

Processing - Generative Art / Design Tutorial

-- level 1 --

The following tutorial was original compiled by Manuel Kretzer. It is a combination of text excerpts and illustrations from “[Learning Processing, Second Edition: A Beginner’s Guide to Programming Images, Animation, and Interaction](#)” by Daniel Shiffman (2015), “[Generative Art](#)” by Matt Pearson (2011), and “[Make: Getting Started with Processing](#)” by Casey Reas and Ben Fry (2010) as well as online sources such as tutorials provided on the Processing website (<https://processing.org/tutorials/>).

All sketches in this tutorial can be downloaded [here](#).

Contents

level 1	level 2
1 Introduction	16 Random
2 Sketching	17 Examples
3 Libraries	17.1 Circle Grid
4 Your First Program	17.2 Square Grid
5 The Coordinate System	17.3 Birds Nest
6 Functions	17.4 Network
7 Basic Shapes & Modes	18. Arrays
8 Comments	19. Examples Part
9 Drawing Order	19.1 Wiggle Lines
10 Variables	19.2 Noise Spiral
11 While Loop	19.3 Polygon Scribble
12 Color	20. Export
14 Structure	

1. Introduction

Processing is a simple programming environment that was created to make it easier to develop visually oriented applications with an emphasis on animation and providing users with instant feedback through interaction. The developers wanted a means to sketch ideas in code.

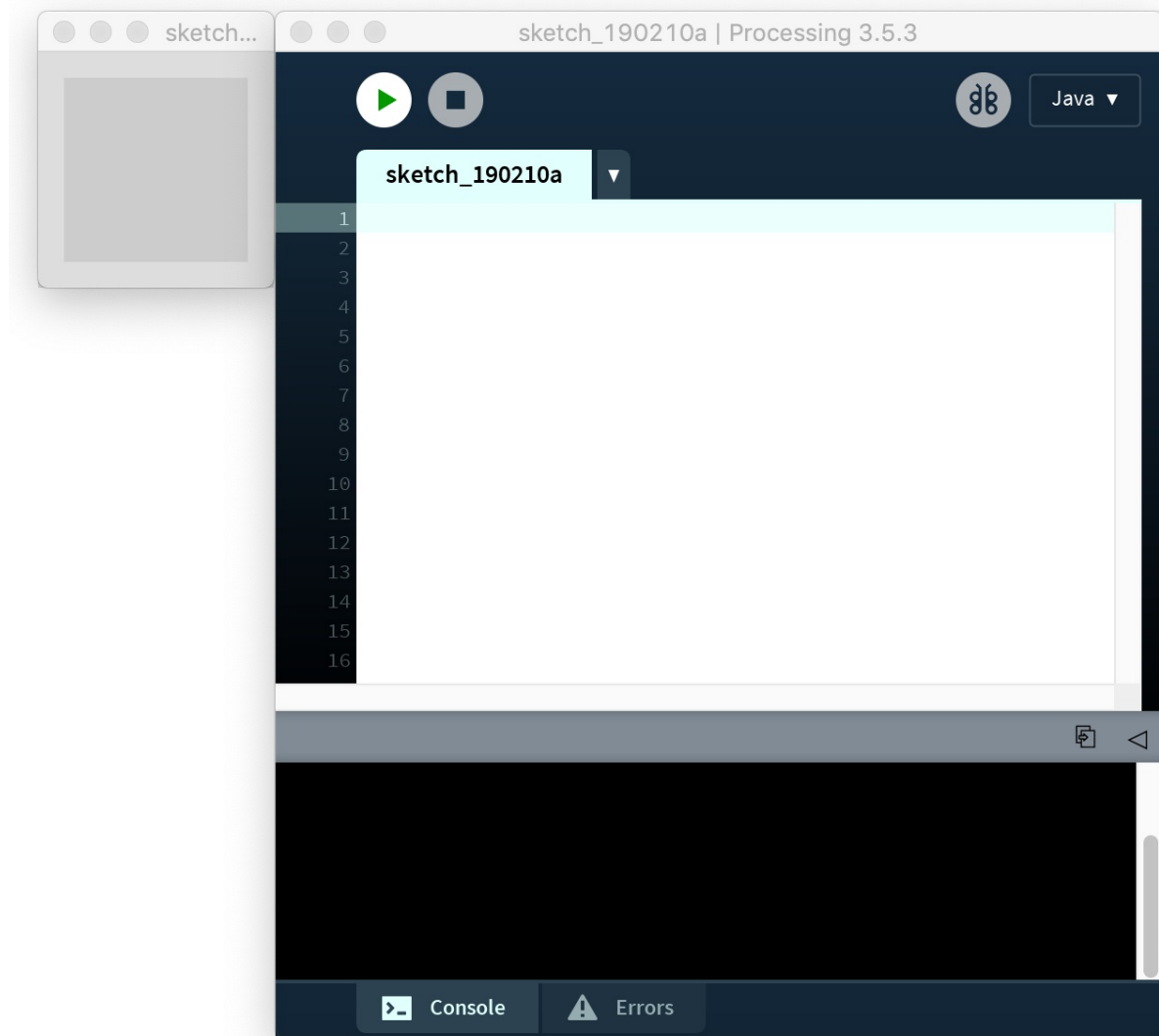
As its capabilities have expanded over the past decade, Processing has come to be used for more advanced production-level work in addition to its sketching role. Originally built as a domain-specific extension to Java targeted towards artists and designers, Processing has evolved into a full-blown design and prototyping tool used for large-scale installation work, motion graphics, and complex data visualization.

Examples of Processing usages can be found on <https://processing.org/exhibition/>

The latest version of Processing can be downloaded at <http://processing.org/download>

2. Sketching

A Processing program is called a sketch. The idea is to make Java-style programming feel more like scripting, and adopt the process of scripting to quickly write code. Sketches are stored in the sketchbook, a folder that's used as the default location for saving all of your projects. Sketches that are stored in the sketchbook can be accessed from File → Sketchbook. Alternatively, File → Open... can be used to open a sketch from elsewhere on the system.

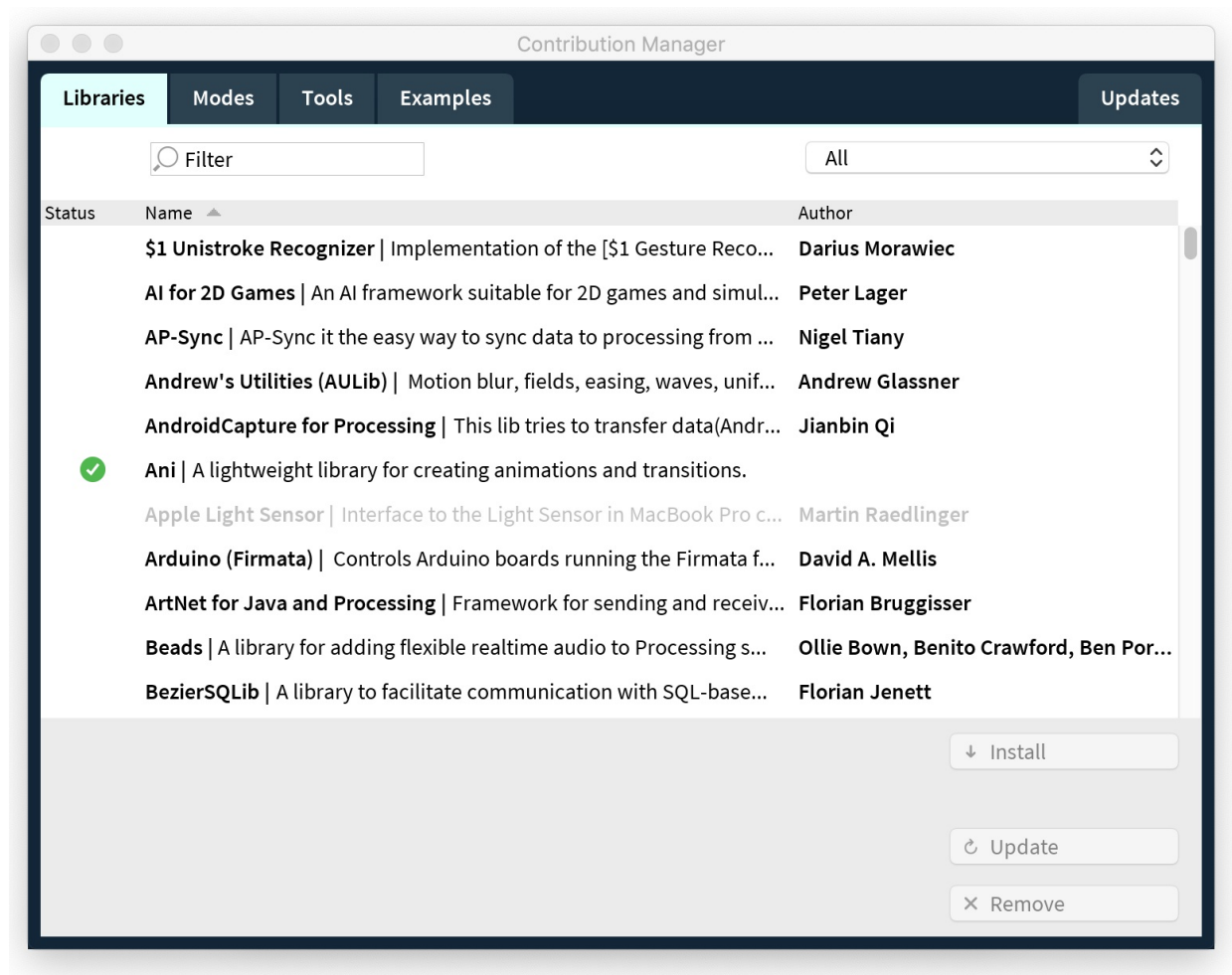


The Processing Development Environment

The large area is the Text Editor, the row buttons on top is the tool-bar. Below the editor is the Console, which can be used for messages and more technical details. The small square is the Display Window, the graphical output of the Sketch.

3. Libraries

The core functionality of Processing should be sufficient in the beginning. Yet, if you need to do something that's not available in Processing, you can use a library that adds the functionality you need. Libraries can be installed by opening Sketch → Import Library → Add Library.



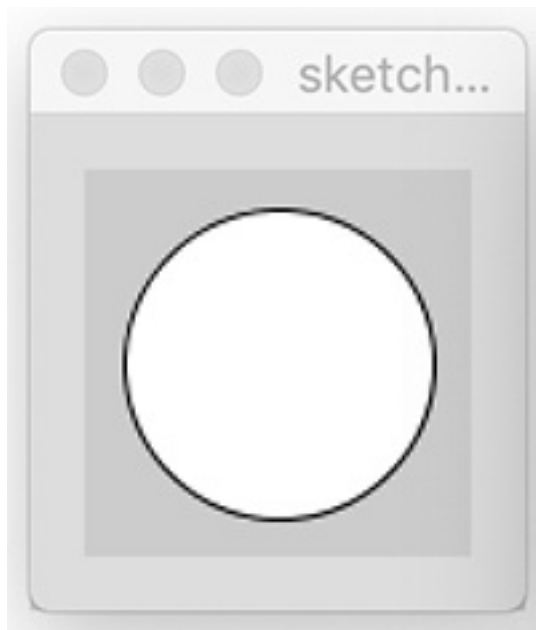
The Processing Contribution Manager

4. Your First Program

In the editor, type the following:

```
ellipse(50, 50, 80, 80);
```

This line of code means “draw an ellipse, with the center 50 pixels over from the left and 50 pixels down from the top, with a width and height of 80 pixels.” Click the Run button in the tool-bar or Sketch → Run.



If you've typed everything correctly, you'll see the ellipse image above. If you didn't type it correctly, the Console Area will turn red and complain about an error. If this happens, make sure that you've copied the example code exactly: the numbers should be contained within parentheses and have commas between each of them, and the line should end with a semicolon.

One of the most difficult things about getting started with programming is that you have to be very specific about the syntax. The Processing software isn't always smart enough to know what you mean, and can be quite fussy about the placement of punctuation. You'll get used to it with a little practice.

5. The Coordinate System

Processing uses the upper-left corner for the origin of the window. CAD applications usually prefer a different point for the origin of their drawing surface.

The `size()` function sets the dimensions of the sketch window. The default is `size(100,100)`. The first parameter is used to set the value of the system variable `width`, the second parameter is used to set the value of the system variable `height`. So, if the display window is 100×100 pixels, the upper-left is (0, 0), the center is at (50, 50), and the lower-right is (99, 99) or (`width-1`, `height-1`). If you now want to draw a point at the origin, you'll use:

```
point(0, 0);
```

This line of code will fill the first pixel on the first row. As we start counting at 0, the last pixel on that row would be

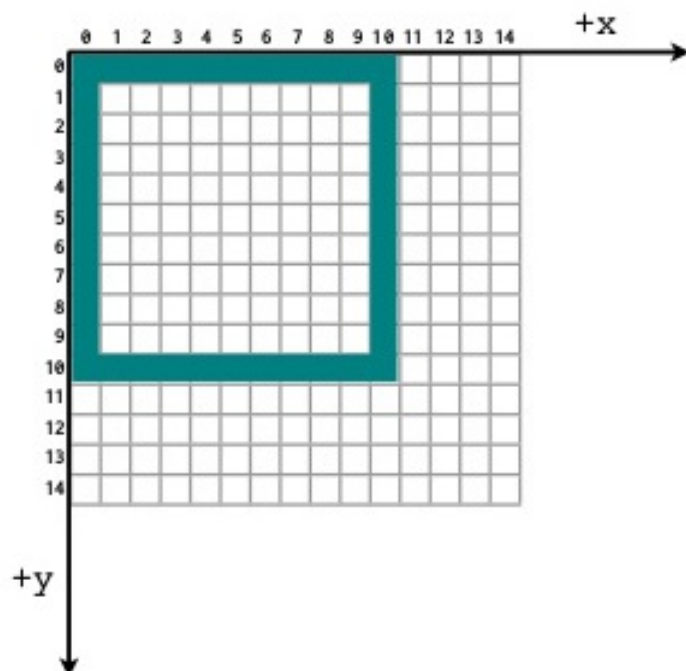
99, or width-1. The same is true for the height.

The following code will draw a red point with a thickness of five pixels at (0,0) and another one at (99,99)

```
strokeWeight(5);  
stroke(255,0,0);  
point(0, 0);  
point(width-1, height-1);
```



```
stroke(0, 128, 128);  
noFill();  
rect(0, 0, 10, 10);
```

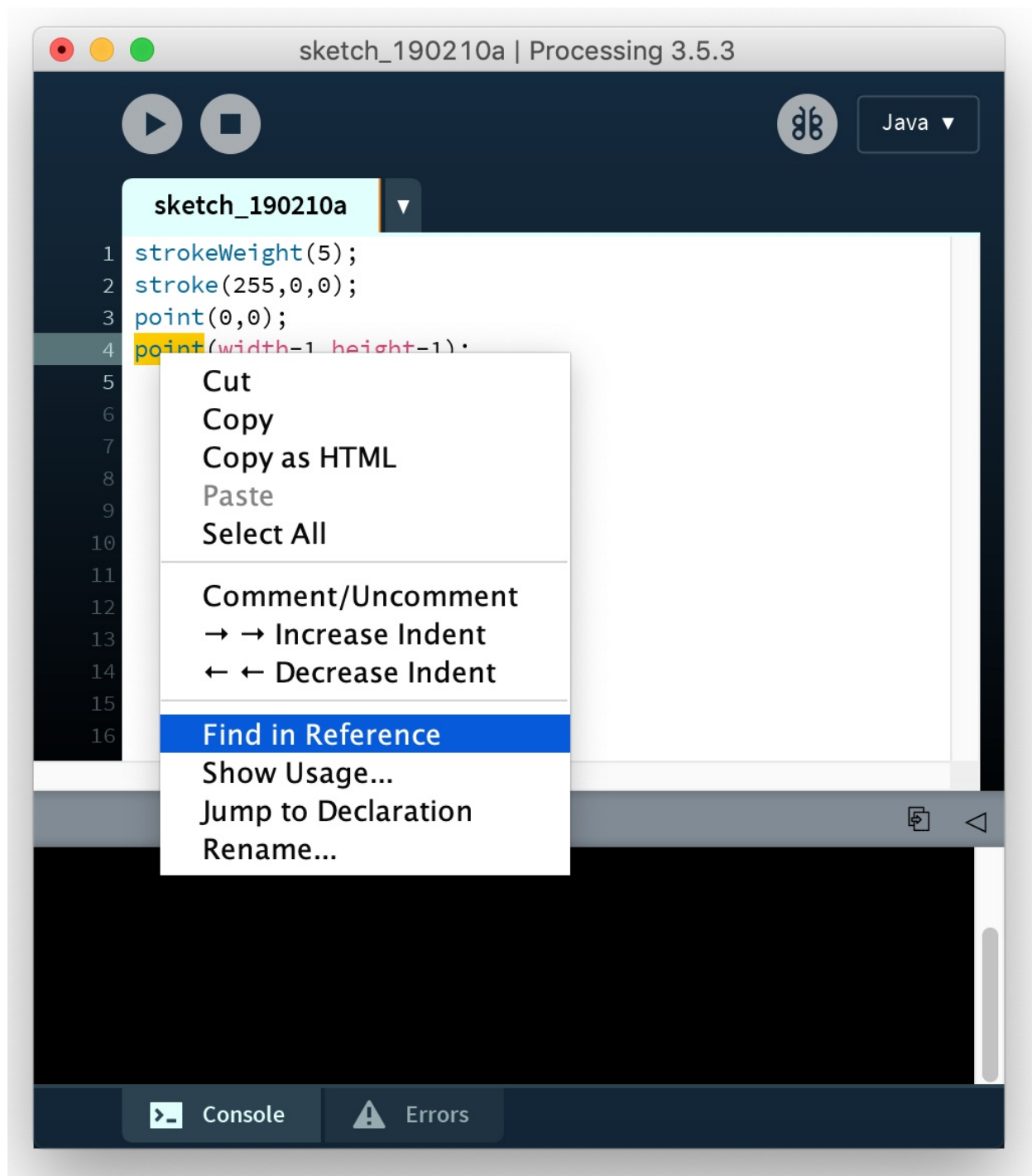


The Processing coordinate system

6. Functions

Functions are the basic building blocks of a Processing program. The behaviour of a function is defined by its parameters, a set of arguments enclosed in parentheses. Each function call must always end with a semicolon.

Processing will execute a sequence of functions one by one and finish by displaying the drawn result in a window. An overview of all functions can be found in the Processing reference (<https://processing.org/reference/>). The usage of a certain function is explained by highlighting a function and right-clicking or through Help → Find in Reference.



Finding a function in reference

Name `point()`

Examples



```
noSmooth();  
point(30, 20);  
point(85, 20);  
point(85, 75);  
point(30, 75);
```



```
size(100, 100, P3D);  
noSmooth();  
point(30, 20, -50);  
point(85, 20, -50);  
point(85, 75, -50);  
point(30, 75, -50);
```

Description

Draws a point, a coordinate in space at the dimension of one pixel. The first parameter is the horizontal value for the point, the second value is the vertical value for the point, and the optional third value is the depth value. Drawing this shape in 3D with the z parameter requires the P3D parameter in combination with `size()` as shown in the above example.

Use `stroke()` to set the color of a `point()`.

Syntax

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

Description of the `point()` function as found in the reference

7. Basic Shapes & Modes


Processing includes a group of functions to draw basic shapes or 2D primitives. Simple shapes like lines can be combined to create more complex forms like a leaf or a face. To draw a single line, we need four parameters: two for the starting location and two for the end.

The basic shape primitive functions are:

```
arc(), circle(), ellipse(),  
line(), point(), quad(),  
rect(), square(), triangle().
```


```
point(20, 30);
```

(20, 30)



```
line(3, 5, 30, 50);
```


(3, 5)



(30, 50)

```
rect(0, 0, 10, 10);
```

(0, 0)

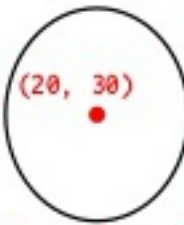


10

10

```
ellipse(20, 30, 40, 60);
```

(20, 30)



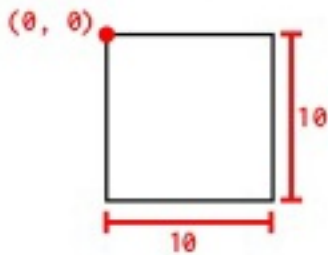
60

40

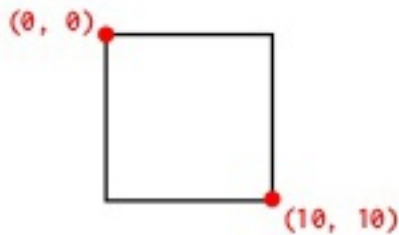
common shapes in Processing and their parameters

You must have noticed that `rect()` and `ellipse()` deal differently with the location from which they are drawn. You can change this default behaviour with the functions `rectMode()` & `ellipseMode()` to either CORNER, CORNERS, CENTER, or RADIUS.

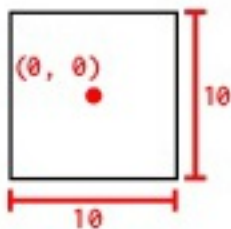
```
rectMode(CORNER);  
rect(0, 0, 10, 10);
```



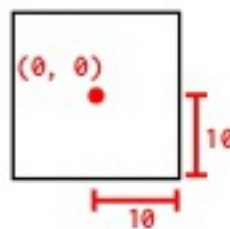
```
rectMode(CORNERS);  
rect(0, 0, 10, 10);
```



```
rectMode(CENTER);  
rect(0, 0, 10, 10);
```



```
rectMode(RADIUS);  
rect(0, 0, 10, 10);
```



the 4 different drawmodes of a rect & ellipse

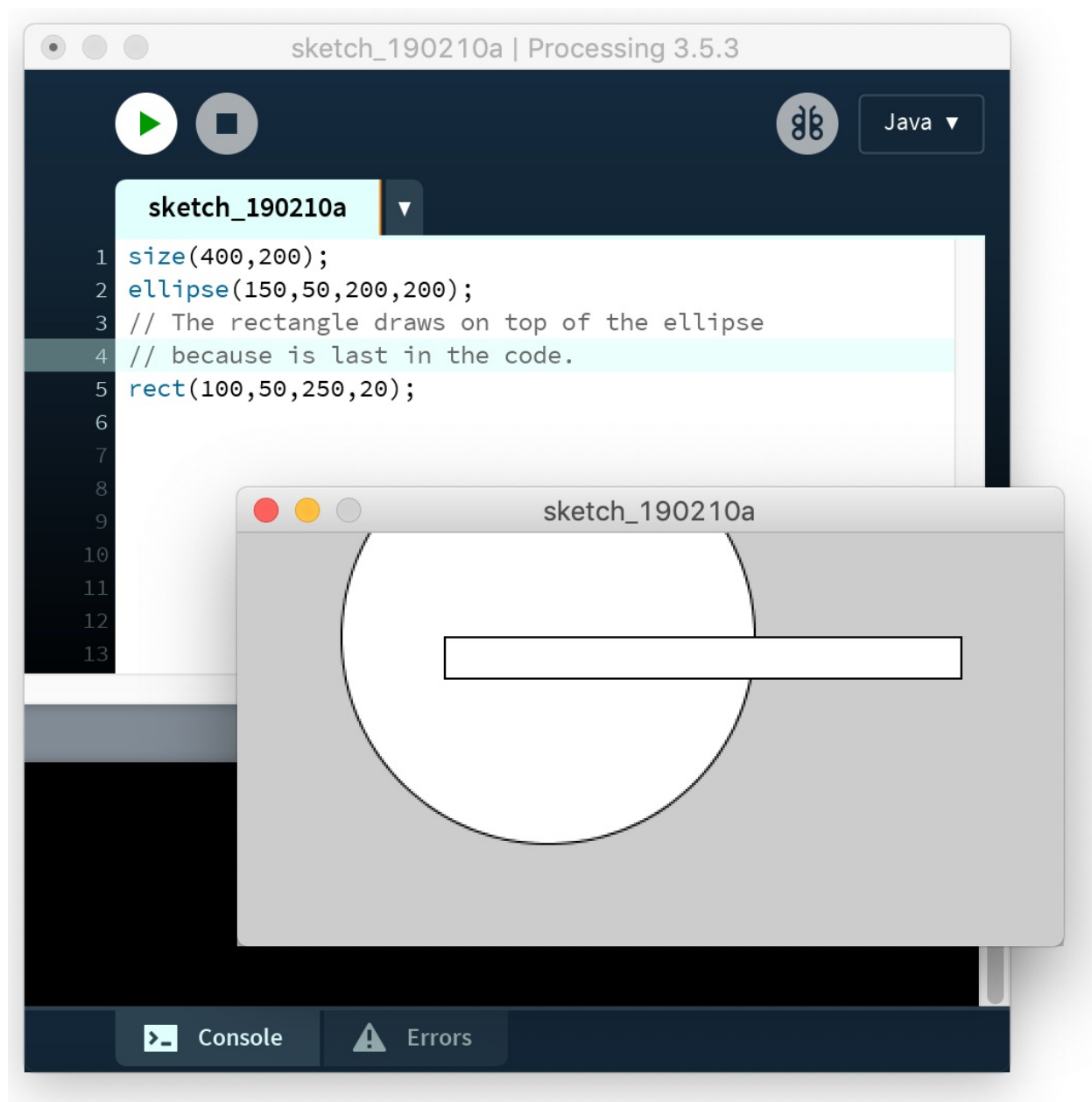
8. Comments

Comments are parts of the program that are ignored when the program is run. They are useful for making notes for yourself that explain what's happening in the code. If others are reading your code, comments are especially important to help them understand your thought process. Comments are also useful for trying things in your code without losing the original attempt.

```
// This is a comment on one line  
/* This is a comment that  
   spans several lines  
   of code */
```

9. Drawing Order

When a program runs, the computer starts at the top and reads each line of code until it reaches the last line and then stops. If you want a shape to be drawn on top of all other shapes, it needs to follow the others in the code.



Processing drawing order

10. Variables

A variable stores a value in memory so that it can be used later in a program. The variable can be used many times within a single program, and the value is easily changed while the program is running. When you create variables, you determine the name, the data type, and the value. The name is what you decide to call the variable. Choose a name that is informative about what the variable stores, but be consistent and not too verbose. For instance, the variable name “radius” will be clearer than “r” when you look at the code later. When declaring a variable, you also need to specify its data type (such as `int`), which indicates what kind of information is being stored. There are data types to store each kind of data: integers (whole

numbers), floating-point (decimal) numbers, characters, words, images, fonts, and so on. After the data type and name are set, a value can be assigned to the variable. Remember that each variable can only be used once with the same name in the same part of the program.

```
int x = 12; // Declare x as an int variable and assign a value
```

The most common Processing data types:

Data	Type	Example of Usage	Usage Description
char	char	varName = 'a';	A letter or Unicode symbol, such as a or #. Note the single quotation marks used around the symbol.
int	int	varName = 12;	An integer (a whole number). Can be positive or negative.
float	float	varName = 1.2345;	A floating-point number. A number that may have a decimal point.
boolean	boolean	varName = true;	A true or false value. Used for logical operations because it can only ever be one of two states.
String	String	varName = "hello";	A list of chars, such as a sentence. Note the capital S on String, signifying that this is a composite type (a collection of chars).

11. While Loop

As you write more programs, you'll notice that patterns occur when lines of code are repeated, but with slight variations. A code structure called a loop makes it possible to run a line of code more than once to condense this type of repetition into fewer lines. This makes your programs more modular and easier to change.

```
int number = 99;
while (number > 0) {
    println(number);
}
```

```
    number--;  
}  
println("zero");
```

This outputs the value of the variable *'number'* to the console window 99 times. The while command checks a condition and, if the condition is met, executes the code inside the braces; it then loops back up to the top of the block. The execution continues to the final line only after the condition is no longer met (in this case, when *'number'* is 0).

Note that if you don't include the *'number--'* line inside the loop, which subtracts 1 from the number every time it loops, the condition will never be met and the loop will go on forever.

12. For Loop

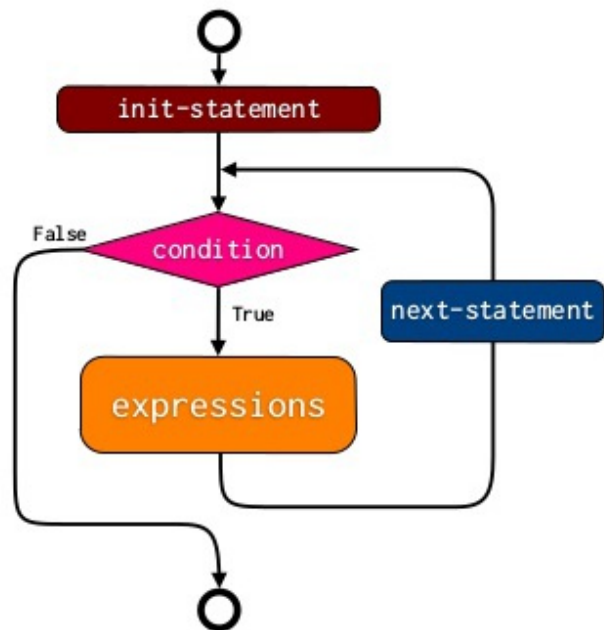
The for loop is used when you want to iterate through a set number of steps, rather than just wait for a condition to be satisfied. The syntax is as follows:

```
for (init; test; update) {  
    code to be executed  
}
```

The code between the curly brackets { } is called a block. This is the code that will be repeated on each iteration of the loop. Inside the parentheses are three statements, separated by semicolons, that work together to control how many times the code inside is run. From left to right, these statements are referred to as the initialization (init), the test, and the update. The *'init'* typically declares a new variable to use within the for loop and assigns a value. The variable name *'i'* is frequently used. The *'test'* evaluates the value of this variable, and the *'update'* changes it's value.

```
for (init; cond; next)
{
  expressions
}
```

- init-statement
- condition
- next-statement
- expressions



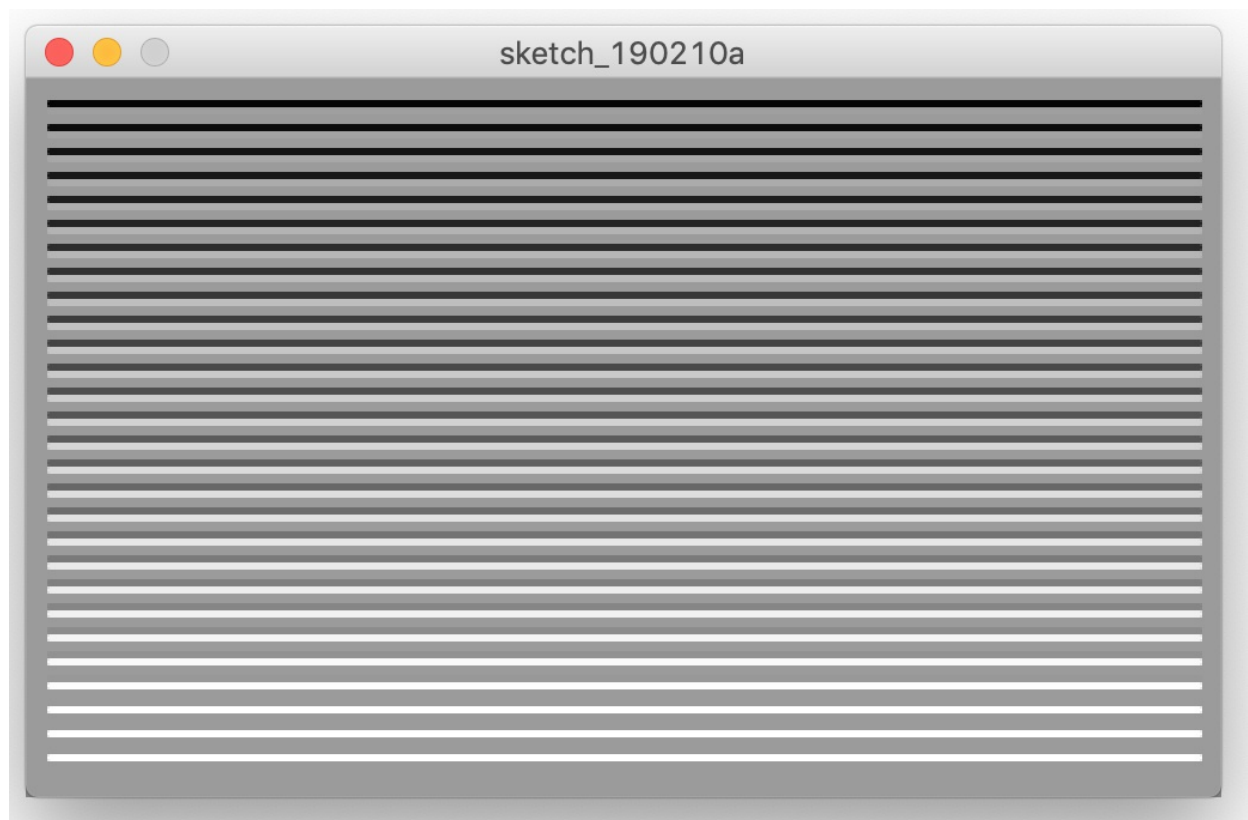
Flow diagram of a for loop

The test statement is always a relational expression that compares two values with a relational operator. The most common relational operators are:

- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to
- == Equal to
- != Not equal to

The relational expression always evaluates to true or false. When it's true, the code inside the block is run, when it's false, the code inside the block is not run and the for loop ends (sketch_01).

```
size(500, 300);
background(155);
strokeWeight(3);
for (int h = 10; h <= (height - 15); h+=10) {
  stroke(0, 255-h);
  line(10, h, width - 10, h);
  stroke(255, h);
  line(10, h+3, width - 10, h+3);
}
```

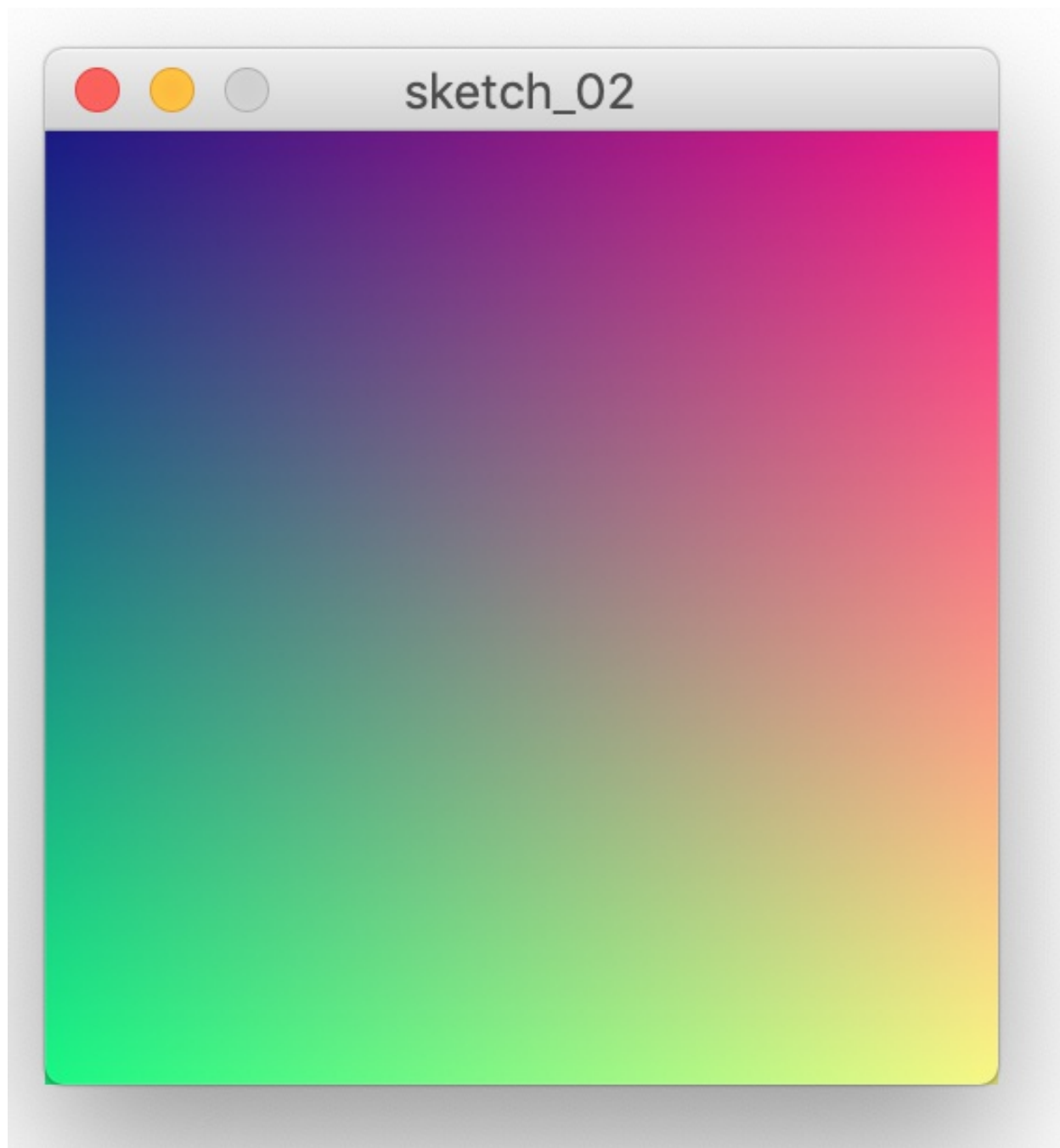


The initial state of the for loop sets a variable `h` to 10. The code in the loop executes until `h <= (height-15)` (the end condition). Every time the loop is executed, the value of `h` increases by 10, according to the step you've defined (`h += 10`). This means the code inside the parentheses of the for loop will execute 28 times, with `h` set to 10, 20, 30 ... 270, 280. Knowing that the `h` variable follows this pattern, you can use it in multiple ways. The lines you're drawing are in 10-pixel steps down the canvas, because you use `h` for the `y` value. But the alpha transparency of the lines also varies as `h` varies: the black line gets lighter, and the white line gets darker.

13. Color

To change color in your shapes use the `background()`, `fill()`, and `stroke()` functions. The values of the parameters are in the range of 0 to 255, where 255 is white, 128 is medium gray, and 0 is black. To move beyond gray-scale values, you use three parameters to specify the red, green, and blue components of a color. They also range from 0 to 255. Using RGB color isn't very intuitive, so to choose colors, you can use Tools → Color Selector. By adding an optional fourth parameter to `fill()` or `stroke()`, you can control the transparency. This fourth parameter is known as the alpha value, and also uses the range 0 to 255 to set the amount of transparency. The value 0 defines the color as entirely transparent (it won't display), the value 255 is entirely opaque, and the values between these extremes cause the colors to mix on screen (sketch_02).


```
size(255, 255);  
for (int y=0; y<height; y+=1) {  
  for (int x=0; x<width; x+=1) {  
    stroke(x, y, 122);  
    point(x, y);  
  }  
}
```



When one for loop is embedded inside another, the number of repetitions is multiplied. For each line in y-direction ($y < \text{height}$) the code iterates through every pixel in x-direction ($x < \text{width}$) and draws a point at the respective location with a red and green color value

corresponding to x and y.

14. Structure: setup() and draw()

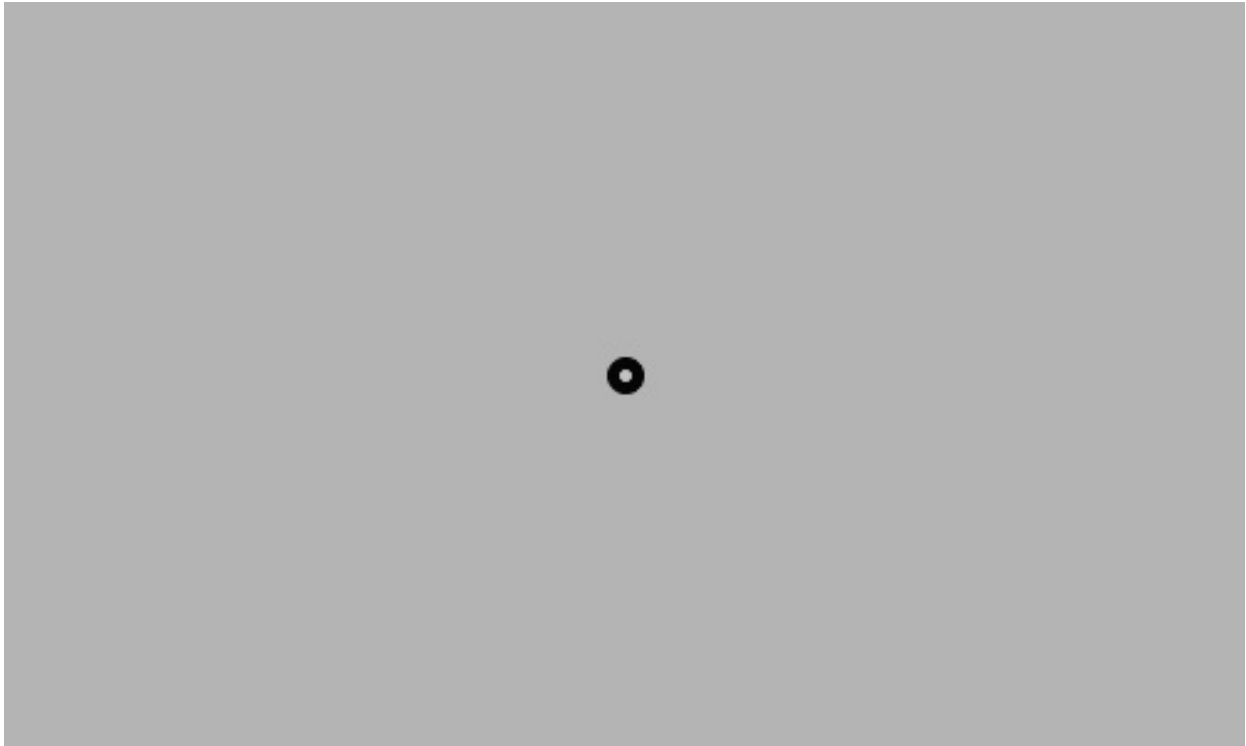
In Processing scripts can be structured into two distinct parts, so-called function blocks:

`setup()` and `draw()`. A function block is a way of chunking a group of commands together. The code inside the `setup()` function block is called once when the program launches, so it should contain all your initialization code as setting the canvas size, setting the background color, initializing variables, and so on. The code you write inside `draw()` is then called repeatedly, triggered on every frame. You can set the speed with which `draw()` is called by using the `frameRate()` function. If you give it a number (12, 24, 25, and 30 are typical), it will attempt to maintain that rate, calling `draw()` regularly. Otherwise, it will perform the frame loop as quickly as the machine can handle (sketch_03).

```
int diam = 10;
float centX, centY;

void setup() {
  size(500, 300);
  frameRate(24);
  smooth();
  background(180);
  centX = width/2;
  centY = height/2;
  stroke(0);
  strokeWeight(5);
  fill(255, 50);
}

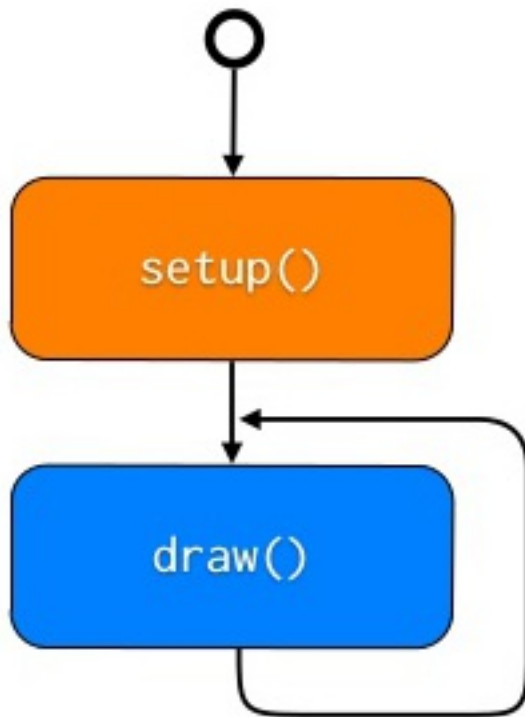
void draw() {
  if (diam <= 400) {
    background(180);
    ellipse(centX, centY, diam, diam);
    diam += 10;
  } else {
    diam = 10;
  }
}
```



When you run this you'll see a circle grow, stopping when the diameter reaches 400 pixels. The diameter and the center points are kept in variables, with the center points calculated in 'setup()'. The frame loop then checks that the diameter is smaller than 400, redraws the background, draws the circle, and increases the diameter by 10 for the next time it goes around the loop. The effect is that it draws a circle of diameter 10, 20, 30, and so on until the diameter variable gets to 400.

Notice that if you create a variable inside of setup(), you can't use it inside of 'draw()' and vice versa. A variable within a function block is only available within that block - locally. It's good practice to do this if a variable is only needed within a single function. Yet in order to make variables available everywhere, you need to place them somewhere else. Such variables are called global variables, because they can be used anywhere or globally in the program. This is clearer when we list the order in which the code is run:

1. Variables declared outside of 'setup()' and 'draw()' are created.
2. Code inside 'setup()' is run once.
3. Code inside 'draw()' is run continuously.



15. Conditionals

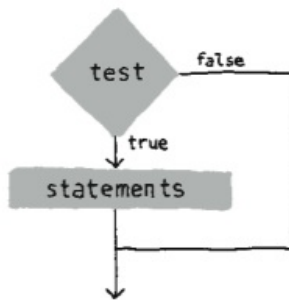
A conditional checks that a condition has been met before executing the code inside the block marked by the braces that follow it. In this case, the conditional asks whether the value of `diam` is less than or equal to 400. If it is, the code in the block executes. If not, the code in the block is skipped:

```
// check a condition
if (diam <= 400) {
  // execute code between the braces
  // if condition is met
}
```

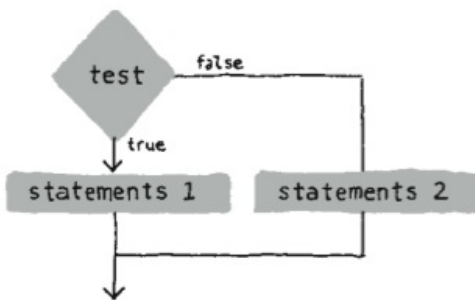
You can also use an `else` clause to provide a block of code to be executed if the condition isn't met:

```
if (diam <= 400) {
  // execute this code if diam <= 400
} else {
  // execute this code if diam > 400
}
```

```
}
```



```
if (test) {  
    statements  
}
```



```
if (test) {  
    statements 1  
} else {  
    statements 2  
}
```

If you imagine the flow of execution as a trickle of water running down the script, by setting a conditional you're effectively creating different channels for the stream to follow. With an if ... else clause, the stream can go one of two ways, either through the block or around. In addition to operators as described in [12. For Loop](#) you can also use logic operators to group conditions:

- `||` logical OR
- `&&` logical AND
- `!` logical NOT

move on [the next level](#).