▶ Table of Contents - click to expand!

P5.JS • Foundation: expressions, variables, conditionals, loops, …

In the previous tutorial, we went through the basics of working with p5.js. This tutorial goes over a couple of fundamental concepts of computer programming with expressions (or math), variables, conditionals and loops. We will apply these concepts in working with motion.
But we will start with something that is pointless for the machine but, in time, will certainly prove its value to yourself and potential co-coders, *comments*.

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 loosing the original attempt.

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

Expressions (or math)

An expression is a way to make a new value by combining other values with mathematical operators. This is incredibly useful.

the p5.js calculator

```
console.log(1 + 1);
console.log(0 - 1);
console.log(1 * 0.01);
console.log(1 / 2);
```

`console.log` (or `print`) is a function that prints a message to your browser's web console as seen in previous chapter.

a more exciting visual example

```
function setup() {
  createCanvas(400, 300);
```

```
    stroke(0);
    strokeWeight(2);
    background(238);
    noFill();
    rectMode(CENTER);
    // rect
    rect(width/2, height*0.5, 200, 200);
    // circle
    ellipse(width/2, height*0.5, 200, 200);
    // triangle
    triangle(width/2, 51, width-113, 200, 113, 200);
  }
```

Note:

- Everything happens in the `setup()` function as we don't need to write this over and over. Just once is fine.
- The 'width' and 'height' variables contain the width and height of the display window as set in the createCanvas() function. If we change the canvas size we don't have to change all the shape drawing functions.
- `rectMode(CENTER)` is far more handy here than the default `rectMode(DEFAULT`.
- /2 is actually the same *0.5

Common Mathematical Operators are:

Character	Operator
+	Addition
-	Subtraction
*	Multiplication
/	Division
=	Assignment

Yet there are some rules to remember:

- The order of operations matters, the PEMDAS rule. 3+2*3 will yield 9. If you want it to result 15 you must write it as (3+2)*3.
- You can not use **x** as a symbol for multiplication. x is a letter and will be viewed by p5.js as a variable.
- An equals sign works a bit differently than it does in math class. It is used to assign a value to a variable. More on this soon.

Variables

Now that you have an understanding of values and expressions, we can tackle one of the most powerful concepts in computer programming: **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.

A variable declaration has the following format:

- the keyword let (var is possible too), followed by
- the name of the variable (you get to pick this!), followed by
- an equals sign (=), followed by
- a value you want to “assign” to the variable.

The name is what you decide to call the variable. Choose a name that is informative about what the variable stores, but be consistent, not too criptical nor too verbose. For instance, the variable name 'radius' will be clearer than 'r' when you look at the code later.

When declaring a variable in processing (that is Java and not Javascript as p5.js) you also need to specify its **data type**, which indicates what kind of information is being stored. The most common data types are: integers (whole numbers), floating-point (decimal) numbers, booleans (true or false), characters, words or strings, and so on. In p5.js you **do not need to do this**. This is actually handy but can also be confusing as you still need to know what kinds of variables there are and how they work.

To make things even confusing there are two words we can use to set a variable, **"let"** and **"var"**. The difference between them is in their 'scope'. But mostly both will work. Let is newer, and will probably become the standard, so let's use let. If you want to learn about the difference check this video.

Declaring the variable before the setup area means the variable is **'global'** and accessible throughout the entire sketch. 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, thus is **'local'**. It's good practice to do this if a variable is only needed within a single function.

9 vertical lines

```
let ypos = 180;
let lenght = 40;

function setup() {
  createCanvas(400, 200);
}

function draw() {
  background(255);
  line(100, ypos, 100, lenght);
  line(125, ypos, 125, lenght);
  line(150, ypos, 150, lenght);
  line(175, ypos, 175, lenght);
  line(200, ypos, 200, lenght);
  line(225, ypos, 225, lenght);
  line(250, ypos, 250, lenght);
  line(275, ypos, 275, lenght);
  line(300, ypos, 300, lenght);
}
```

The real power of variables is that you can use them in any context that you would normally need to write a value. This means that you can use variables in expressions. The example above can be adapted as followed.

9 vertical lines bis

```
let xpos = 100;
let xstep = 25;
let ypos_top = 20;
let ypos_bottom = 180;

function setup() {
  createCanvas(400, 200);
}

function draw() {
  background(255);
  line(xpos + (xstep * 0), ypos_top, xpos + (xstep * 0), ypos_bottom);
  line(xpos + (xstep * 1), ypos_top, xpos + (xstep * 1), ypos_bottom);
  line(xpos + (xstep * 2), ypos_top, xpos + (xstep * 2), ypos_bottom);
  line(xpos + (xstep * 3), ypos_top, xpos + (xstep * 3), ypos_bottom);
  line(xpos + (xstep * 4), ypos_top, xpos + (xstep * 4), ypos_bottom);
  line(xpos + (xstep * 5), ypos_top, xpos + (xstep * 5), ypos_bottom);
  line(xpos + (xstep * 6), ypos_top, xpos + (xstep * 6), ypos_bottom);
  line(xpos + (xstep * 7), ypos_top, xpos + (xstep * 7), ypos_bottom);
  line(xpos + (xstep * 8), ypos_top, xpos + (xstep * 8), ypos_bottom);
}
```

There is actually a way to more compactly express this set of instructions with **a loop**. More on that later. First we will add some randomness to our drawing and set our sketch in motion.

Random

Unlike the smooth, linear motion common to computer graphics, motion in the physical world is usually idiosyncratic. We can simulate the unpredictable qualities of the world by generating random numbers. The `random()` function calculates these values and we can set a range to tune the amount of disarray in a program.

The following short example prints random values to the console, with the range limited by the x position (on the horizontal axis) of the mouse. The `random()` function always returns a floating-point value.

```
function draw() {
  let r = random(0, mouseX);
  console.log(r);
}
```

Random Dots

```
let x, y;  // create two variables x, y for position

function setup() {
  createCanvas(400, 400);
  background(0);
  noStroke();
}

function draw() {
  fill(255, 120, 0, 250);
  x = random(0, width);
  y = random(0, height);
  ellipse(x, y, 15, 15);
  filter(BLUR, 1);
}
```

note:

- `let x, y;` is a shorthand notation for `let x; let y;`
- The filter BLUR executes a Gaussian blur, the parameter 1 specifies the intensity of the filter.

back to our 9 lines in the wind

```
let xpos = 100;
let xstep = 25;
let ypos_top = 20;
let ypos_bottom = 180;
let wind = 5;

function setup() {
  createCanvas(400, 200);
}

function draw() {
  background(255);
  line(xpos + (xstep * 0)+random(-wind,wind), ypos_top, xpos + (xstep * 0), ypos_bottom);
  line(xpos + (xstep * 1)+random(-wind,wind), ypos_top, xpos + (xstep * 1), ypos_bottom);
  line(xpos + (xstep * 2)+random(-wind,wind), ypos_top, xpos + (xstep * 2), ypos_bottom);
  line(xpos + (xstep * 3)+random(-wind,wind), ypos_top, xpos + (xstep * 3), ypos_bottom);
  line(xpos + (xstep * 4)+random(-wind,wind), ypos_top, xpos + (xstep * 4), ypos_bottom);
```

```
    line(xpos + (xstep * 5)+random(-wind,wind), ypos_top, xpos + (xstep * 5), ypos_bottom);
    line(xpos + (xstep * 6)+random(-wind,wind), ypos_top, xpos + (xstep * 6), ypos_bottom);
    line(xpos + (xstep * 7)+random(-wind,wind), ypos_top, xpos + (xstep * 7), ypos_bottom);
    line(xpos + (xstep * 8)+random(-wind,wind), ypos_top, xpos + (xstep * 8), ypos_bottom);
  }
```

Adding the line `wind = 1 + mouseX/20;` in our draw loop will make the random range restricted from 1 to 20 (400/20) depending on the x position of the mouse.

Note: There is actually a nicer, less machine-like, random function, `noise()` form Perlin noise. It produces a more naturally ordered, harmonic succession of numbers. It was invented by Ken Perlin in the 1980s and been used since in graphical applications to produce procedural textures, natural motion, shapes, terrains etc.

Linear Motion

We have seen that code inside the `draw()` function is called on every program cycle repeatedly and we can set the speed with the `frameRate()` function.

Well, another power of using variables is we can change them on every cycle.

our line on the move

```
let xpos = 10;
let xstep = 25;
let ypos_top = 20;
let ypos_bottom = 180;

function setup() {
  createCanvas(400, 200);
  background(255);
  frameRate(5);
}

function draw() {
  line(xpos, ypos_top, xpos, ypos_bottom);
  xpos += xstep;
  filter(BLUR,1);
}
```

When you run this you'll see the line move from left to right. It's position on the x axis is kept in a variable as well as the step (or step size) by which it moves. The draw loop draws the line, and increases the x position diameter by 25.

It would be good if we could prevent the line from moving into infinity. Wouldn't it? With condtionals in the next chapter we can.

Note:

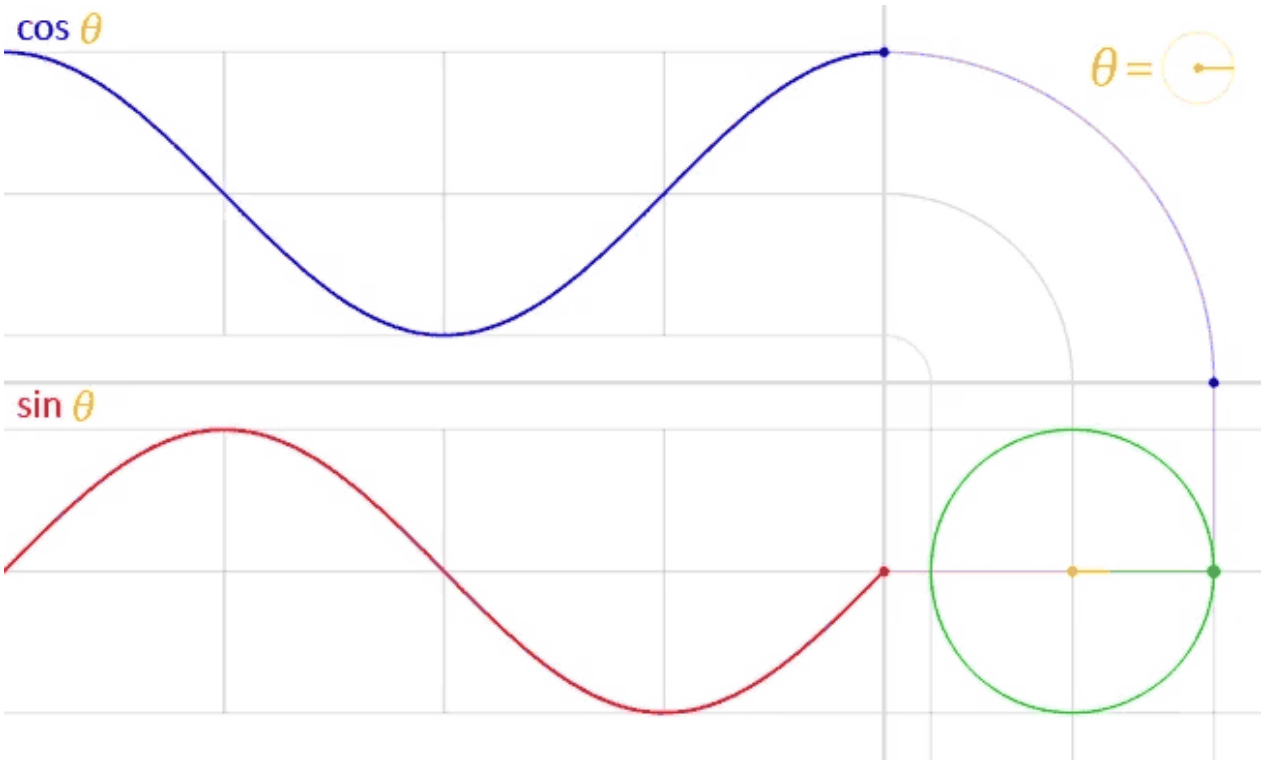
- `xpos += xstep;` is actually a shorthand notation of `xpos = xpos + xstep;`
- Even so is writing `a++` equivalent to `a = a + 1` and writing `a--` equivalent to `a = a - 1`.

Circular Motion

Circular motion is a movement in which an object travels along the circumference of a circle. However, this simple movement has much more beauty in it than it might seem. We will only lift a tip of the veil of a very fascinating domain including Simple Harmonic Motion. Think about the swing of a pendulum, a weight that swings up and down on a spring.

Working with circular motion requires a little bit of trigonometry knowledge but we will limit this to the sine / `sin()` and cosine / `cos()` functions and their relationship.

Sine and Cosine? Basically, if you were to move around the perimeter of a circle, your horizontal position would trace out a cosine function while your vertical position would trace out a sine.



How do angles work in p5.js

Angles are set in radians rather than degrees. Radians are angle measurements based on the value of pi (3.14159).

A full circle is 360 DEGREES, which is equal to TWO_PI (2π) in RADIANS.
furthermore 45° = QUARTER_PI, 90° = HALF_PI and 180° = PI
See this chart on the conversion between degrees and radians

If you prefer to use degree measurements, you ca convert to radians using the `radians()` function or use the `angleMode(DEGREES)` function.

Sine and Cosine

Using sin(angle) * radius, we can calculate the x coordinate of a point on the circumference of a circle.
Using cos(angle) * radius, we can calculate the y coordinate of the same point.

As a result, sine and cosine are two numbers that oscillate between 1 and -1 according to angle change.

```
let radius;
let angle = 0;
let speed = 0.05;
let xOffset, yOffset;

function setup() {
  createCanvas(400, 300);
  radius = width / 4;
  xOffset = (width/2);
  yOffset = (height/2);
}

function draw() {
  background(200);
  // Empty Circle as path
```

```
  noFill();
  stroke(100);
  circle(0+xOffset, 0+yOffset, radius * 2);
  // Rotating Circle
  noStroke();
  fill(255,0,0);
  let x = cos(angle) * radius;
  let y = sin(angle) * radius;
  circle(x+xOffset, y+yOffset, 20);
  // Increase angle every frame
  angle += speed;
}
```

Note:

- To make our circle travel around the centre of the canvas, we need to work with those xOffset and yOffset variables. The `translate()` function, that we will see later, simplifies this process considerably.
- add the code below just before the `angle += speed;` line to see the sine and cosine in action in a simple harmonic motion.

```
circle(xOffset-50, yOffset, cos(angle)*100)
circle(xOffset+50, yOffset, sin(angle)*100)
```

Challenge: modify the code to create a spiralling motion.

Conditionals

Conditions are like questions. They allow a program to decide to take one action if the answer to a question is **"true"** or to do another action if the answer to the question is **"false"**.

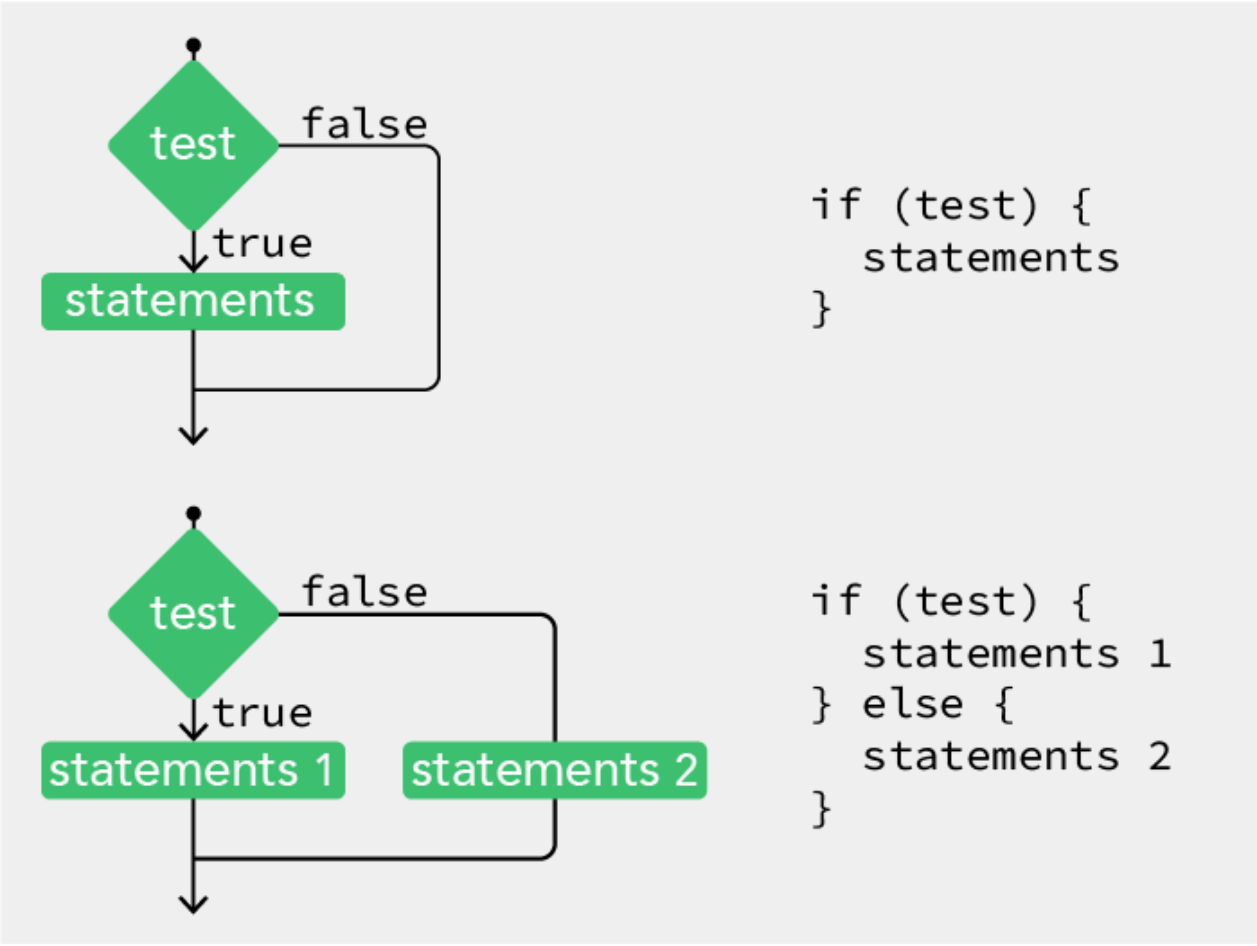
Thus, it checks that a condition has been met before executing the code inside the block marked by the braces that follow it.

In the case below, 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
}
```



Flow diagram of conditional test

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.

The most common relational operators are:
Character	Operator
>	Greater than
<|less than |>=| Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

In addition you can also use **logic operators** to group conditions:
Character	Operator
			logical OR
&&	logical AND
!	logical NOT

our line on the move in a bounded loop

```
let xpos = 10;
let xstep = 25;
let ypos_top = 20;
let ypos_bottom = 180;
```

```
function setup() {
  createCanvas(400, 200);
  background(255);
  frameRate(5);
}

function draw() {
  line(xpos, ypos_top, xpos, ypos_bottom);
  filter(BLUR, 0.8);
  if (xpos > width - 10) {
    xpos = 10;
  } else {
    xpos += xstep;
  }
}
```

Note:

- As is usually the case, you can do things in more than one way. The following was equally good.
  ```javaScript
  function draw() {
  line(xpos, ypos_top, xpos, ypos_bottom);
  filter(BLUR, 0.8);
  xpos += xstep;
  if (xpos > width - 10) {
  xpos = 10;
  }
  }

```
Thus the else part is not always necessary and can be omitted.


But we don't live in an either-or world and so sometimes just having an if and an else isn't enough.
Then **else if** comes to the rescue.

![](img/p5js_else_if.png)



#### Lets try a more advanced conditional with 2 tests
```

JavaScript
// Draw a Random Shape
let dice;
let border = 10;

function setup() {
createCanvas(300,300);
stroke(0);
background(255);
fill(0);
rectMode(CENTER);
dice = random(1);
if (dice < 0.333) { // circle ellipse(width / 2, height / 2, width - border, height - border); } else if ((dice > 0.333) && (dice < 0.666)) {
// rect
rect(width / 2, height / 2, width - border, height - border);
} else {
// triangle
triangle(border / 2, 0 + border / 2, width - border / 2, height - border / 2, border / 2, height - border / 2);
```

}
}

```
Note:

- the **else-if** statement combines 2 relational expressions with **&&**
So, if the result of the function random given to the variable dice is greater than 0.333 **and** less than 0

- Recap. Any **if** statement can have any number of associated **else if** clauses. Even if you have an **el



#### a Bouncing Ball
And to finish this chapter on conditionals a somewhat classic example

Draw an ellipse that moves from left to right. When it reaches the right hand side of the sketch, make it mov
```

javaScript
let xspeed = 4;
let xpos = 0;
let diam = 50;

function setup() {
createCanvas(400, 400);
}

function draw() {
background(50);
stroke(255);
strokeWeight(8);
ellipse(xpos, 200, diam, diam);
if (xpos > width-diam/2) {
xspeed = -4;
}
if (xpos < diam/2) {
xspeed = 4;
}
xpos += xspeed;
}

```
## While Loop

As you write more programs, you'll notice that patterns occur when lines of code are repeated, but with sligh
```

JavaScript
let number = 99;
function draw() {
while (number > 0) {
console.log(number);
number--;
}
console.log("zero");
}

```
This outputs the value of the variable *'number'* to the console window 99 times. The while command checks a

Note that if you don't include the 'number--' line inside the loop, which subtracts 1 from the number every t
```

```
![](img/p5js_while_loop.png)
<sub>Flow diagram of a while loop</sub>


#### Lets draw a line repeatedly as we did before.
```

javaScript
// a while loop example of our vertical lines
let xpos = 80;
let xstep = 20;
let ypos_top = 20;
let ypos_bottom = 180;

function setup() {
createCanvas(400, 200);
background(255);
}

function draw() {
while (xpos < width-80) {
line(xpos + (xstep * 0), ypos_top, xpos + (xstep * 0), ypos_bottom);
xpos += xstep;
}
}

```
And now try out adding the wind factor by displacing the top x coordinate!!
```

javaScript
let xpos = 80;
let xstep = 25;
let ypos_top = 20;
let ypos_bottom = 180;
let wind = 5;

function setup() {
createCanvas(400, 200);
}

function draw() {
background(255);
wind = 1 + mouseX/20;
while (xpos < width-80) {
line(xpos + (xstep * 0)+random(-wind,wind), ypos_top, xpos + (xstep * 0), ypos_bottom);
xpos += xstep;
}
xpos = 80;
}

```
Note:
- background(255) needs to move from the setup function to draw.
- you need to reset the variable xpos to is base value at th end.


## For Loop

The for loop is used when you want to iterate through a set number of steps, rather than just wait for a cond
```

```
![](img/p5js_for_loop.png)
<sub>Flow diagram of a for loop</sub>

The code between the curly brackets { } is called **a block**. This is the code that will be repeated on each

The test statement is always **a relational expression** that compares two values with a **relational operato
`>` Greater than, `<` Less than, `>=` Greater than or equal to, `<=` Less than or equal to, `==` Equal to, `!
The relational expression always evaluates to **true** or **false**. When it's true, the code inside the bloc
```

JavaScript
let origx = 200;
let origy = 100;
let destx = 200;
let desty = 300;

function setup() {
createCanvas(400, 400);
noLoop();
}

function draw() {
background(255);
fill(255);
strokeWeight(3);
for (let i = 25; i < 400; i += 25) {
stroke(50);
line(i, 200, origx, origy);
stroke(240);
line(i, 200, destx, desty);
}
}

```
The initial state of the for loop sets a variable h to 10. The code in the loop executes until h <= (height-1

#### Time for Serious Tiling.
```

JavaScript
// a line of circles
let diam = 30;
let step = 15;

function setup() {
createCanvas(510, 200);
background(255, 204, 0);
noFill();
}

function draw() {
for (var x = diam/2; x <= width - diam/2; x += step) {
ellipse(x, height / 2, diam, diam);
}
}

JavaScript
// 2 nested for loops forming a grid of circles
let diam = 30;
let step = 15;

function setup() {
createCanvas(510, 210);
background(255, 204, 0);
noFill();
}

function draw() {
for (let x = diam/2; x <= width - diam/2; x += step) {
for (let y = diam/2; y <= height - diam/2; y += step) {
ellipse(x, y, diam, diam);
}
}
}

```
#### the GOTO 10 example
"goto 10" refers to "10 PRINT CHR$(205.5+RND(1)); : GOTO 10"
a random maze generation program in one line of Commodore 64 Basic.
```

JavaScript
// GOTO 10 or Random Diagonal Lines in a grid
let dice = 0;
let tile = 20;

function setup() {
createCanvas(600, 600);
background(255);
stroke(0);
noLoop();
}

function draw() {
for (let x = tile / 2; x <= width; x += tile) {
for (let y = tile / 2; y <= height; y += tile) {
dice = random(1);
if (dice <= 0.5) {
line(x - tile / 2, y - tile / 2, x + tile / 2, y + tile / 2);
} else {
line(x - tile / 2, y + tile / 2, x + tile / 2, y - tile / 2);
}
}
}
}

```
```javaScript
// GOTO 10 the Horizontal / Vertical approach
let dice = 0;
let tile = 20;

function setup() {
  createCanvas(600, 600);
  background(255);
  stroke(0);
```
```

```
  noLoop();
}

function draw() {
  for (let x = tile / 2; x <= width; x += tile) {
    for (let y = tile / 2; y <= height; y += tile) {
      dice = random(1);
      if (dice <= 0.5) {
        line(x, y - tile / 2, x, y + tile / 2);
      } else {
        line(x - tile / 2, y, x + tile / 2, y);
      }
    }
  }
}
```

```
// Draw Random Shapes in a 10 by 10 grid
let dice = 0;
let tile = 30;
let gutter = 3;

function setup() {
  createCanvas(300, 300);
  background(255);
  stroke(0);
  fill(0);
  rectMode(CENTER);
  noLoop();
}

function draw() {
  for (let x = tile / 2; x <= width; x += tile) {
    for (let y = tile / 2; y <= height; y += tile) {
      dice = random(1);
      if (dice <= 0.5) {
        ellipse(x, y, tile - gutter, tile - gutter);
      } else {
        rect(x, y, tile - gutter, tile - gutter);
      }
    }
  }
}
```

```
// Draw a Full Grid of Random Shape
// Triangles can be orientated in 4 directions
let dice = 0;
let dice2 = 0;
let tile = 100;
let border = tile / 10;

function setup() {
  createCanvas(300, 300);
  background(255);
  fill(0);
  noStroke();
  rectMode(CENTER);
  noLoop();
}

function draw() {
  for (let x = tile / 2; x <= width; x += tile) {
    for (let y = tile / 2; y <= height; y += tile) {
      dice = random(1);
      dice2 = random(1);
      if (dice <= 0.333) {
        // circle
        console.log("circle");
        ellipse(x, y, tile - border, tile - border);
      } else if ((dice > 0.333) && (dice < 0.666)) {
```

```
        // rect
        console.log("square");
        rect(x, y, tile - border, tile - border);
      } else {
        if (dice2 <= 0.25) {
          // triangle 1
          console.log("triangle 1");
          triangle(x - tile / 2 + border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y + tile
        } else if ((dice2 > 0.25) && (dice2 <= 0.5)) {
          // triangle 2
          console.log("triangle 2");
          triangle(x - tile / 2 + border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y - tile
        } else if ((dice2 > 0.5) && (dice2 <= 0.75)) {
          // triangle 3
          console.log("triangle 3");
          triangle(x - tile / 2 + border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y - tile
        } else {
          // triangle 4
          console.log("triangle 4");
          triangle(x + tile / 2 - border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y + tile
        }
      }
    }
  }
}
```

nesting for loops

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

Our last sketch with color

```
// Draw a Full Grid of Random Shape
// Triangles can be orientated in 4 directions
// With colors this time
let dice = 0;
let dice2 = 0;
let tile = 100;
let border = tile / 10;

function setup() {
  createCanvas(300, 300);
  background(255);
  fill(0);
  noStroke();
  rectMode(CENTER);
  noLoop();
}

function draw() {
  for (let x = tile / 2; x <= width; x += tile) {
    for (let y = tile / 2; y <= height; y += tile) {
      dice = random(1);
      dice2 = random(1);
      fill(100*dice+155,dice2*255,100);
      if (dice <= 0.333) {
        // circle
        console.log("circle");
        ellipse(x, y, tile - border, tile - border);
      } else if ((dice > 0.333) && (dice < 0.666)) {
        // rect
        console.log("square");
        rect(x, y, tile - border, tile - border);
      } else {
        if (dice2 <= 0.25) {
          // triangle 1
          console.log("triangle 1");
```

```
          triangle(x - tile / 2 + border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y + tile
        } else if ((dice2 > 0.25) && (dice2 <= 0.5)) {
          // triangle 2
          console.log("triangle 2");
          triangle(x - tile / 2 + border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y - tile
        } else if ((dice2 > 0.5) && (dice2 <= 0.75)) {
          // triangle 3
          console.log("triangle 3");
          triangle(x - tile / 2 + border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y - tile
        } else {
          // triangle 4
          console.log("triangle 4");
          triangle(x + tile / 2 - border / 2, y - tile / 2 + border / 2, x + tile / 2 - border / 2, y + tile
        }
      }
    }
  }
}
```