

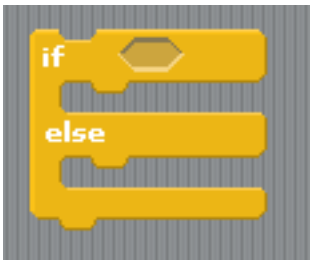

Processing Notes

Chapter 5: Conditionals and Booleans

We often use conditional statements such as

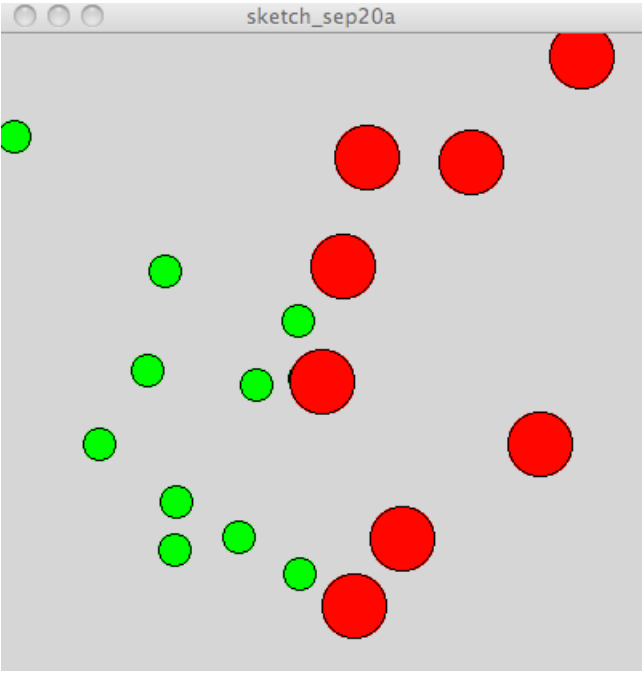
```
if a certain condition is true
    do one set of commands
else
    do another set of commands.
```

In Processing/Java the if/else statement has following form as show on the left below. The Scratch equivalent is show on the right.

<pre>if () { } else { }</pre>	 A yellow Scratch 'if-else' block. It features a diamond-shaped condition slot at the top, followed by two horizontal slots for code blocks. The word 'if' is on the left of the top slot, and 'else' is on the left of the bottom slot.
<pre>if () { }</pre>	 A yellow Scratch 'if' block. It features a diamond-shaped condition slot at the top, followed by one horizontal slot for a code block. The word 'if' is on the left of the slot.

What the statement in the first row means is if the expression in the parenthesis is true do the commands encapsulated by the first set of braces, otherwise do the commands encapsulated by the second set of braces. The condition in parenthesis is the equivalent to the expression put in the sideways diamond space at the top of the Scratch block. You do not need to have an else clause. The statements in the second row are for an “if” without an “else”

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:

<pre> 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) ; } } </pre>	
---	--

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
}

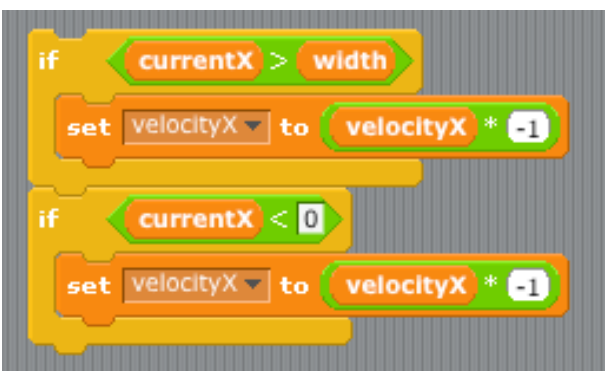
void draw()
{
  background(150) ; // wipe out previous drawings

  // 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 mad more concise and neat. The logic for updating velocityX is currently (in processing and scratch):

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

This can be rewritten as:

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

In the Processing 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
<ol style="list-style-type: none">1) 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.2) 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 Boolean expressions in Processing/Java and Scratch. 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	Scratch	
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)	no equivalent in scratch 1.4	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))	

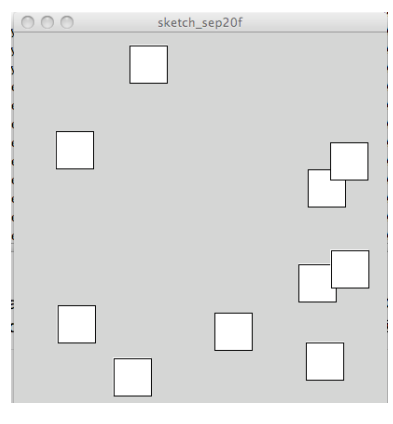
Chapter 6: For-Loops

Often in programming you want to run a chunk of code multiple times. For example, lets 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.

Lets 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 counter ;
float rx ;
float ry ;

for (counter = 0 ; counter < 10 ; counter = counter + 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:

```
for ( counter = 0 ; counter < 10 ; counter = counter + 1 )
{
  // loop contents
}
```

initialize **check** **change**

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. The we repeat: check, run, change. Check, run, change..... until the check fails. In this case:

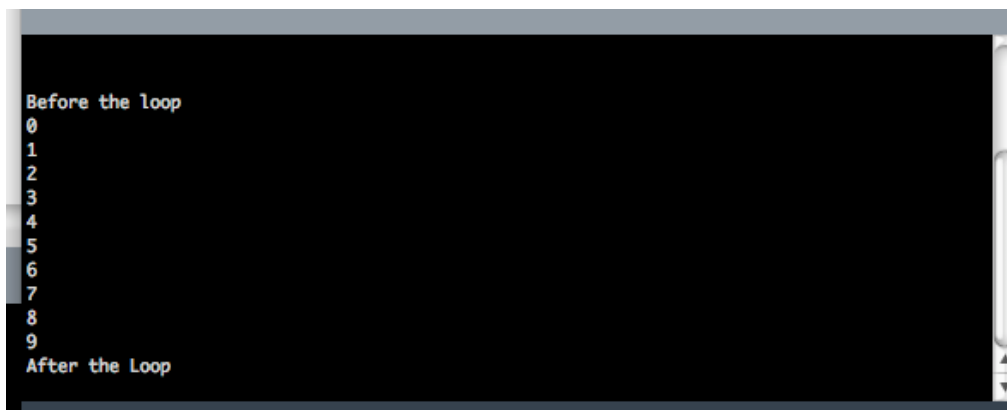
- 1) initialize counter to the value 0
- 2) check if counter is < 10. YES. Run the loop. Change counter to 1.
- 3) check if counter is < 10. YES. Run the loop. Change counter to 2.
- 4) check if counter is < 10. YES. Run the loop. Change counter to 3.
- 5) check if counter is < 10. YES. Run the loop. Change counter to 4.
- 6) check if counter is < 10. YES. Run the loop. Change counter to 5.

- 7) check if counter is < 10. YES. Run the loop. Change counter to 6.
- 8) check if counter is < 10. YES. Run the loop. Change counter to 7.
- 9) check if counter is < 10. YES. Run the loop. Change counter to 8.
- 10) check if counter is < 10. YES. Run the loop. Change counter to 9.
- 11) check if counter is < 10. YES. Run the loop. Change counter to 10.
- 12) check if counter 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 counter 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:



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

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

```
counter ++ ;
```

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

```
varName -- ;
```

Thus, the loop above can also be written:

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

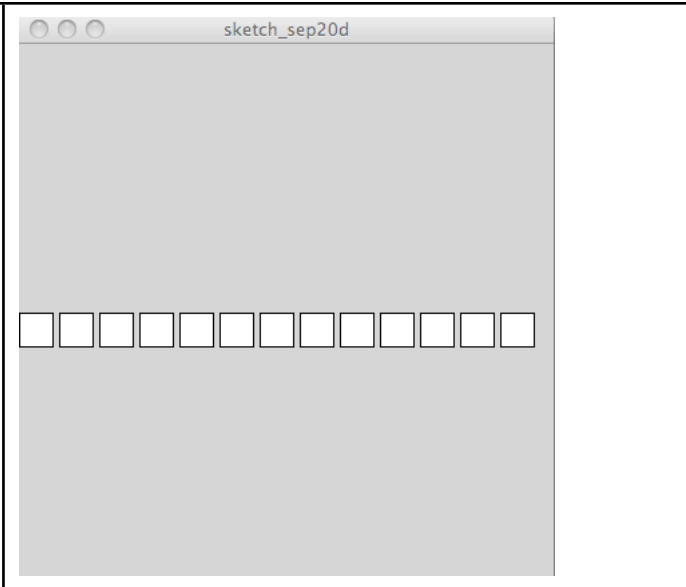
Another common convention is to use the variable name “i” for a loop counter as follows:

```
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:

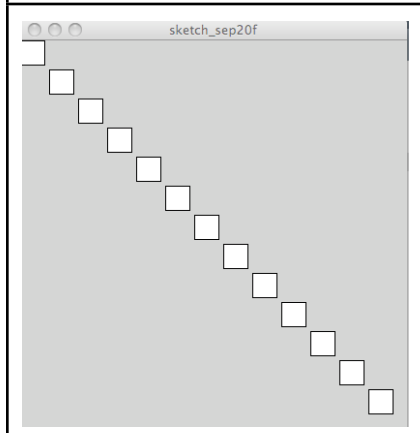
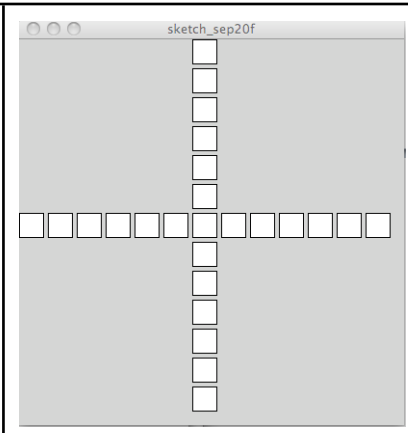
```
for ( int i = 0 ; i < 10 ; i++)  
{  
}
```

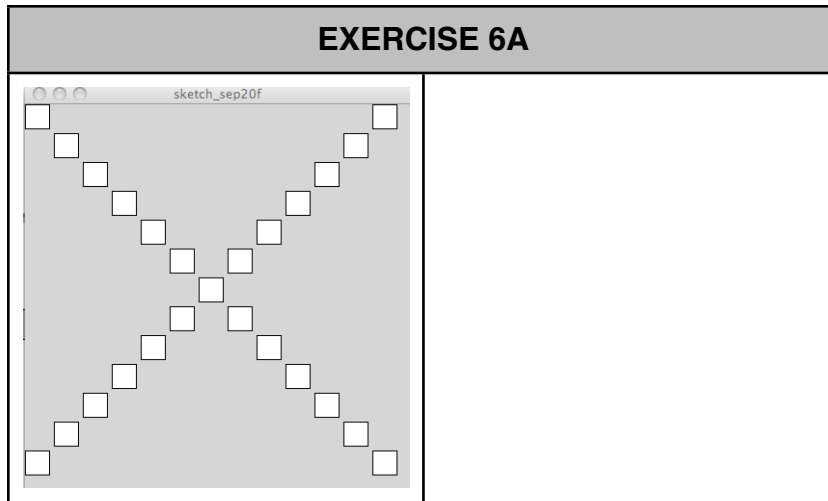
Sometimes the value of the loop counter is used inside the loop as well as being used to determine how many times the loop executes. An example is below. Notice how the x-coordinate of each rectangle is generated from the current value of the loop variable “i”. The rectangle’s x-coordinates are $i*30$,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

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

EXERCISE 6A

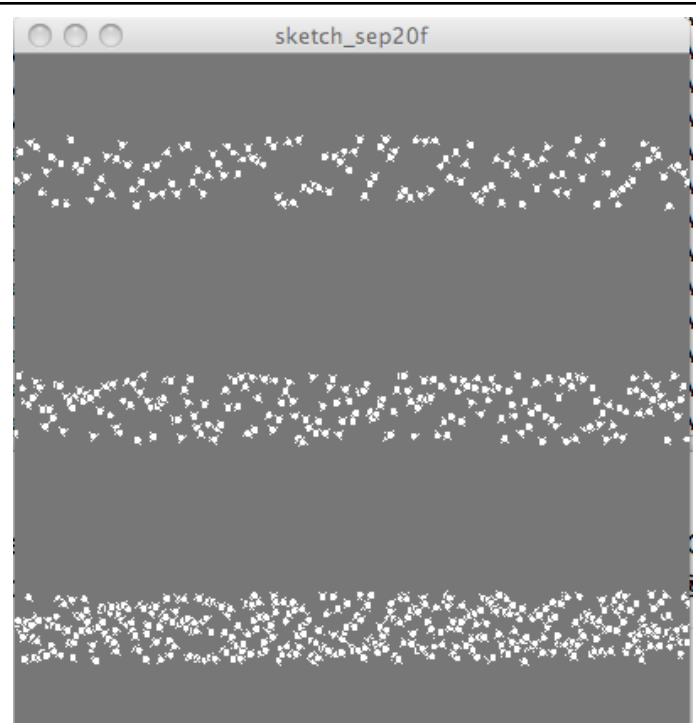
Use one or two for loops to create the following images (or something close to them). Do NOT write a separate rect(x,y,w,h) command for each rectangle, rather, put the command inside a for loop.

	
---	--



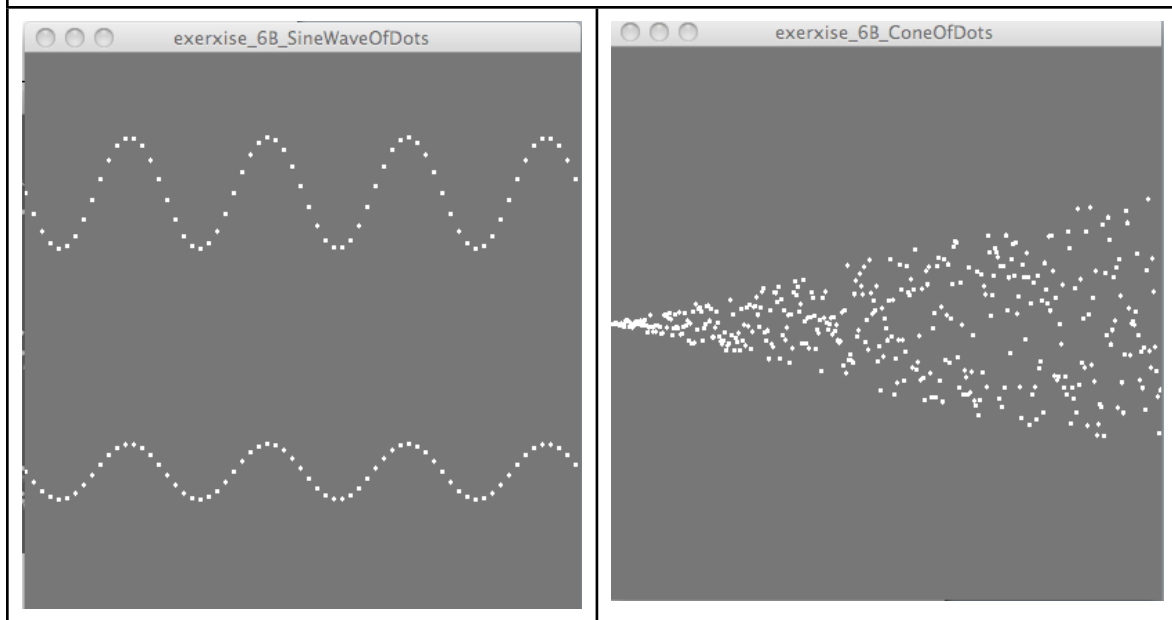
If you make small circles and create lots of of them you can get more interesting pictures such as in the following example. This example, and the exercise following, are motivated by some examples in chapter 6 of the book “Processing. Create Coding and Computational Art.” by Ira Greenberg. Notice that 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 out loop starts it initialized “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:

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

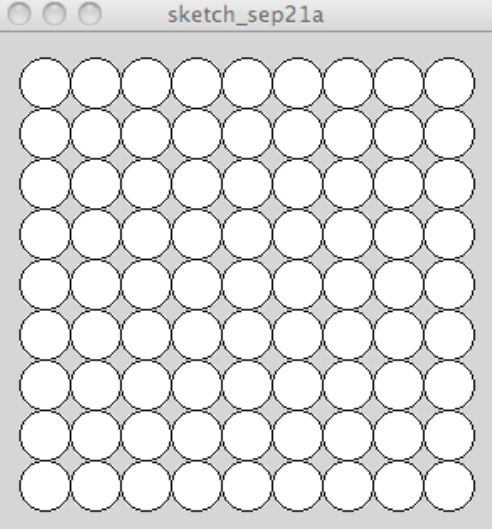
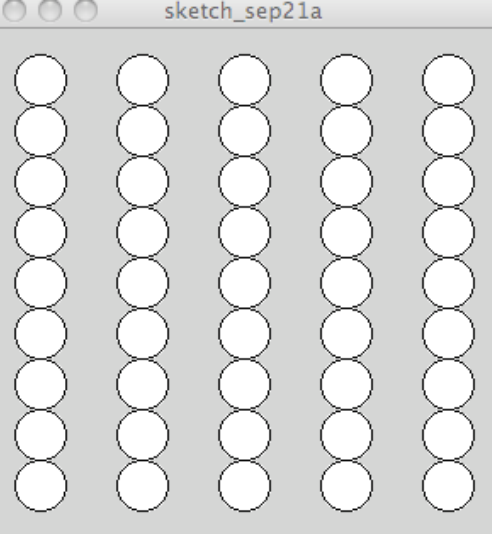
Notice that in the inner loop the initialization condition is:

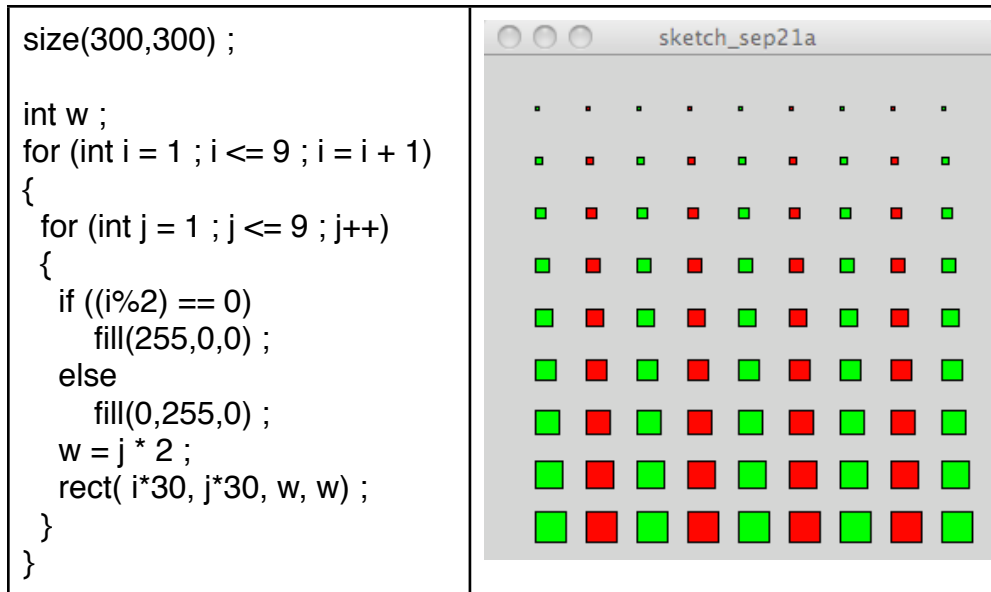
```
int j = i ;
```

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	
<p>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.</p>	
<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:

<pre>size(300,300) ; for (int i = 1 ; i <= 9 ; i++) { for (int j = 1 ; j <= 9 ; j++) { ellipse(i*30, j*30, 30, 30) ; } }</pre>	 A screenshot of a Processing sketch window titled "sketch_sep21a". The window displays a 9x9 grid of 81 white circles, each with a thin black outline. The circles are arranged in a regular grid pattern, filling the entire sketch area.
<pre>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) ; } }</pre>	 A screenshot of a Processing sketch window titled "sketch_sep21a". The window displays a 5x9 grid of 45 white circles, each with a thin black outline. The circles are arranged in five vertical columns, with nine circles in each column. The sketch area is gray.



Note in the example above the statement:

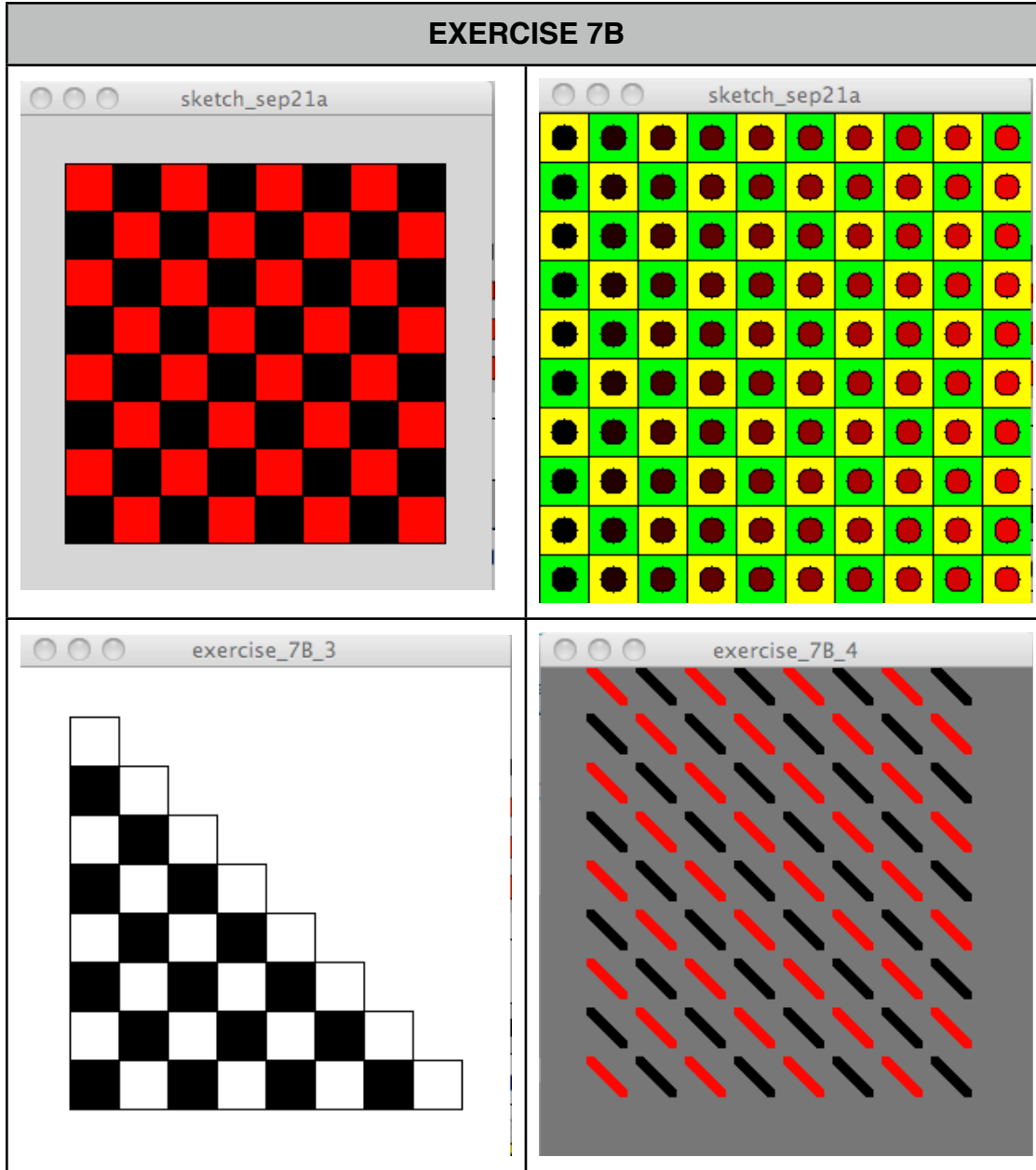
```
if ( (i%2) == 0)
```

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`

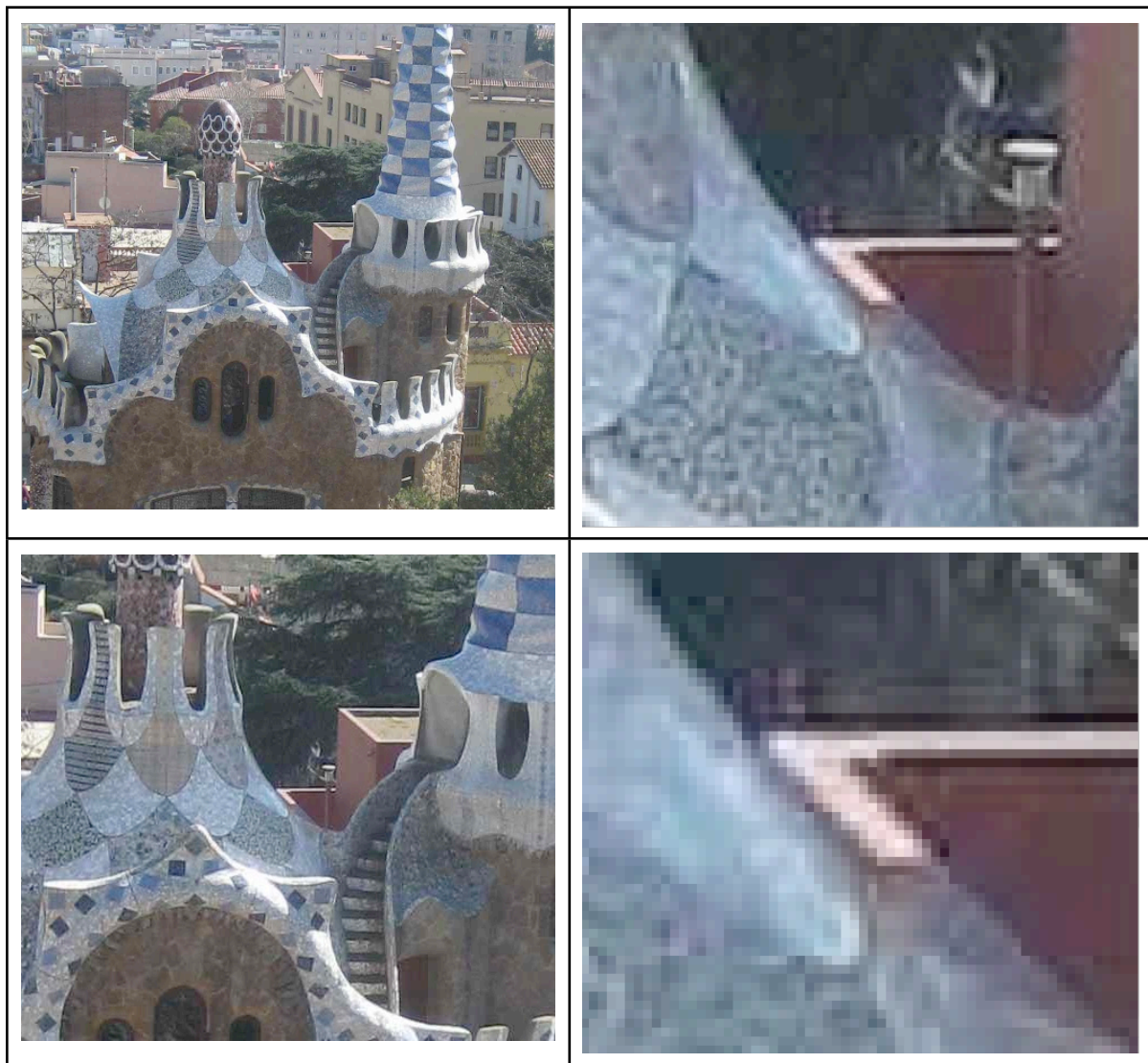


Chapter 8: PImage: Image Manipulation

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 you 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 particularly 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.

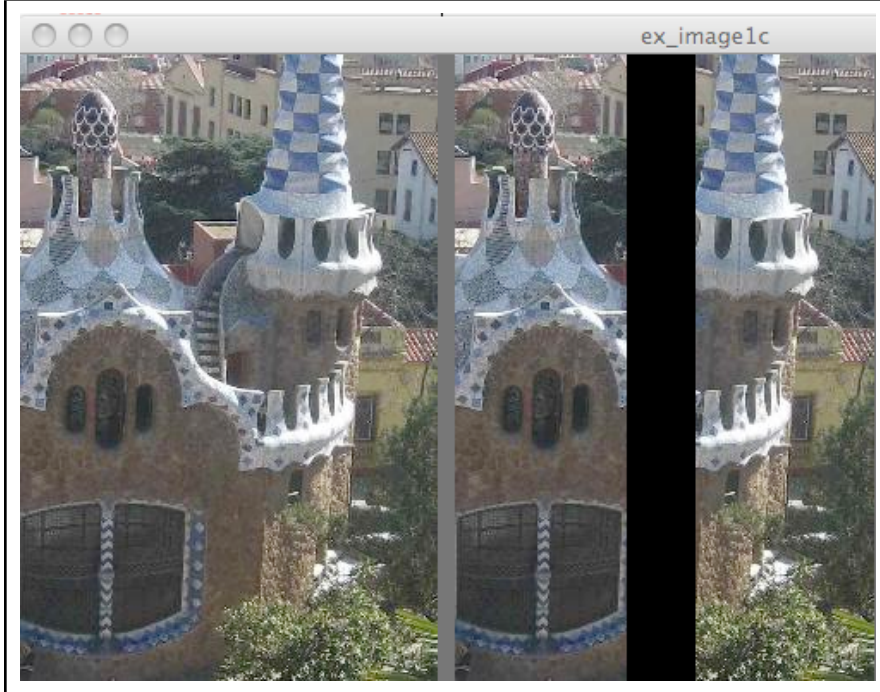
```
PImage theImage ;
int imx = 0 ;
int imy = 0 ;

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) ;
```

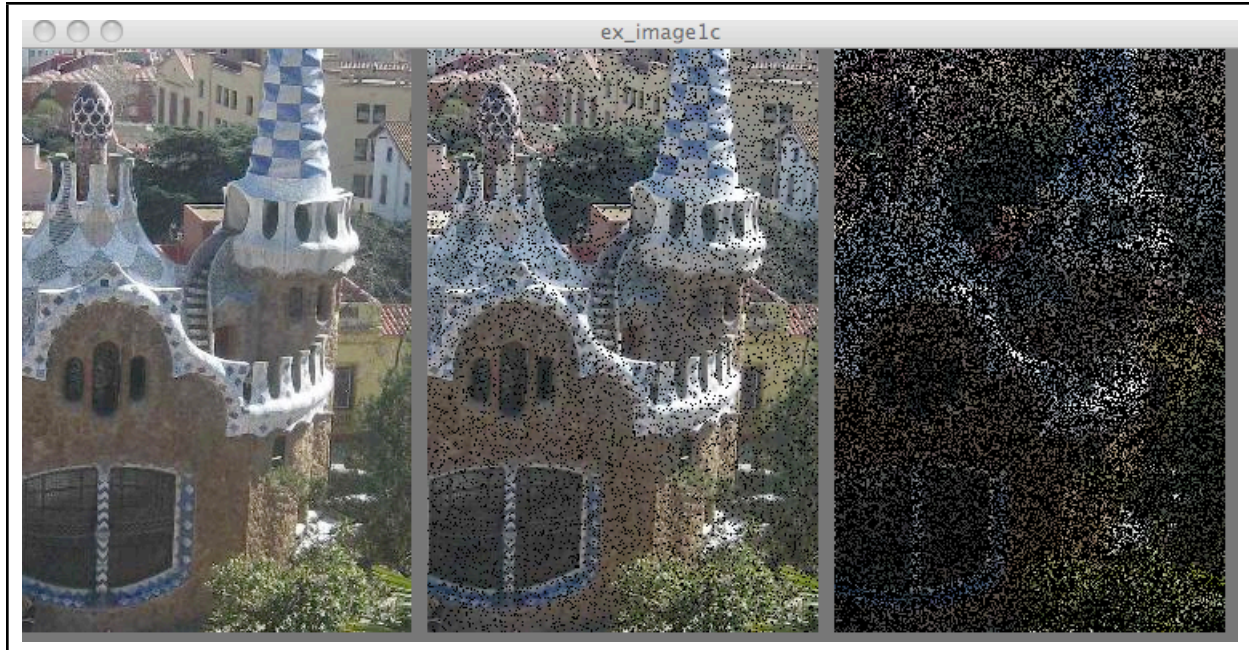


As we can see, there is a big black bar in the right side image. The command “theImage.set(i,j,color1)” will replace the contents of the pixel at location (i,j) with the color specified in color1. We can see that we have made “color1” a variable of type color and set it to black. If we change the nested 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) ;  
background(100) ;  
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 two 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.

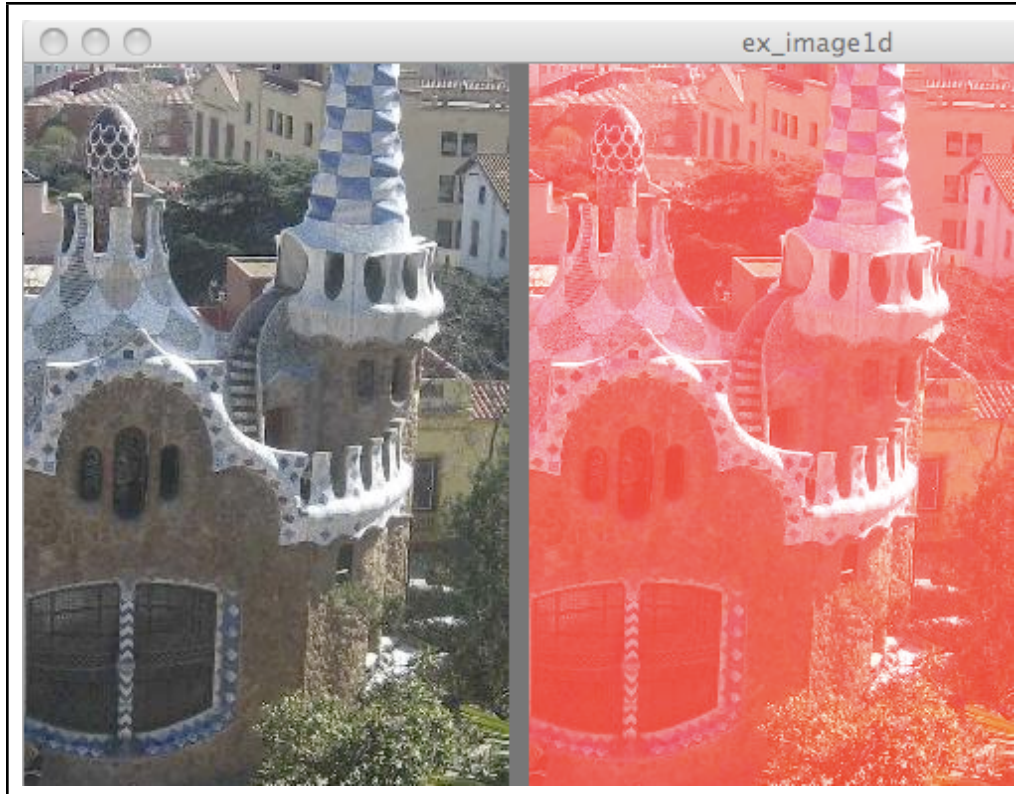
```
size(800,800) ;
background(100) ;
PImage im1, im2, im3 ;
im1 = loadImage("parkguell.jpg") ;
im2 = loadImage("parkguell.jpg") ;

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

// two color variables
color tempColor, newColor ;

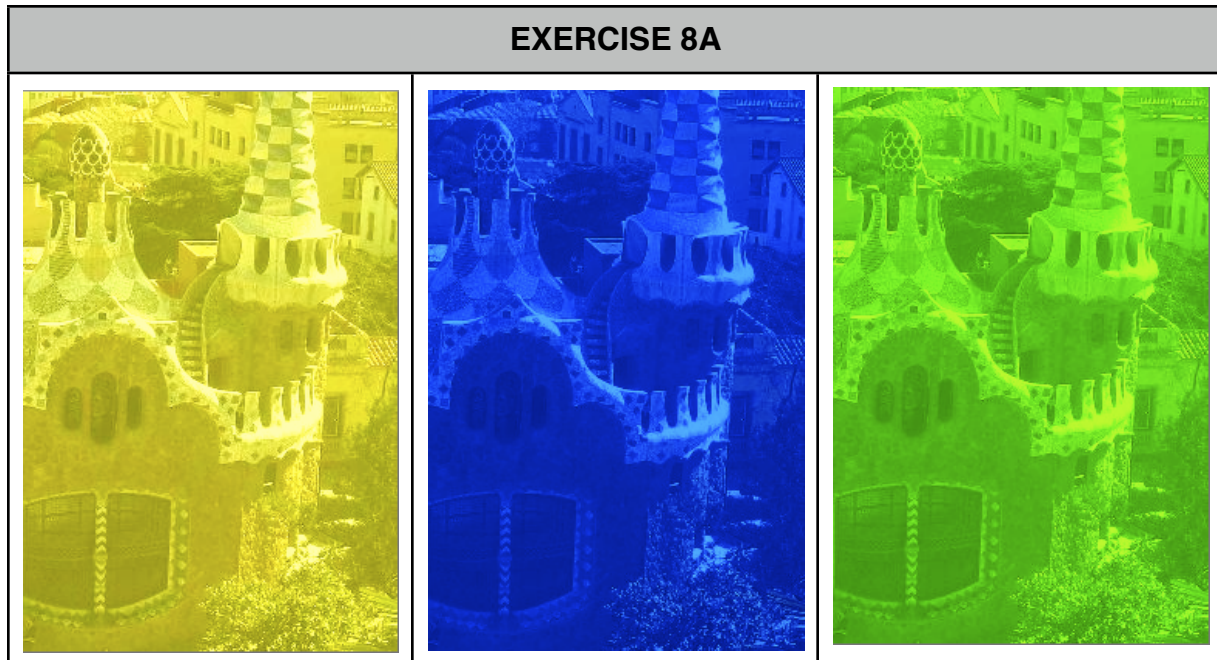
for (int i = 0 ; i < im1.width ; i++)
  for (int j = 0 ; j < im1.height ; j++)
  {
    tempColor = im1.get(i,j) ; // get the color at i,j
    r = red(tempColor) ; // the red value
    g = green(tempColor) ; // the green
    b = blue(tempColor) ; // the blue
    newColor = color( 255, g, b) ; // max out 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!

