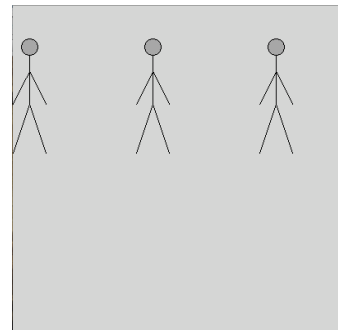
could just modify the function so that when called we could specify the (x, y) location where we want the stick figure drawn, something like:

```
drawPerson( 200, 200 ) ;
```

which would then draw the stick figure at location (200,200). To do that we need to modify the function as follows:

```
void drawPerson(int inX, int inY)
{
  line( inX+20,inY, inX+20,inY+70) ; // body
  line( inX+20,inY+70, inX,inY+130) ; // left leg
  line( inX+20,inY+70, inX+40,inY+130) ; // right leg
  line( inX+20,inY+30, inX,inY+70) ; // left arm
  line( inX+20,inY+30, inX+40,inY+70) ; // right
  arm
  fill(150) ; // grey for head fill
  ellipse(inX+20,inY,20,20) ; // head
}

void setup()
{
  size(400,400) ;
  drawPerson(0,50) ;
  drawPerson(150,50) ;
  drawPerson(300,50) ;
}
```



In this example the drawPerson function now has two parameters: inX and inY. Thus, when the function is called, we must supply values for those parameters such as:

```
drawPerson(0, 150) ;
```

This says call drawPerson, and when the code of drawPerson is run initialize the values of inX and inY with 0 and 150 respectively. In order for this to work we had to change the code inside the drawPerson

function to now be dependent on the input parameters. One way to do that for this drawing example is to find the smallest x-coordinate and the smallest y-coordinate, and then change all the code to be relative to the input parameters. For example, in the original function, the statement for the head was:

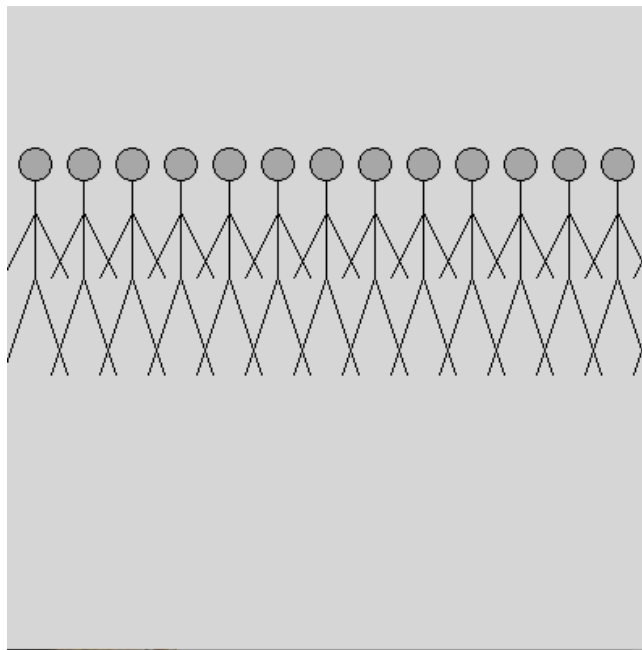
```
ellipse(40,100,20,20) ;
```

The smallest x value in the original code was 20, and the smallest y was 100, so, to make the statement relative to the input parameters we change it to:

```
ellipse( inX+20, inY, 20, 20) ;
```

Now, if we call the function as `drawPerson(20,100,20,20)`; the person will be drawn in the same place as in the original function. But by using the function with a parameter, we can put in other values when calling the function and hence draw the figure anywhere we want simply by changing the values in the function call rather than changing all the lines of code. We can also put the call inside a for loop as follows:

```
for (int i = 0 ; i < 400 ; i = i+30)  
  drawPerson(i, 100) ;
```



We can also make a the person move using the draw() function as follows:

```
int currentX ; // global var to hold x-coord of figure
```

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

```
void draw()
{
  background(200) ;
  drawPerson(currentX, 100) ;
  currentX = currentX + 2 ;
}
```

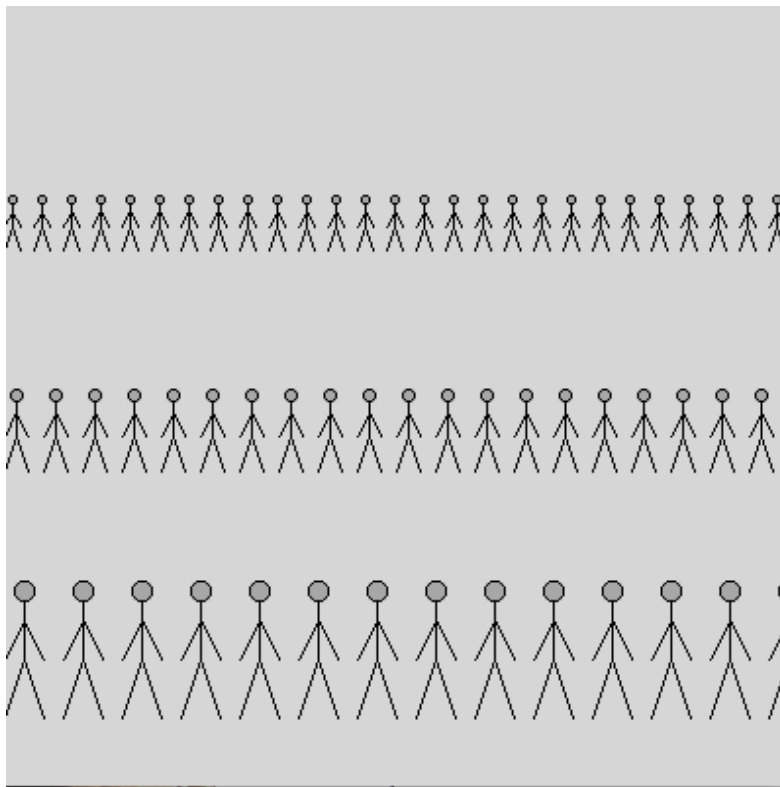
```
void drawPerson(int inX, int inY)
{
  line( inX+20,inY, inX+20,inY+70) ; // body
  line( inX+20,inY+70, inX,inY+130) ; // left leg
  line( inX+20,inY+70, inX+40,inY+130) ; // right leg
  line( inX+20,inY+30, inX,inY+70) ; // left arm
  line( inX+20,inY+30, inX+40,inY+70) ; // right arm
  fill(150) ; // grey for head fill
  ellipse(inX+20,inY,20,20) ; // head
}
```

Finally, you may want to be able to specify the size of the figure when you call drawPerson(). The following code allows you to specify the size. In the example there is a third parameter called “sF”, short for scale-factor, added to specify how big the image should be. All numbers that result in

the distance of a line are multiplied by “sF”, the input value, to make them bigger or smaller. In the example below we make three rows of stick figures, each line above with smaller figures than the line below.

```
void setup()
{
  size(400,400) ;
  for (int i = 0 ; i < 400 ; i = i+15)
    drawPerson(i, 100,0.20) ;
  for (int i = 0 ; i < 400 ; i = i+20)
    drawPerson(i, 200 ,0.30) ;
  for (int i = 0 ; i < 400 ; i = i+30)
    drawPerson(i, 300,0.50) ;
}

void drawPerson(int inX, int inY, float sF)
{
  // sF means "scale factor"
  line( inX+20*sF, inY, inX+20*sF, inY+70*sF) ; // body
  line( inX+20*sF, inY+70*sF, inX, inY+130*sF) ; // left leg
  line( inX+20*sF, inY+70*sF, inX+40*sF, inY+130*sF) ; // right leg
  line( inX+20*sF, inY+30*sF, inX, inY+70*sF) ; // left arm
  line( inX+20*sF, inY+30*sF, inX+40*sF, inY+70*sF) ; // right arm
  fill(150) ; // grey for head fill
  ellipse(inX+20*sF, inY, 20*sF, 20*sF) ; // head
}
```



As another example, here is a way to draw “keyhole ken”. Keyhole ken is a simple cartooning exercise for drawing people. The book “The cartoonist’s Workbook Drawing, Writing Gags, Selling”, by Robin Hall, introduces keyhole ken.

```

size(500,500) ;

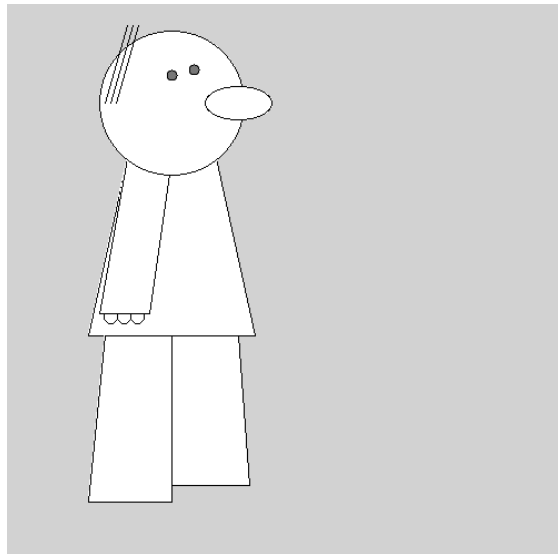
quad(75,300,225,300,190,140,110,140)
; // body
ellipse(95,283,12,12) ; //left finger
ellipse(107,283,12,12) ; // midle finger
ellipse(119,283,12,12) ; // right finger
quad(85,280, 130,280, 150,140, 110,
140) ; // arm
ellipse(150,90,130,130) ; // head
ellipse(210,90,60,30) ; // nose

fill(100) ; // make fill black for the eyes
ellipse(150,65,9,9) ; // center eye
ellipse(170,60,9,9) ; // right eye

// add a few lines for hair
line(90,90, 110,20) ;
line(95,90, 115,20) ;
line(100,90, 120,20) ;

fill(255) ; // set fill back to white
quad(140,300, 140,435, 220,435,
210,300 ) ; // right leg
quad(90,300, 75,450, 150,450, 150,300
) ; // left leg

```



This can also be made into a function with parameters for the (x,y) location as follows:

```

void drawKeyholeKen(int x, int y)
{
quad(x,y+280, x+150,y+280, x+115,y+120, x+35,y+120) ; // body

ellipse(x+20,y+263,12,12) ; //left finger
ellipse(x+32,y+263,12,12) ; // midle finger
ellipse(x+44,y+263,12,12) ; // right finger

```

```

quad(x+10,y+260, x+55,y+260, x+75,y+120, x+35, y+120) ; // arm
ellipse(x+75,y+70,130,130) ; // head
ellipse(x+135,y+70,60,30) ; // nose
fill(100) ; // make fill black for the eyes
ellipse(x+75,y+45,9,9) ; // center eye
ellipse(x+95,y+40,9,9) ; // right eye

// add a few lines for hair
line(x+15,y+70, x+35,y) ;
line(x+20,y+70, x+40,y) ;
line(x+25,y+70, x+45,y) ;

fill(255) ; // set fill back to white
quad(x+65,y+280, x+65,y+415, x+145,y+415, x+135,y+280 ) ; // right leg
quad(x+15,y+280, x,y+430, x+75,y+430, x+75,y+280 ) ; // left leg
}

```

Then, with the following setup() and draw() commands you can make two keyhole kens to cyclicly walk the screen together:

```

int x1 ; // x of first ken
int x2 ; // x of second ken

void setup()
{
  size(900,500) ;
  background(100) ;
  x1 = 0 ;
  x2 = 175 ;
}

void draw()
{
  background(100) ;
  drawKeyholeKen(x1,10) ;
  drawKeyholeKen(x2,30) ;
  x1 += 2 ;
}

```



```
x2 += 2 ;  
if (x1 > width + 150)  
    x1 = -150 ;  
if (x2 > width + 150)  
    x2 = -150 ;  
}
```

EXERCISE 9A

Come up with your own function to draw a simple image. Perhaps a house, or a tree, both are pretty easy to do. Make three versions of your function: 1) hardcoded values, i.e. no parameters; 2) a function that takes parameters inX and inY and draws your figure at location (inX,inY); 3) a function that takes in inX, inY, and a scale factor.

Write code in your setup() function that calls your figure drawing function and shows how it works with the three parameters (inX, inY, scaleFactor).

Chapter 10: Functions Returning a Value

The previous function example all drew stuff on the screen but did not return a value. We saw functions without parameters and with parameters. Lets consider some numerical functions that return values. Lets say I want to write a function that returns the average of three numbers. I could do this as follows:

```
float avg(float in1, float in2, float in3)
{
    float total, averageValue ;
    total = in1 + in2 + in3 ;
    averageValue = total / 3.0 ;
    return( averageValue ) ;
}
```

```
void setup()
{
    float theAvg ;

    theAvg = avg( 10.0, 9.0, 6.0 ) ;
    System.out.println( theAvg ) ;
}
```

When the above code is run the value 8.3333 is printed out to the black output bar. What is important to see here is:

- 3) The function does NOT start with void, instead it says “float”, this means the function will return a floating point value.
- 4) Inside the function, the last line is “ return(averageValue) ;“, this is returning the value of variable averageValue to the calling statement.
- 5) Insider setup(), the function is called in the line “theAvg = avg(10.0, 9.0, 6.0) ;”. This statement says call the function avg(), passing in parameters 10.0, 9.0 and 6.0, and take the value returned by the

function call and assign it to the variable “theAvg”.

Lets consider another example:

```
float maxValue(float in1, float in2, float in3, float in4)
{
    float maxVal = 0 ;
    if ( (in1 >= in2) && (in1 >= in3) && (in1 >= in4) )
        maxVal = in1 ;
    if ( (in2 >= in1) && (in2 >= in3) && (in2 >= in4) )
        maxVal = in2 ;
    if ( (in3 >= in1) && (in3 >= in2) && (in3 >= in4) )
        maxVal = in3 ;
    if ( (in4 >= in1) && (in4 >= in2) && (in4 >= in3) )
        maxVal = in4 ;

    return( maxVal ) ;

}

void setup()
{
    float theMax ;

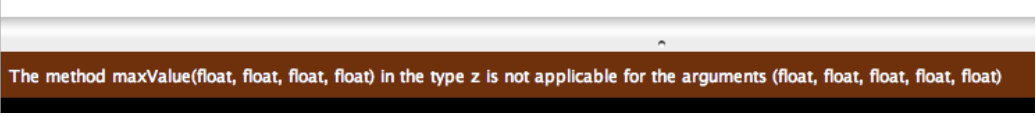
    theMax = maxValue( 10.2, 9.0, 6.0, 7.0 ) ;
    System.out.println( theMax ) ;
}
```

The above will print out “10.2”. It is obvious just looking at the code what the answer is, but a computer is not a human, it needs a set procedure to calculate an answer. In this case the if statements find with of the four input parameters is the largest and returns that value.

You may want to print the maximum of 10 numbers, or 20, or 98 numbers. Using the function like this won’t work. If I were to say:

```
theMax = maxValue( 10.2, 9.0, 6.0, 7.0, 99.1) ;
```

I would get the error message:



```
The method maxValue(float, float, float, float) in the type z is not applicable for the arguments (float, float, float, float, float)
```

The reason is that the function `maxValue` is expecting 4 parameter values to be in, no more, not less, and they need to be float values (not a float variable will accept an int as we have stated before). So, how do we deal with this? The answer is to use something called **arrays** which we will do later. For now, just realize that when you declare a function you specify the number and type of parameters and when you call it your call must match that number and type. The number and type of parameters for a function is called its **signature**.

EXERCISE 10A

Write a function that has three input parameters of type float, that returns the value of the first parameter times the second divided by the third. Say you name the function `myFunc()`, show that this function works correctly by calling and printing out the results using `System.out.println()` of:

```
myFunc( 4, 7, 2)  => should give you 14  
myFunc( 3, 3, 2)  => should give you 4.5  
myFunc( 2, 8, 4)  => should give you 4
```

Chapter 11: Using Functions With Colors

Functions can take any defined type (or class which we will explain later) as input and return a defined type or class. Hence, we can use functions to manipulate and return colors also. Consider the following:

```
PImage im1, im2, im3 ;
```

```
color swapRedGreenColor(color inColor)
{
  float r,g,b ;
  r = red(inColor) ;
  g = green(inColor) ;
  b = blue(inColor) ;
  color tempColor = color(b,g,r) ;
  return(tempColor) ;
}
```

```
void setup()
{
  size(800,800) ;
  background(100) ;
  im1 = loadImage("parkguell.jpg") ;
  im2 = loadImage("parkguell.jpg") ;

  color tempColor, tempColor2 ;
  float r,g,b ;
  for (int i2 = 0 ; i2 < im1.width ; i2++)
    for (int j2 = 0 ; j2 < im1.height ; j2++)
    {
      tempColor = im1.get(i2,j2) ;
      // call swapRedGreenColor to modify the color
      tempColor2 = swapRedGreenColor(tempColor) ;
      // set the i2,j2 pixel of im2 to be the new color
      im2.set(i2,j2,tempColor2) ;
    }
  image(im1,0,0) ;
  image(im2,im1.width,0) ;
}
```

```
}
```

In this example there is a function named “swapRedGreenColor()”. Notice that it take on parameter and that parameter is of type color. Also notice that it returns a variable of type color also! You can see this return in the last line of the function where it says “return(tempColor);”, where tempColor is a variable of type color. The function creates a color variable named “tempColor” and set the red value of tempColor to be the green value of the color passed in the parameter “inColor”, and sets the green value of tempColor to be the red value of inColor. The setup() function just loads the Park Guell image into PImage variables im1 and im2, and then loops through all the pixels of im2 swapping the red and green values for each pixel by calling the function swapRedGreenColor() before calling im2.set().

The above example showed how to write a function that takes a color as a parameter and returns a color as a parameter, but the setup() function is rather complicated. It can be made less complicated by changing the responsibilities of the function so that the function does a bit more of the work for us. For example, we may want to write a function that takes an image as a parameter and returns a modified image. This way using the function is cleaner/simpler. Consider the following:

```
PImage im1, im2, im3, im4 ;
```

```
PImage swapBG( PImage inImage)
```

```
{  
  float r,g,b ;  
  color tempColor, tempColor2 ;  
  PImage newImage = inImage.get(0,0,inImage.width,inImage.height) ;
```

```
  for (int i2 = 0 ; i2 < inImage.width ; i2++)  
    for (int j2 = 0 ; j2 < inImage.height ; j2++)  
    {  
      tempColor = im1.get(i2,j2) ;  
      r = red(tempColor) ;
```

```

    g = green(tempColor) ;
    b = blue(tempColor) ;
    tempColor2 = color(b,g,r) ;
    newImage.set(i2,j2,tempColor2) ;
}
return(newImage) ;
}

```

```

PImage darkenImage( PImage inImage, float darkenFactor)
{
    float r,g,b ;
    color tempColor, tempColor2 ;
    PImage newImage = inImage.get(0,0,inImage.width,inImage.height) ;

    for (int i2 = 0 ; i2 < inImage.width ; i2++)
        for (int j2 = 0 ; j2 < inImage.height ; j2++)
        {
            tempColor = im1.get(i2,j2) ;
            r = red(tempColor) ;
            g = green(tempColor) ;
            b = blue(tempColor) ;
            tempColor2 = color(r-darkenFactor, g-darkenFactor, b-darkenFactor) ;
            newImage.set(i2,j2,tempColor2) ;
        }
    return(newImage) ;
}

```

The first function, `swapRB()`, takes an input of an image, loops through all the pixels and swaps the red and blue color values, and then returns the image. The second function, `darkenImage()`, takes two parameters, an image and a float, and returns an image where every pixel has been darkened by the second parameter amount. It does this by looping through the pixels and subtracting the value of the second parameter from each of the RGB values in every pixel. The functions are called as follows:

```

void setup()
{
  size(800,800) ;
  background(100) ;
  im1 = loadImage("parkguell.jpg") ;
  im2 = swapRB(im1) ;
  im3 = darkenImage(im1,50) ;
  image(im1,0,0) ;
  image(im2,im1.width,0) ;
  image(im3,0,im1.height) ;
}

```

By moving the doubly nested for-loops inside the functions the code in setup() is much more clean, and, one could argue, much more logical. The functions swapRB() and darkenImage() do all the work associated with that task and are used as tools to manipulate the images from setup(). A lot of computer programming can be viewed as making (and using) elegant virtual tools. Don't have the right tool for the job: just make a new function (or a **method** as they are called in OO programming coming up soon) to do the job.

Lets say we want to average two images together: i.e. create a new image where each pixel is the average value of the pixel at the location in each image. The following function does exactly this:

```

PImage blendImages(PImage inIm1, PImage inIm2)
////////////////////////////////////
// Input: two images
// Output: An image that is the blend (average) of the two
// input images.
// Algorithm: This code assumes the two images are the same
// size and loops over the width and heith of the first image,
// getting
// the color at each (i,j) location and setting the new image pixel

```



```

// color to be the average of the two
//
////////////////////////////////////
{
  PImage resultIm ;
  color tcolor1, tcolor2, tcolor3 ;
  float r1, r2, g1, g2, b1, b2 ;

  resultIm = inIm1.get(0,0,inIm1.width,inIm1.height) ;

  for (int i2 = 0 ; i2 < inIm1.width ; i2++)
    for (int j2 = 0 ; j2 < inIm1.height ; j2++)
      {
        tcolor1 = inIm1.get(i2,j2) ;
        r1 = red(tcolor1) ;
        g1 = green(tcolor1) ;
        b1 = blue(tcolor1) ;
        tcolor2 = inIm2.get(i2,j2) ;
        r2 = red(tcolor2) ;
        g2 = green(tcolor2) ;
        b2 = blue(tcolor2) ;
        tcolor3 = color( (r1+r2)/2, (g1+g2)/2, (b1+b2)/2) ;
        resultIm.set(i2,j2,tcolor3) ;
      }
  return(resultIm) ;
}

```

The following code then calls this function, and also the `darkenImage()` function, to create the images below:

```

void setup()
{
  size(800,800) ;
  background(100) ;
}

```

```

im1 = loadImage("photoScott.jpg") ;
im1 = im1.get(140,20,400,400) ; // select 400x400 pixels

im2 = loadImage("photoBear.jpg") ;
im2 = im2.get(170,60,400,400) ; // select 400x400 to align eyes

im3 = darkenImage(im1, 50) ; // darken scott
im4 = blendImages(im3,im2) ; // blend scott and the bear

// show the 4 images
image(im1,0,0) ;
image(im2,400,0) ;
image(im3,0,400) ;
image(im4,400,400) ;
}

```



Note im1 and im2 (top left and top right) are just a 400x400 subset of the images loaded from the .jpg files. The offsets were chosen to roughly line

up the eyes. Image im3, shown in the bottom left, is the result of calling `darkenImage(im1,50)`, and image im4, shown in the bottom right, is the result of calling `blendImages(im3,im2)`. Note, the bear face and middle-age man's face are not the same shape (thankfully!) so the resultant images is rather funky. There are algorithms to morph two images together but are more advanced and beyond the scope of this class.

EXERCISE 11A

Write a function named `brightenImage()` that takes two parameters, a `PImage` and a float named "inBrightenFactor", and returns a `PImage` where each pixel has each R, G, and B value increased by the value in input parameter "inBrightenFactor". Demonstrate the use of your function by creating three images and displaying them, along with the original, where you use different values for the input value including a negative value for one of the three.

EXERCISE 11B

Write a function that takes an image as an input parameter and returns an image that diagonally mirrors the top right diagonally down to the bottom left as show in the two images below. Your `setup()` function call this function and display the original and modified image side-by-side.

