Processing Workshop

In this workshop we will be introduced to Processing - a programming language and environment developed "to promote literacy within the visual arts and visual literacy within technology". Processing is used across a variety of communities but has had particular success within the art and design community (and data visualization community) for its strength in generating visual and interactive output. We will will use Processing to:

1. Become familiar with the fundamental concepts of programming visual outputs with code.
2. Build our confidence in using code as a material/tool to turn our ideas into something tangible.
3. Learn about the process of deconstructing tasks into modular steps (don't worry if this doesn't mean anything to you yet!).
4. Explore the ways in which code can create accountability for the visualization process.

We will use the concepts learned in this workshop throughout the rest of the course to learn how to systematically approach data visualization problems and explore other tools and methods of producing visuals with data.

What You'll Need

- Download Processing for your specific operating system.
- The confidence, curiosity and enthusiasm to learn and the drive to teach yourself and the patience to help those around you.

Form & Code

[insert examples here]

Introduction to Processing

"Processing is for writing software to make images, animations, and interactions. The idea is to write a single line of code, and have a circle show up on the screen. Add a few more lines of code, and the circle follows the mouse. Another line of code, and the circle changes color when the mouse is pressed. We call this sketching with code. You write one line, then add another, then another, and so on. The result is a program created one piece at a time. " - Casey Reas & Ben Fry, *Getting Started With Processing*

"Processing relates software concepts to principles of visual form, motion, and interaction. It integrates a programming language, development environment, and teaching methodology into a unified system. Processing was created to teach fundamentals of computer programming within a visual context, to serve as a software sketchbook, and to be used as a production tool. Students, artists, design professionals, and researchers use it for learning, prototyping, and production" - Casey Reas, *Processing, A Programming Handbook for Visual Designers and Artists*

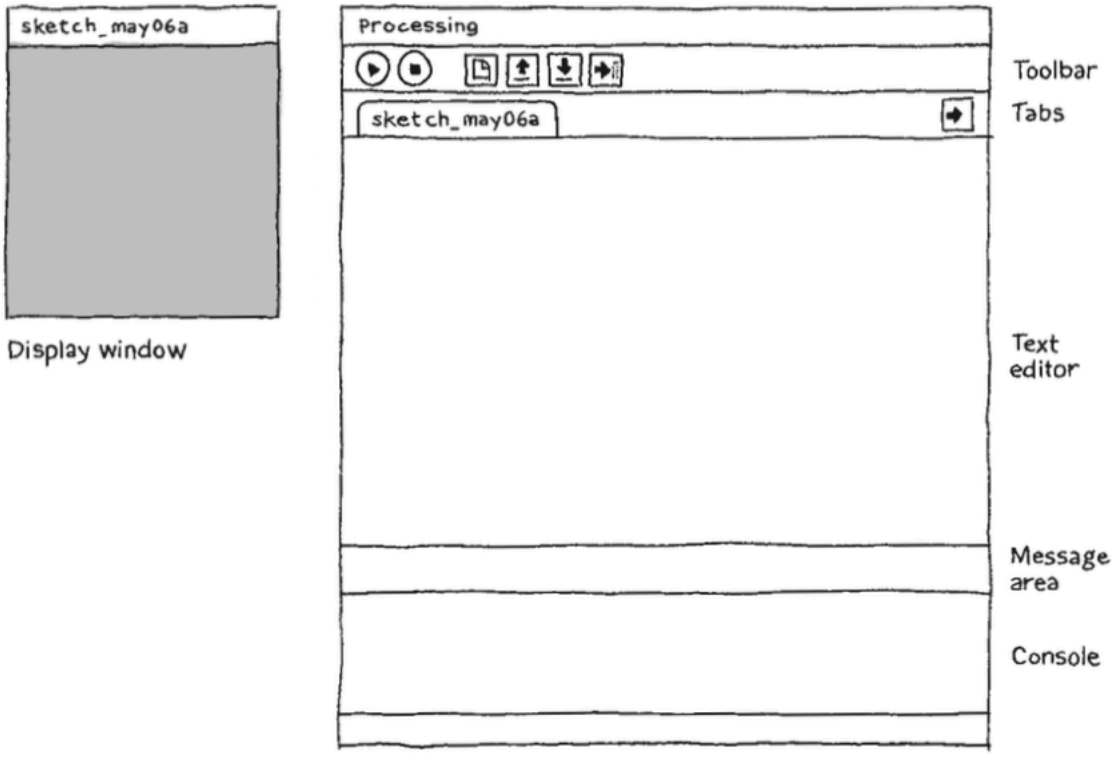
"Software is a unique medium with unique qualities... Every programming language is a distinct material...Sketching is necessary for the development of ideas...Programming is not just for engineers..." - Casey Reas, *Processing, A Programming Handbook for Visual Designers and Artists*

In the last X years, Processing has become one of the most used tools for creating visual outputs (among other things). It has a wide usership in many communities, one being the data visualization community because it allows for flexible and custom exploration of data, it is powerful in dealing with large datasets, and has a relatively friendly syntax and large user community.

The Processing Environment

The Text Editor & Display Window

The Processing Development Environment (IDE) looks like this:



Credits: "Getting started with Processing" - Fry

PDE Elements:

- Text editor: a nice place to put your code
- Play Button: Run your "sketch"
- Stop Button: Stop your "sketch"
- Message Area: One-line messages are printed here - usually error messages.
- Console: The console will print out feedback - usually text that you instruct it to print out while you are building your sketch.
- Display Window: "A computer screen is a grid of light elements called pixels. Each pixel has a position within the grid defined by coordinates. " This is where your sketches will be shown.

The Community

The ethos of Processing is to create an open environment to engage with technology. There's an active community of artists, designers, and scientists from all different domains helping to develop the language & make Processing more accessible. If you continue to work in Processing, you'll definitely find lots of examples and support to help turn your ideas into something tangible.

Getting Started

Make a set of instructions on how to draw a circle, triangle, and square on a piece of paper.

- grab a piece of paper
- grab a pen
- go the the middle of the page and draw a circle
- go to the 1/3 of the width of the page and draw a triangle
- go the the 2/3 of the width of the page and draw a square

Translate your instructions into code

Let's start with an example and code our first program together! You can literally copy and paste the following code bits into your text editor and run your code each time (press the play button) to see how each step changes the program.

First: we need a canvas - something to draw on.

We can use Processing's *size()* function to do this. The size() function takes 2 parameters, *width* and *height*. Below we create a canvas that is 700 pixels wide by 500 pixels high:

```
// size(width, height);
size(700, 500);
```

*Notice we have a semi-colon ";" at the end of the size() function. This tells the computer that this is the end of the statement. Make sure to end each statement with a the semi-colon!

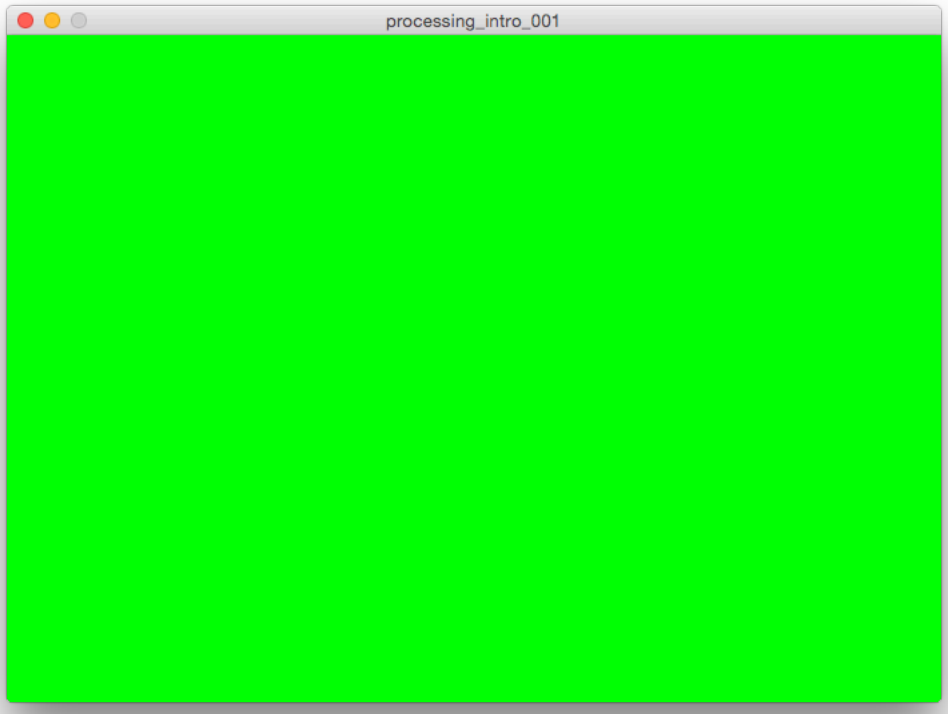
Second: select the color of your canvas

We can set the background color with the function *background()*.

The *background()* function takes colors in RGB, HSV, and even hex codes. Let's try a few here:

```
// a grey background - RGB
background(150);

// a white background - RGB
//background(255, 255, 255);

// a red background - RGB
//background(255, 0, 0);

// a green background - RGB
//background(0, 255, 0);

// a blue background - RGB
//background(0, 0, 255);

// a black background - hex
//background(#000000);
```

```
// a grey background - RGB
//background(150);

// a white background - RGB
//background(255, 255, 255);

// a red background - RGB
//background(255, 0, 0);

// a green background - RGB
background(0, 255, 0);

// a blue background - RGB
//background(0, 0, 255);

// a black background - hex
//background(#000000);
```



*Notice the double forward slashes? these are called **comments** - they are invisible to the computer and do not get evaluated by the computer. These are useful when programming to help you (and others who read your code) keep track of what's happening in your program and allows you to test different functions and statements.

If you want to try some of the other background color options, try **uncommenting** one of the other background() functions by removing the "//" that come before it while commenting out the *background(150);*

*Notice the ";"?

Draw a green circle:

Let's first draw a circle in the middle of the canvas.

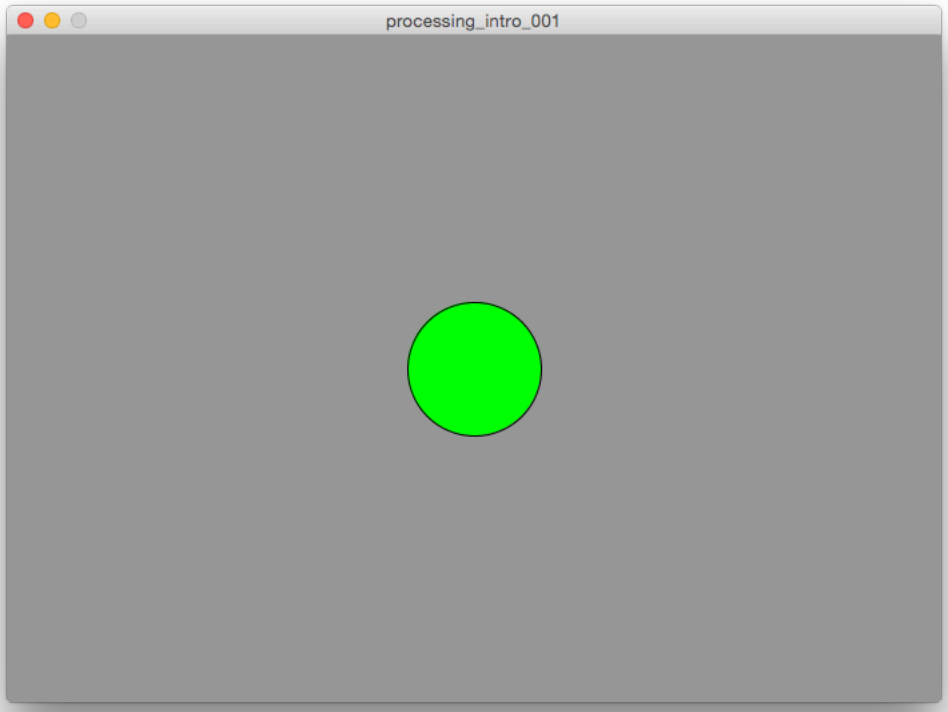
To draw a circle, we can use the *ellipse()* function.

The *ellipse()* function takes 4 arguments - the centroid X, centroid Y, the width radius, and the height radius.

BUT before we can draw the circle, we need to choose the *fill()* color.

Remember we had those RGB color options from the background? Let's try using the same 3 arguements as the green background and see what happens.

```
// a circle
//ellipse(x, y, radiusWidth, radiusHeight);
fill(0, 255, 0);
ellipse(350, 250, 100, 100);
```

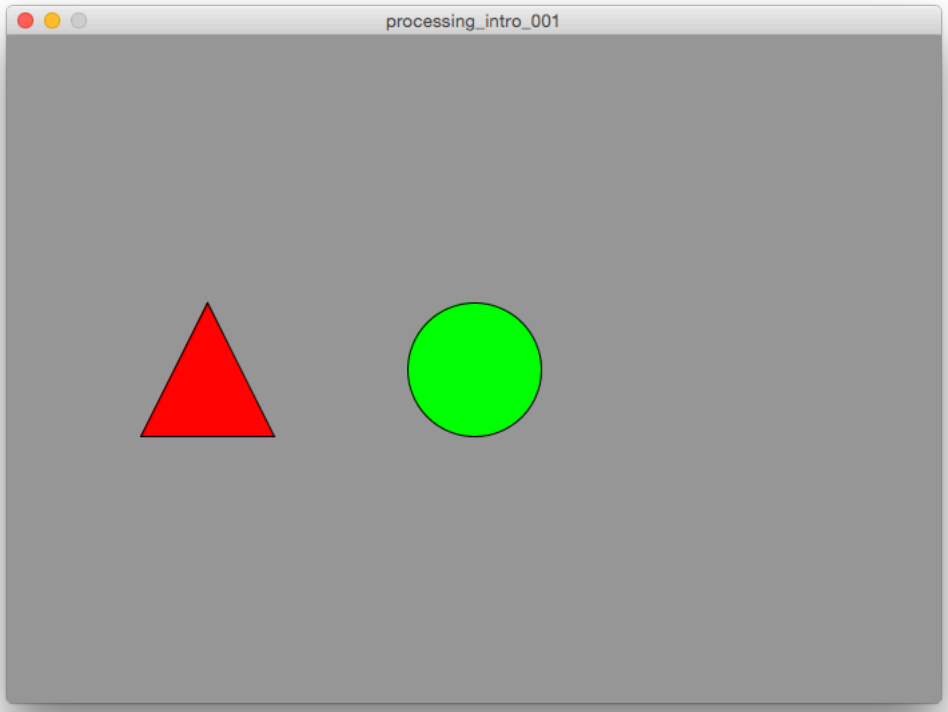


Draw a red triangle:

Let's now draw a triangle in the vertical center and third width of the page.

Using the same color principle from the ellipse, let's create a red triangle using the *fill()* function.

```
// a triangle
// triangle(x1, y1, x2, y2, x3, y3);
fill(255, 0, 0);
triangle(150, 200, 200, 300, 100, 300);
```



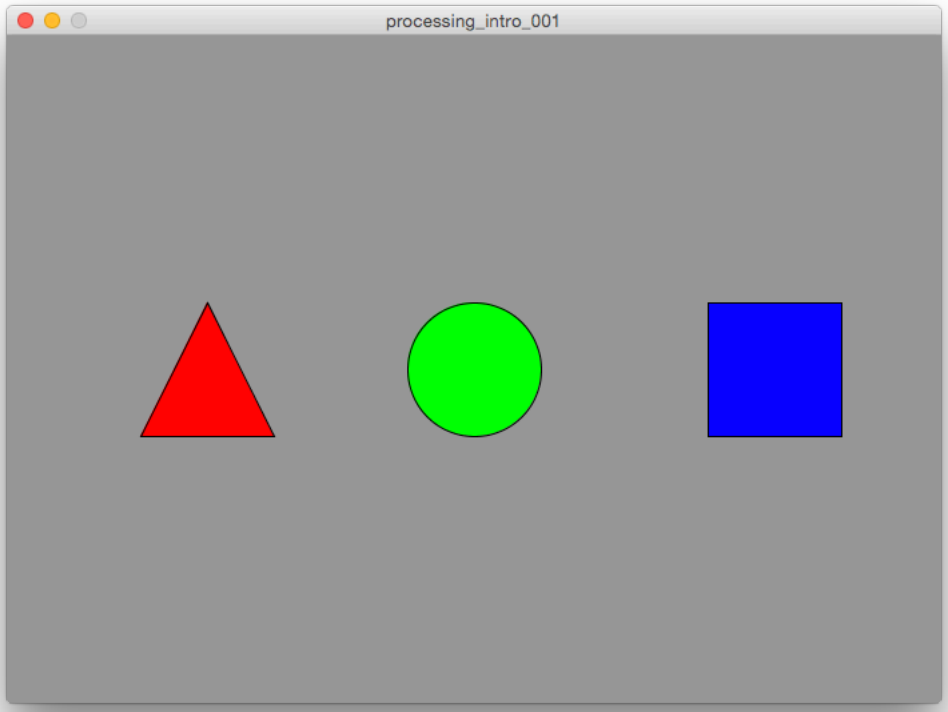
Draw a blue square:

Let's now draw a square on the other side!

Remember to use the *fill()* function to color the square blue.

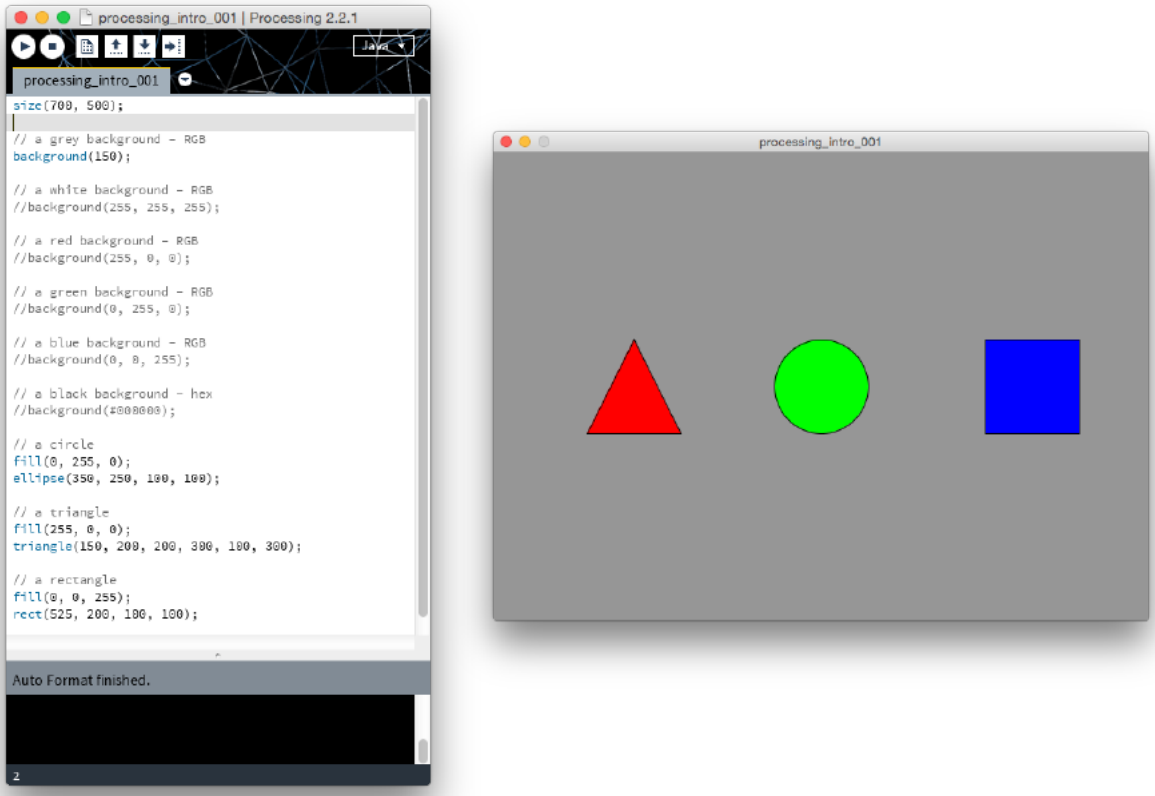
```
// a square
// rect(x, y, width, height);
fill(0, 0, 255);
rect(525, 200, 100, 100);
```

Notice, the x & y of the rectangle starts from the top-left corner. We can change this using rectMode()* and arguements such as "CENTER"... but we can adjust this later ;)



Now your code should looks like this:

```
size(700, 500);

// a grey background - RGB
background(150);

// a white background - RGB
//background(255, 255, 255);

// a red background - RGB
//background(255, 0, 0);

// a green background - RGB
//background(0, 255, 0);

// a blue background - RGB
//background(0, 0, 255);

// a black background - hex
//background(#000000);

// a circle
fill(0, 255, 0);
ellipse(350, 250, 100, 100);

// a triangle
fill(255, 0, 0);
triangle(150, 200, 200, 300, 100, 300);

// a rectangle
fill(0, 0, 255);
rect(525, 200, 100, 100);
```



CONGRATULATIONS, YOU JUST WROTE YOUR FIRST PROGRAM!!!

How do you feel? Empowered? Confused? Excited?

Let's learn more!

Fundamentals

Structure

- **comments**

  We can use two methods of comments:

  For **single line comments**: //

  ```
  // fill(255, 0, 0);
  // ellipse(250, 100, 50, 50);
  ```

  OR

  Block comments for multiple lines of code:

```
  /*
  fill(255, 0, 0);
  ellipse(250, 100, 50, 50);
  */
```

TIP: We can use comments to help us structure our logical steps when writing code. For example:

```
  // first set the canvas size

  // next set the canvas color

  // next draw a circle in the center of the canvas. Since the canvas is 700 pixels wide and 500 pixels tall, the center must
be at (350, 250).

  // next draw a triangle about 1/3 the width of the canvas

  // ...
```

- **functions**

  Functions allow you to draw shapes, set colors, calculate numbers, and to execute many other types of actions. A function's name is usually a lowercase word followed by parentheses. The comma-separated elements between the parentheses are called **parameters**, and they affect the way the function works. Some functions have no parameters and others have many. This program demonstrates the size() and background() functions. cont.

```
  // The size function has two parameters. The first sets the width 1-02 // of the display window and the second sets the
height
  size(200, 200);


  // This version of the background function has one parameter.
  // It sets the gray value for the background of the display window // in the range of 0 (black) to 255 (white)
  background(102);
```

- **expressions & statements:**

  Using an analogy to human languages, a software expression is like a phrase. Software expressions are often combinations of operators such as +, *, and / that operate on the values to their left and right. A software expression can be as basic as a single number or can be a long combination of elements. An expression always has a value, determined by evaluating its contents.

```
  // expression       // value
  10 < 50             true

  // expression       // value
  5*3                 15
```

  A statement - composed of a set of expressions - is like a sentence that gets translated into machine readable code that instructs the computer to do something. For example we see each of these statements (below) tell the computer to do a particular task.

```
  size(200, 200);     // Runs the size() function to set the canvas size
  int x;              // Declares a new variable x as an integer type
  x = 102;            // Assigns the value 102 to the variable x
  background(x);      // Runs the background() function
```

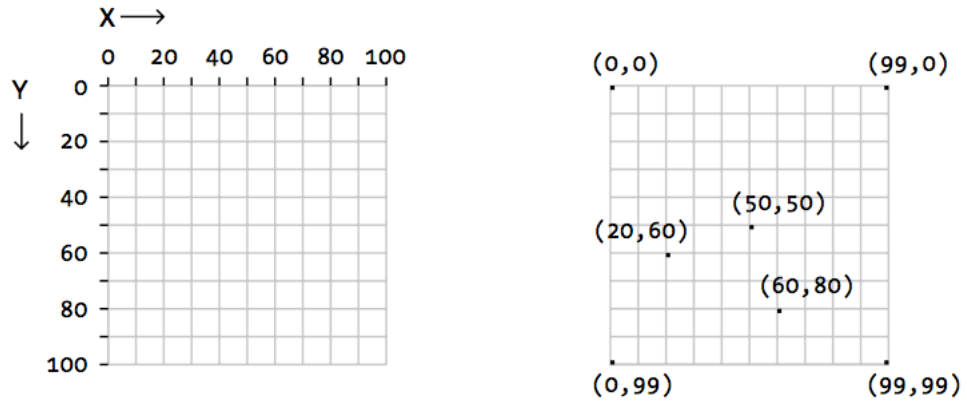
Shape

In the above example, we made our first program consisting of a circle, triangle, and square. Let's dive into how we can draw shapes with Processing.

- **coordinates**

  The coordinate space of a Processing canvas is set using the *size()* function. The parameters for *width* and *height* sets the number of pixels that will be in the *x-coordinate* space and *y-coordinate* space.

  The Processing canvas starts at (0,0) at the top-left corner of the canvas.



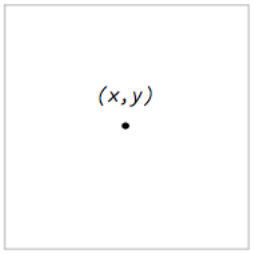
Credits: "A Programming Handbook for Visual Artists and Designers" - Reas & Fry

*for your reference: Processing also comes with a number of other renderers including a 3D renderer, but we won't get into this now - just something to think about!

- **primitives**

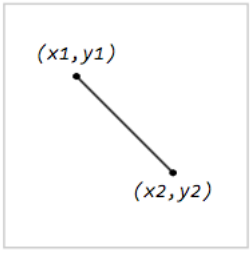
  Processing's primitive shapes are the bread and butter of making visual output with code. As we saw in our first program in which we made a circle, triangle, and square with one function, there are other primitive shapes that we can use. These primitives are listed below:

  point(x,y)



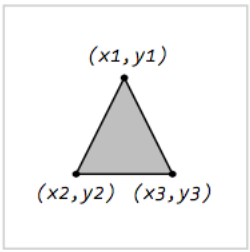

```
point(x, y)
```

  line(x1, y1, x2, y2)



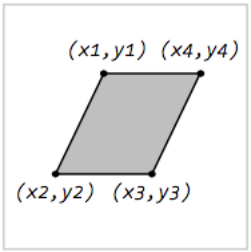

```
line(x1, y1, x2, y2)
```

  triangle(x1, y1, x2, y2, x3, y3)



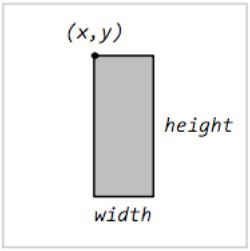

```
triangle(x1, y1, x2, y2, x3, y3)
```

quad(x1, y1, x2, y2, x3, y3, x4, y4)



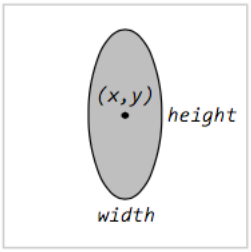

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

rect(x, y, width, height)



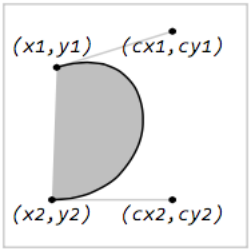

```
rect(x, y, width, height)
```

ellipse(x, y, width, height)




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

bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)




```
bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)
```

- **properties**

  In Processing we can change the properties of the shapes to affect their fill color, stroke color, and drawing mode. Let's go over a few of them here:

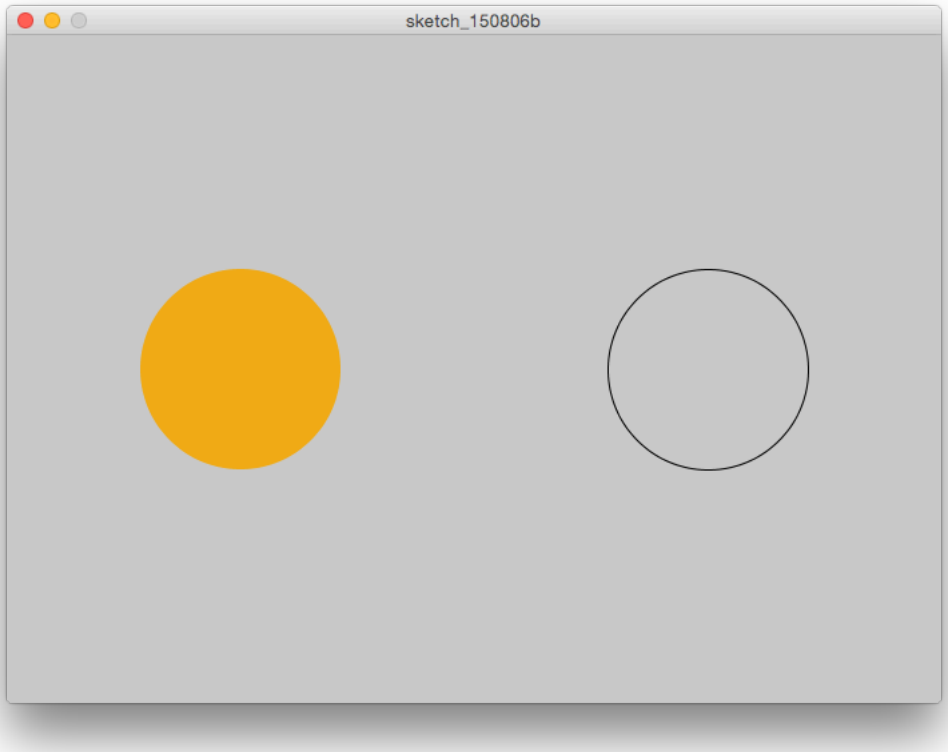
  fill()

```
fill(value1, value2, value3)
fill(value1, value2, value3, alpha)
```

  noFill()

  Use the *noFill()* function if you don't want a fill color

```
size(700, 500);

// with fill
noStroke();
fill(242, 172, 20);
ellipse(width*0.25, height * 0.5, 150, 150);

// no fill
stroke(0);
noFill();
ellipse(width*0.75, height * 0.5, 150, 150);
```



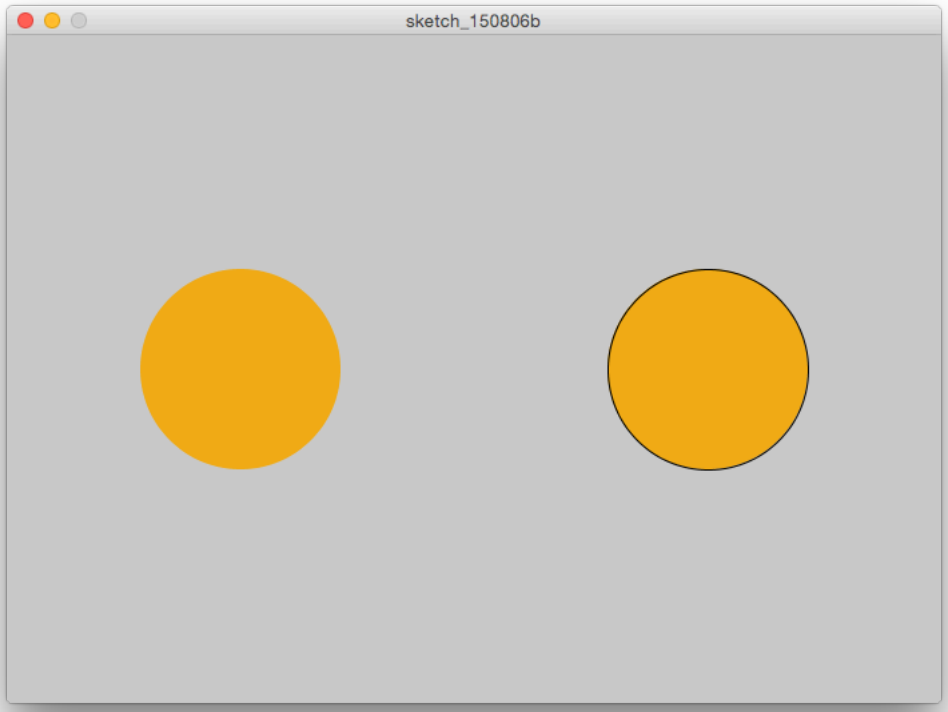
  stroke()

```
stroke(value1, value2, value3)
stroke(value1, value2, value3, alpha)
```

noStroke()

Use the *noStroke()* function if you don't want a stroke color

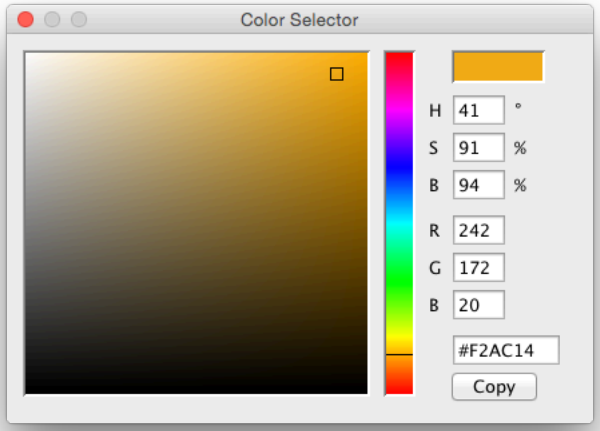
```
size(700, 500);

// no stroke
noStroke();
fill(242, 172, 20);
ellipse(width*0.25, height * 0.5, 150, 150);

// black stroke
stroke(0);
fill(242, 172, 20);
ellipse(width*0.75, height * 0.5, 150, 150);
```



If we use RGB color space:

```
* value1 = red          (between 0 - 255)
* value2 = green        (between 0 - 255)
* value3 = blue         (between 0 - 255)
* alpha = transparency  (between 0 - 100)
```

Processing comes with a handy Color tool to select colors



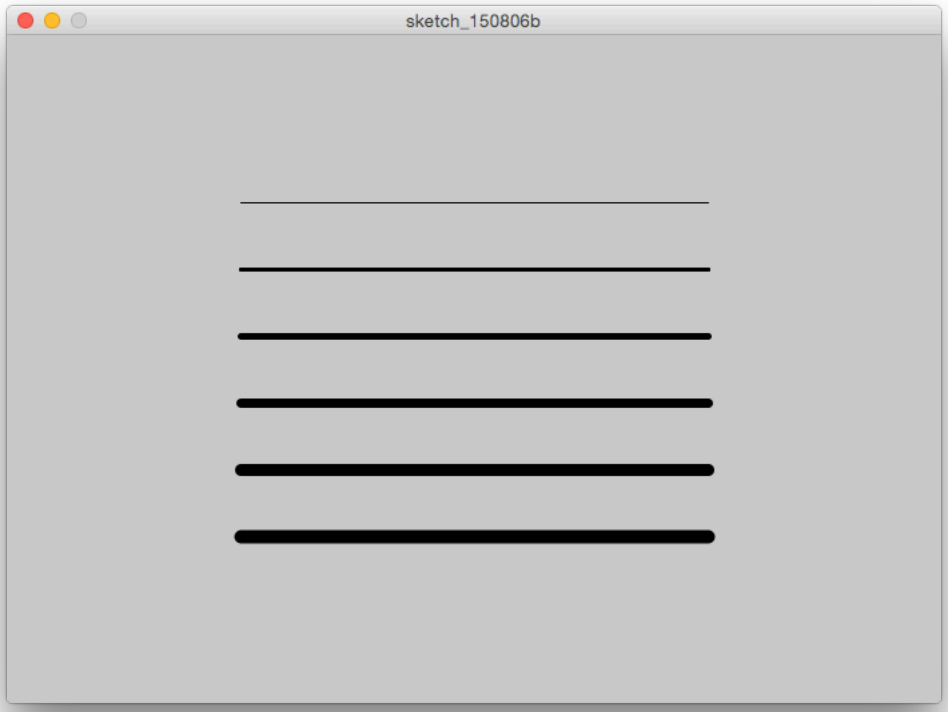
NOTE: if you want to save a color to a variable, you must use the color* type:

```
color pink = color(255, 8, 152);
background(pink);
```

strokeWeight()

You can use the *strokeWeight()* function to change the width of the stroke.

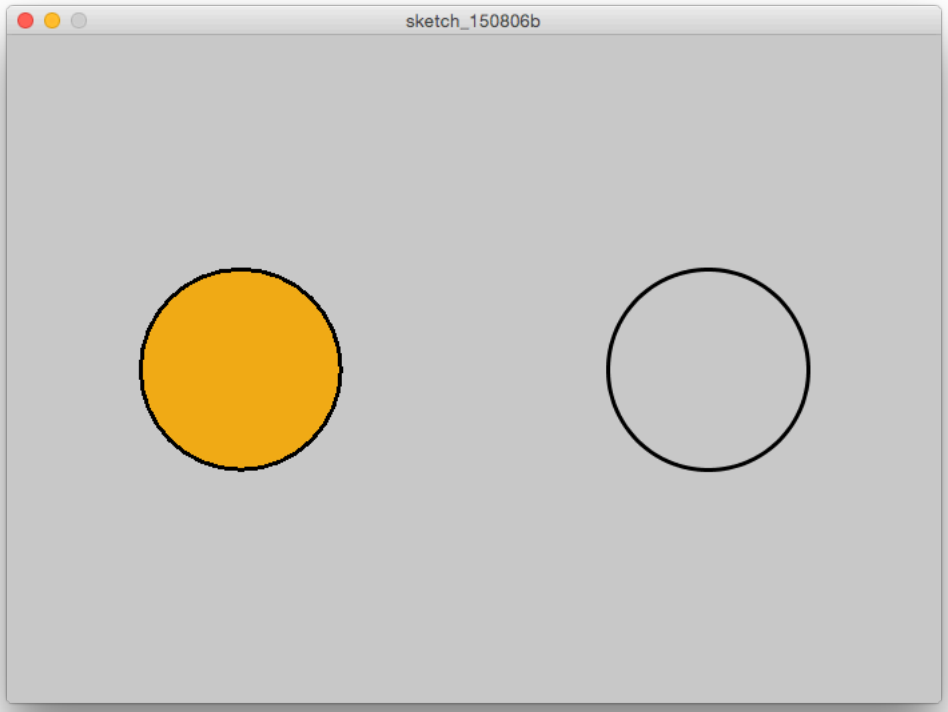
```
size(700, 500);

strokeWeight(1);
line(width*0.25, height * 0.25, width*0.75, height * 0.25);

strokeWeight(3);
line(width*0.25, height * 0.35, width*0.75, height * 0.35);

strokeWeight(5);
line(width*0.25, height * 0.45, width*0.75, height * 0.45);

strokeWeight(7);
line(width*0.25, height * 0.55, width*0.75, height * 0.55);

strokeWeight(9);
line(width*0.25, height * 0.65, width*0.75, height * 0.65);

strokeWeight(10);
line(width*0.25, height * 0.75, width*0.75, height * 0.75);
```



smooth()

Use the *smooth* function to create smooth strokes.

```
size(700, 500);

// not smooth line
noSmooth();
stroke(0);
strokeWeight(3);
fill(242, 172, 20);
ellipse(width*0.25, height * 0.5, 150, 150);

// smooth line
smooth();
stroke(0);
strokeWeight(3);
noFill();
ellipse(width*0.75, height * 0.5, 150, 150);
```

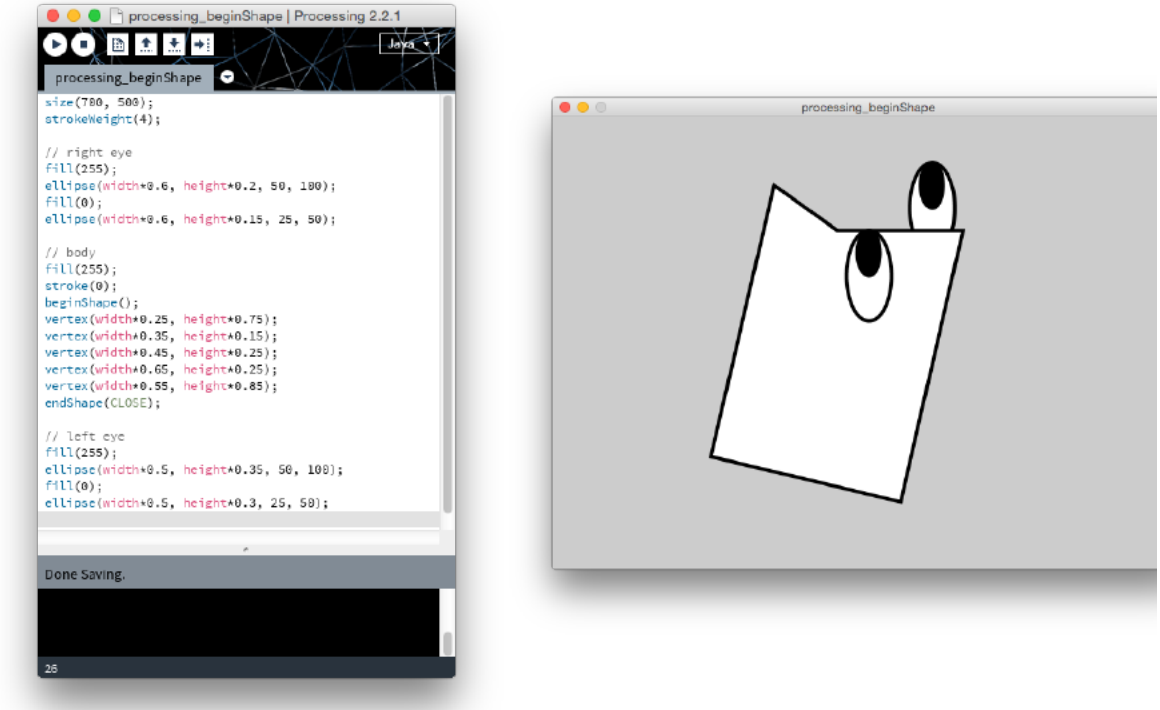


- **custom shapes**

  In Processing we can create our own shapes from a series of vertices. We can do by sandwiching a series of *vertex(x,y)* functions in between the *beginShape()* and *endShape()* functions.

  For example - let's make a funny fish:

```
size(700, 500);
strokeWeight(4);

// right eye
fill(255);
ellipse(width*0.6, height*0.2, 50, 100);
fill(0);
ellipse(width*0.6, height*0.15, 25, 50);

// body
fill(255);
stroke(0);
beginShape();
vertex(width*0.25, height*0.75);
vertex(width*0.35, height*0.15);
vertex(width*0.45, height*0.25);
vertex(width*0.65, height*0.25);
vertex(width*0.55, height*0.85);
endShape(CLOSE);

// left eye
fill(255);
ellipse(width*0.5, height*0.35, 50, 100);
fill(0);
ellipse(width*0.5, height*0.3, 25, 50);
```



There are a number of shape mode parameters that can be passed into the *beginShape()* function in order to have more control over the custom shapes being produced. These include:

beginShape(POINTS)

```
// Draws a point at each vertex
beginShape(POINTS);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

beginShape(LINES)

```
// Draws a line between each pair of vertices 7-08
beginShape(LINES);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

beginShape(TRIANGLES)

```
// Connects each grouping of three vertices as a triangle 7-09
beginShape(TRIANGLES);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```

beginShape(TRIANGLE_STRIP)

```
// Starting with the third vertex, connects each subsequent vertex to the previous two
beginShape(TRIANGLE_STRIP);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```

beginShape(TRIANGLE_FAN)

```
beginShape(TRIANGLE_FAN);
vertex(10, 20);
vertex(75, 30);
vertex(75, 50);
vertex(90, 70);
vertex(10, 20);
endShape();
```

beginShape(QUADS)

```
beginShape(QUADS);
vertex(30, 25);
vertex(85, 30);
vertex(85, 50);
vertex(30, 45);
vertex(30, 60);
vertex(85, 65);
vertex(85, 85);
vertex(30, 80);
endShape();
```

beginShape(QUAD_STRIP)

```
beginShape(QUAD_STRIP);
vertex(30, 25);
vertex(85, 30);
vertex(30, 45);
vertex(85, 50);
vertex(30, 60);
vertex(85, 65);
vertex(30, 80);
vertex(85, 85);
endShape();
```

All of these shape modes are super exciting and interesting, but we won't go into much more detail here. For more info, check out the Processing Documentation and/or the "Programming Handbook for Visual Artists and Designers" by Reas & Fry.

Data & Variables

What is data? Data could be anything from the time of day, the location of a bus stop, the color of your hair, and even the location of your mouse on the screen and the keys you pressed to login to your email. In general, data often consists of measurements of physical characteristics whether it be a digital photo of your dog or the precipitation in Vancouver.

Processing can store, modify, and manipulate many different kinds of data, including numbers, letters, words, colors, images, fonts, and boolean values (true, false).

One of the key features of programming is the ability to store values and/or data to variables. Below are the main data types in Processing - and programming in general - and examples of how to store them to a variable in Processing. NOTE: Processing (the java version) is declarative and thus we must declare what the data type will be when assigning data to a variable. If this sounds crazy, have a look at the examples below :)

- **data type: Numeric**

  int

  Integers are whole numbers such as 12, -120, 8, and 934.

  Processing represents integer data with the "int" data type.

```
// we declare that joey's nerd level is an int equal to 10
int joeyNerdScore = 10;

// we declare that the number of days in a year is equal to 365
int daysInYear = 365;
```

  we can also write the above as:

```
// first declare that joey's nerd score will be a variable
int joeyNerdScore;
// then assign joeyNerdScore is equal to 10
joeyNerdScore = 10;

// first declare that there will be a variable called daysInYear that is an integer
int daysInYear;
// then assign daysInYear equal to 365
daysInYear = 365;
```

  Why would we declare the variable first then assign it later? This becomes important when we start to get into a thing called "variable scope" and learn more about local and global variables in a program. For now, choose whichever one makes most sense to you.

  float

  Floating-point numbers have a decimal point for creating fractions of whole numbers such as 12.8, -120.75, 8.125, and 934.82736.

  Processing represents floating-point data with the "float" data type

```
// we assign joey's enthusiasm for teaching equal to 100
float joeyEnthusiasm = 100.0;

// We assign the class's enthusiasm for learning equal to 2 times that of joey's
float classEnthusiasm = joeyEnthusiasm * 2.0;
```

  Whoa! Did we just take joey's enthusiasm (joeyEnthusiasm) --> multiply it by 2.0 --> then assign the result to the class's enthusiasm (classEnthusiasm)? Damn right we did! This is just an example of how Processing can:

  - store data in a variable -->
  - do something with that variable -->
  - store the result into another variable

  As in the example above we can also declare our variables first then assign the values to the variable later like this:

```
// we first declare the variables we're going to use
float joeyEnthusiasm;
float classEnthusiasm;

// then we assign the values to the variables - pretty cool!
joeyEnthusiasm = 100.0;
classEnthusiasm = joeyEnthusiasm * 2.0;
```

- **data type: Boolean**

  boolean The simplest data element in Processing is a boolean variable. Variables of this type can have only one of two values—true or false. The name boolean refers to the mathematician George Boole (b. 1815), the inventor of Boolean algebra—the foundation for how digital computers work. A boolean variable is often used to make decisions about which lines of code are run and which are ignored.

```
// it is false that it is July
boolean july = false;

// it is true that we're in vancovuer
boolean vancouver = true;
```

These data types are super exciting and great, but for now, just know that they exist... we can revist them later :)

- **data type: String**

  String

- **data type: Color**

  color

Math

Math can be an important aspect of programming, but it's not necessary to be good at math to understand or enjoy programming.

- **arithmetic & functions**

```
+ (add)
- (subtract)
* (multiply)
/ (divide)
% (modulus)
() (parentheses)
++ (increment)
-- (decrement)
+= (add assign)
-= (subtract assign)
*= (multiply assign)
/= (divide assign)
- (negation)
ceil()
floor()
round()
min()
max()
```

(Add) +

```
int x;
int y;
int z;

x = 10;
y = 5;
z = x + y;

println(z);  // z will equal 15

// let's take z and set the background color
background(z);

// then create an ellipse and set the fill(z, z+100, z+20)
fill(z, z+100, z+20);
ellipse(width/2, height/2, 50, 50);
```

(subtract) -

```
int x;
int y;
int z;

x = 10;
y = 5;
z = x - y;

println(z);  // z will equal 5

// let's make a yellow ellipse with a strokeWeight(z)
strokeWeight(z);
fill(#FFF700);
ellipse(width/2, height/2, 50, 50);
```

(multiply) *

```
int x;
int y;
int z;

x = 10;
y = 5;
z = x * y;

println(z);  // z will equal 50

// let's make a rectangle that's z pixels wide and z*2 pixels high
size(400, 400);
rect(width/2, height/2, z, z*2);
```

(divide) /

```
int x;
int y;
int z;

x = 10;
y = 5;
z = x / y;

println(z);  // z will equal 2

// let's make 5 ellipses each a size "z" smaller than the preceding one
size(400, 400);

fill(z+240, z+15, z*4);
ellipse(width/2, height/2, 100/z, 100/z);

fill(z, z*60, z+60);
ellipse(width/2, height*0.25, 50/z, 50/z);

fill(z, z*35, z+200);
ellipse(width/2, height*0.75, 50/z, 50/z);
```

(modulo) %

The modulo returns the remainder of the division between two numbers

```
int x;
int y;
int z;
int j;
int i;

x = 10;
y = 5;
j = 3;

z = x % y;
i = x % j;

println(z); // z will equal 0 since the remainder of 10/5 is 0
println(i); // i will return 1 since the remainder of 10/3 is 1

// Here's an if/else statement - we're going to learn about these in section: Control!
if (z == 1){
    fill(#00F9FF); // if z == 1, then color the ellipse turqoise
} else {
    fill(#FE00FF); // if z does not equal 1, then color the ellipse purple
}

ellipse(width/2, height/2, 50, 50); // we'll see a purple ellipse!
```

(parentheses) ()

The parentheses can be used to set up your mathematical order of operations (remember PEMDAS?). Be sure to keep the order of operations in mind when writing your statements.

```
int x = 3 + 4 * 5; // Assign 23 to x
int y = (3 + 4) * 5; // Assign 35 to y

println(x);
println(y);
```

(increment)++

The increment is a shortcut for adding 1 value to an existing variable.

```
// incrementing the long way;
int x = 0;
x = x + 1;
println(x); // x will print as 1;

// incrementing the short way:
int y = 0;
y++;
println(y); // y will print as 1;
```

(decrement) --

```
// decrementing the long way;
int x = 10;
x = x - 1;
println(x); // x will print as 9;

// incrementing the short way:
int y = 10;
y--;
println(y); // y will print as 9;
```

+= (add assign), -= (subtract assign), *= (multiply assign), /= (divide assign) The add assign, subtract assign, multiply assign, and divide assign allow you to add, subtract, multiply, or divide a variable by a specified value:

```
// add assign
float a = 10.0;
a += 10;
println(a);     // a will print as 20.0;

// subtract assign
float b = 35.6;
b -= 0.6;
println(b);     // b will print as 35.0

// multiply assign
int c = 100;
c *= 3;
println(c);     // c will print as 300

// divide assign
int d = 36;
d /= 4;
println(d);     // d will print as 8
```

ceil()

The ceil() function rounds whatever parameter **up** to the nearest whole number

```
// round 32.4 up to 33.0
float y = ceil(32.4);
println(y);

// round 67.7 up to 68
float x = ceil(67.7);
println(x);
```

floor()

The floor() function rounds whatever parameter **down** to the nearest whole number

```
// round 32.4 down to 32.0
float y = floor(32.4);
println(y);

// round 67.7 up to 67.0
float x = floor(67.7);
println(x);
```

round()

The round() function returns the closest value.

```
// round 32.4 to 32.0
float y = round(32.4);
println(y);

// round 67.7 up to 68.0
float x = round(67.7);
println(x);
```

min(), **max()**

The min() function returns the minimum value from a sequence of at least 2, but maximum 3 parameter. You can also apply the min() function on an array to return the minimum value in an array of numbers.

```
int u = min(5, 9); // Assign 5 to u

int v = min(-4, -12, -9); // Assign -12 to v

float w = min(12.3, 230.24); // Assign 12.3 to w
```

The max() function returns the maximum value from a sequence of at least 2, but maximum 3 parameters. You can also apply the max() function on an

array to return the maximum value in an array of numbers.

```
int x = max(5, 9); // Assign 9 to x

int y = max(-4, -12, -9); // Assign -4 to y

float z = max(12.3, 230.24); // Assign 230.24 to z
```

Control

The programs we've seen so far run each line of code in sequence. They run the first line, then the second, then the third, etc. The program stops when the last line is run. It's often beneficial to change this order—sometimes skipping lines or repeating lines many times to perform a repetitive action. Although the lines of code that comprise a program are always positioned in an order from top to bottom on the page, this doesn't necessarily define the order in which each line is run. This order is called the flow of the program. Flow can be changed by adding elements of code called control structures.

- **conditionals & decision making**

  Relational Expressions A relational expression is made up of two values that are compared with a relational operator. In Processing, two values can be compared with relational operators as follows:

```
Expression          Evaluation

3> 5                false

3< 5                true

5< 3                false

5> 3                true
```

  In Processing, we have these relational operators at our fingertips in order to evaluate truthy or falsy values:

```
Operator            Meaning
>                   greater than

<                   less than

>=                  greater than or equal to <= less than or equal to

==                  equivalent to

!=                  not equivalent to
```

  A few examples of the above:

```
// 3 is less than 5 is TRUE
println(3 < 5); // Prints "true"

// 3 is less than or equal to 5 is TRUE
println(3 <= 5); // Prints "true"

// 3 is greater than 5 is FALSE
println(3 > 5); // Prints "false"

// 3 is greater than or equal to 5 is FALSE
println(3 >= 5); // Prints "false"

// 3 is equal 5 is FALSE
println(3 == 5); // Prints "false"

// 3 does not equal 5 is TRUE
println(3 != 5); // Prints "true"
```

  Conditionals: if/else/else if statements

  Conditionals allow a program to make decisions about which lines of code run and which do not. They let actions take place only when a specific condition is met. Conditionals allow a program to behave differently depending on the values of their variables.

  if/else statements are ways to control the behavior of your program. This allows us to make decisions about what happens in our code. if/else

statements might allow us to filter out data, change the size of an ellipse or rectangle depending on the value of a variable, or react to a mouse or key input (oooh fancy!).

A simple if statement

Below we have a basic structure of an if statement. It basically says: "if a condition is met, execute the statements within the brackets."

```
// psuedo code - this wont run in Processing
if (test) {
    statements
}
```

Let's try an example:

```
// if x is greater than 20, print "hello", and draw a triangle

int x = 20;

if (x > 20){
    println("hello");
    triangle(width/2, height*0.25, width*0.75, height*0.75, width*0.25, height*0.75);
}
```

But wait! nothing happened! we don't see a hello or a triangle. Well, well, well, turns out the computer understood our directions. Because x is not greater than 20, our statements within the if statement were not run.

Let's change x to 21 and see what happens:

```
// if x is greater than 20, print "hello", and draw a triangle

int x = 21;

if (x > 20){
    println("hello");
    triangle(width/2, height*0.25, width*0.75, height*0.75, width*0.25, height*0.75);
}
```

BOOM! Our if statement worked! Pretty sweet. But what about adding some more control.

A simple if/else statement

The if/else combo allows you to say "if a condition is met, then execute the statements within the brackets of the if statement, HOWEVER if those conditions are NOT met, then run the statements within the brackets of the else statement."

```
// psuedo code - this wont run in Processing
if (test) {
    statements
} else {
    statements
}
```

let's take the example above and add an else statement:

```
// if x is greater than 20, print "I'm a triangle", and draw an orange triangle, HOWEVER if it is less than 20, then draw a
purple ellipse and print "i'm an ellipse".

int x = 20;

noStroke();

if (x > 20){
    println("I'm a triangle");
    fill(#FFA600);
    triangle(width/2, height*0.25, width*0.75, height*0.75, width*0.25, height*0.75);
} else{
    println("I'm an ellipse");
    fill(#9400FF);
    ellipse(width/2, height/2, 50, 50);
}
```

Yeah! now we see our purple ellipse since x is not greater than 20.

Let's change the x value to 21 and see what happens:

```
     // if x is greater than 20, print "I'm a triangle", and draw an orange triangle, HOWEVER if it is less than 20, then draw
a purple ellipse and print "i'm an ellipse".

  int x = 21;

  noStroke();

  if (x > 20){
      println("I'm a triangle");
      fill(#FFA600);
      triangle(width/2, height*0.25, width*0.75, height*0.75, width*0.25, height*0.75);
  } else{
      println("I'm an ellipse");
      fill(#9400FF);
      ellipse(width/2, height/2, 50, 50);
  }
```

Boom! An orange triangle. We're totally telling the computer who's boss!

if/else if/ else

We've not made an if statement, and if/else statement, and now we want to add 1 more layer of control to our decision making. Here we introduce the "else if" conditional. The else if statement allows us to say "if a condition is met, then execute the statements within the brackets of the if statement, HOWEVER if those conditions are NOT met, then run the statements within the brackets of the next else if statement, BUT if those conditions are not met, then go on to the next else if statement, and so on an so forth."

```
  // psuedo code - this wont run in Processing
  if (test) {
      statements
  } else if (test) {
      statements
  } else if {
      statements
  }

  ... (as many else if statements as you'd like)...

  else{
      statements
  }
```

Let's tweek the example from above. In this case, if x is equal to 20, still draw the triangle, but make it turquoise:

```
          // if x is greater than 20, print "I'm a triangle", and draw an orange triangle, BUT if x is equal to 20, then draw a
turqoise triangle,  HOWEVER if x is less than 20, then draw a purple ellipse and print "i'm an ellipse".

  int x = 20;

  noStroke();

  if (x > 20){
      println("I'm an orange triangle");
      fill(#FFA600);
      triangle(width/2, height*0.25, width*0.75, height*0.75, width*0.25, height*0.75);
  } else if (x == 20) {
      println("I'm a turquise triangle");
      fill(#00FFCA);
      ellipse(width/2, height/2, 50, 50);
  } else{
      println("I'm an purple ellipse");
      fill(#9400FF);
      ellipse(width/2, height/2, 50, 50);
  }
```

Logical Operators

We can have multiple "tests" conditions in an if or else if statement. Using the logical operators to combine test conditions in an if or else if statement

makes it possible to do this.

```
Operator            Meaning

&&                  AND

||                  OR

!                   NOT



Expression          Evaluation

true && true        true
true && false       false
false && false      false
true || true        true
true || false       true
false || false      false
!true               false
!false              true
```

Let's look at one example:

```
int a = 10;
int b = 20;


// The expression "a > 5" must be true OR "b < 30"
// must be true. Because they are both true, the code // in the block will run.
if ((a > 5) || (b < 30)) {
  line(20, 50, 80, 50);
}
// The expression "a > 15" is false, but "b < 30"
// is true. Because the OR operator requires only one part // to be true in the entire expression, the code in the
// block will run.
if ((a > 15) || (b < 30)) {
  ellipse(50, 50, 36, 36);
}
```

And 1 example using boolean values:

```
boolean b = true;

// because b is true, the line will draw
if (b == true){
    line(20, 50, 80 ,50);
}

if (!b == true){
    ellipse(50, 50, 36, 36);
}
```

- **repitition**

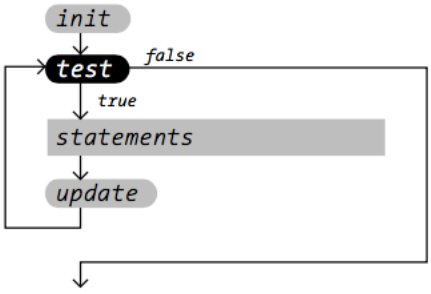
  Computers are excellent at executing repetitive tasks accurately and quickly. Modern computers are also logic machines. Building on the work of the logicians Leibniz and Boole, modern computers use logical operations such as AND, OR, and NOT to determine which lines of code are run and which are not.

  The for loop

  The for loop is probably the coolest thing since ever. At the most basic level for loops allow your to repeat a statement or statements over and over again until a condition is met. The general structure if described in the image below.

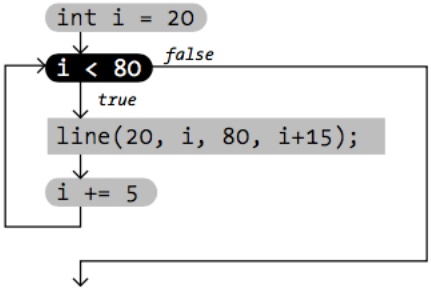
General case `for` structure

```
for (init; test; update) {
  statements
}
```



A specific `for` structure

```
for (int i = 20; i < 80; i += 5) {
  line(20, i, 80, i+15);
}
```



Credits: "A Programming Handbook for Visual Artists and Designers" - Reas & Fry

Let's look closer at the specific "for" structure:

```
for (int i = 20; i < 80; i += 5) {
    line(20, i, 80, i+15);
}
```

If you were to read this out loud it would sound like: "for the integer i equals 20, as long as i is less than 80, increment i by 5, drawing a line each time with new y coordinates based on i."

Why are for loops so awesome? Let's look at an example of how a loop can be used to take a program which takes 14 lines of code to make 14 lines...

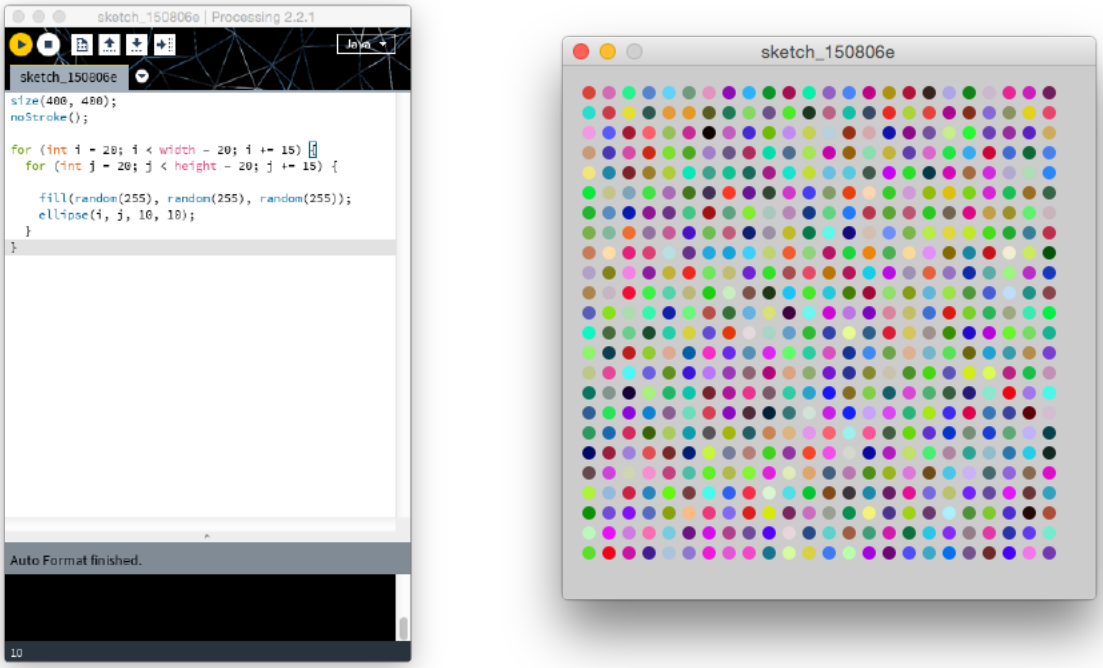
```
// original hard-coded program
size(200, 200);
line(20, 20, 20, 180);
line(30, 20, 30, 180);
line(40, 20, 40, 180);
line(50, 20, 50, 180);
line(60, 20, 60, 180);
line(70, 20, 70, 180);
line(80, 20, 80, 180);
line(90, 20, 90, 180);
line(100, 20, 100, 180);
line(110, 20, 110, 180);
line(120, 20, 120, 180);
line(130, 20, 130, 180);
line(140, 20, 140, 180);
```

... and make the program 4 lines of code:

```
size(200, 200);
for (int i = 20; i < 150; i += 10) {
  line(i, 20, i, 180);
}
```

Let's get a little crazy and make a nested loop. A nested loop? You can do that?! Sure!

```
size(400, 400);
noStroke();

for (int i = 20; i < width - 20; i += 15) {
  for (int j = 20; j < height - 20; j += 15) {
        fill(random(255), random(255), random(255));
        ellipse(i, j, 10, 10);
    }
}
```



How does a nested loop work? First, the outer loop starts. Then the inner loop runs until the condition is met at which point the outer loop increments, repeating the inner loop again until the condition is met at whihc point the outer loop increments, and over and over until the outer loop condition is met. This means that columns of ellipses are drawn from left to right with each complete loop.

*NOTE: the random() function generates a random number from 0 to the value of the input parameter. In this case, for each iteration in the loop, a random number is generated between 0 and 255 for each parameter in the fill() function.

WOW, we just learned a bunch! Do your brains hurt? Mine too. Let's take some time to let that settle and explore all this new knowledge!

Let's make our first program ("sketch")

Using the shapes and properties we just learned, take 15 minutes to make a simple drawing. Use your i-m-a-g-i-n-a-t-i-o-n.

(At the end, we'll all go around to see what we've all made)

Response

- **setup & draw - the processing logic/structure**
- **mouse & keyboard**

Web Searching

"Programming is just about how good you are at google searching."