The Joy of Coding Algorithmic Art Workbook #3 Arrays and Phyllotaxis

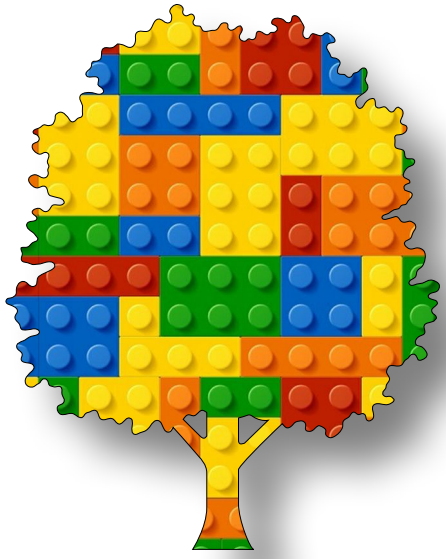


Table of Contents

MIT Licence

Workbook #3 Quick Content Summary

As you should have completed workbook #2 we can now dig a bit deeper into the code. Arrays are a very key part of coding and although you might wonder what they might have to do with Algorithmic Art, you will find out that they will be very useful. Then we will look at some other shapes that are not, what we call primitive shapes. Finishing off with a beautiful pattern found in nature.

The code is in the yellow boxes, any new code is highlighted in blue, so you don't have to type out the whole thing again each time. Set up an account (highly recommended) so you can save your work as you go along rather than starting again each time. It will also remind you where you were up to.

Most browsers work but Chrome is best as there is some functionality missing in some browsers. I suspect it has been tested on Chrome to guarantee it works in Chrome browser.

The Joy of Coding Algorithmic Art

Module B Unit #7 arrays

Module B Unit #7: arrays

An array is a key and vital part of coding. It is a way of storing data so that it can be accessed, added to, or altered at a later date. One way to think of it is as a series of boxes that can hold bits of data (whether numbers or words). An array has a numbering system for each box, called an `index`. In coding, counting starts with `zero`, not `one`. So the first box is `index 0`, the second box is `index 1`, the third is `index 2`, and so on (see Fig.1). You can imagine that it can be confusing that the third box is `index 2`.

The format to identify an array is square brackets such as `[23, 15, 37, 42, ...]` so we can describe this array as follows (see fig.2):
`index[0]` is `23`,
`index[1]` is `15`,
`index[2]` is `37`, and
`index[3]` is `42`, etc.

You may be wondering what arrays may have to do with creative coding; a lot is the quick answer, for there will be instances where you will want to store information, for instance, colour names or other values for accessing later.

Figure: 1 index numbering system

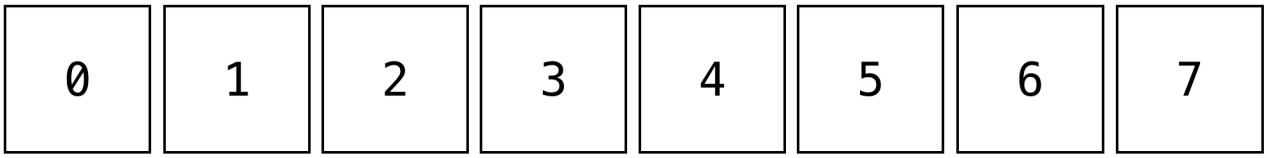
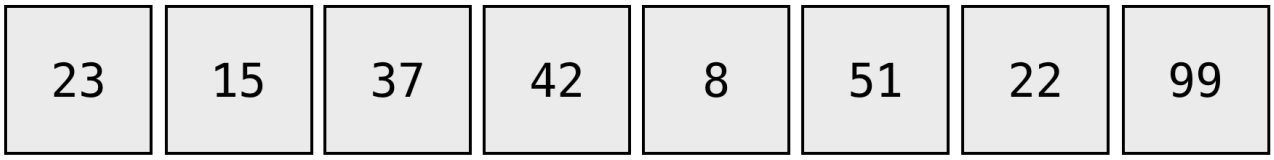


Figure 2: the values (elements) inside each box



On the top row are the `index[]` references and on the bottom row are the actual values at those reference points. We need to give the array a name. We can use let to define the array, such as:

`let numbers = []`	This is an empty array.

We can initialise it with some initial values if we want.

`let numbers = [23, 15, 37, 42, 8, 51, 22, 99]`	The array has now got some values in it

Sketch B7.1 starting sketch

! A new sketch

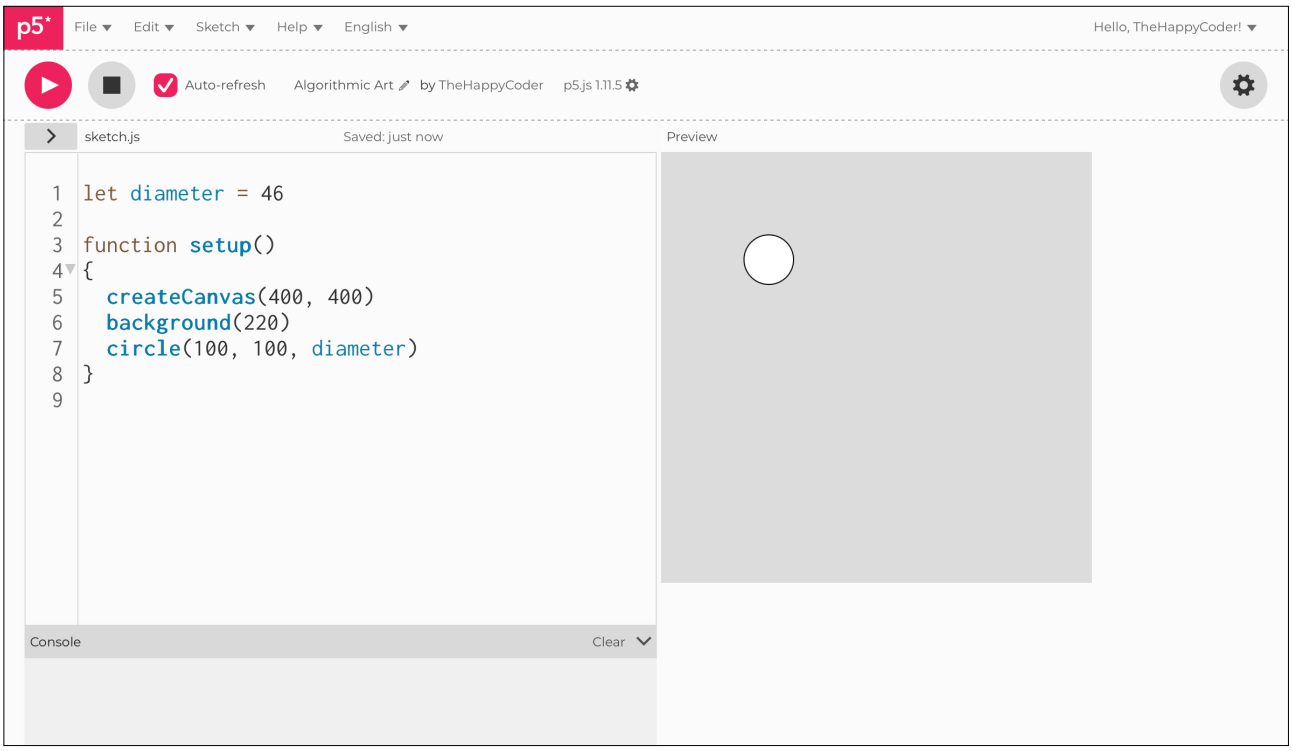
A simple sketch with a circle at (100, 100) with a diameter of 46.

```
let diameter = 46

function setup()
{
  createCanvas(400, 400)
  background(220)
  circle(100, 100, diameter)
}
```

Notes

We are doing all of this in the `setup()` function because later on we don't want to loop in the `draw()` function.

Figure B7.1



Sketch B7.2 an array

We will put the `diameter` in an array. To get at the diameter, we will need to access the array specifying the index number, which in this case will be index `0`. The array is called `diameter` and it is identified as an array by the square brackets `[]`.

```
let diameter = [46]


function setup()
{
  createCanvas(400, 400)
  background(220)
  circle(100, 100, diameter[0])
}
```

Notes

Well done, you have created your first array.

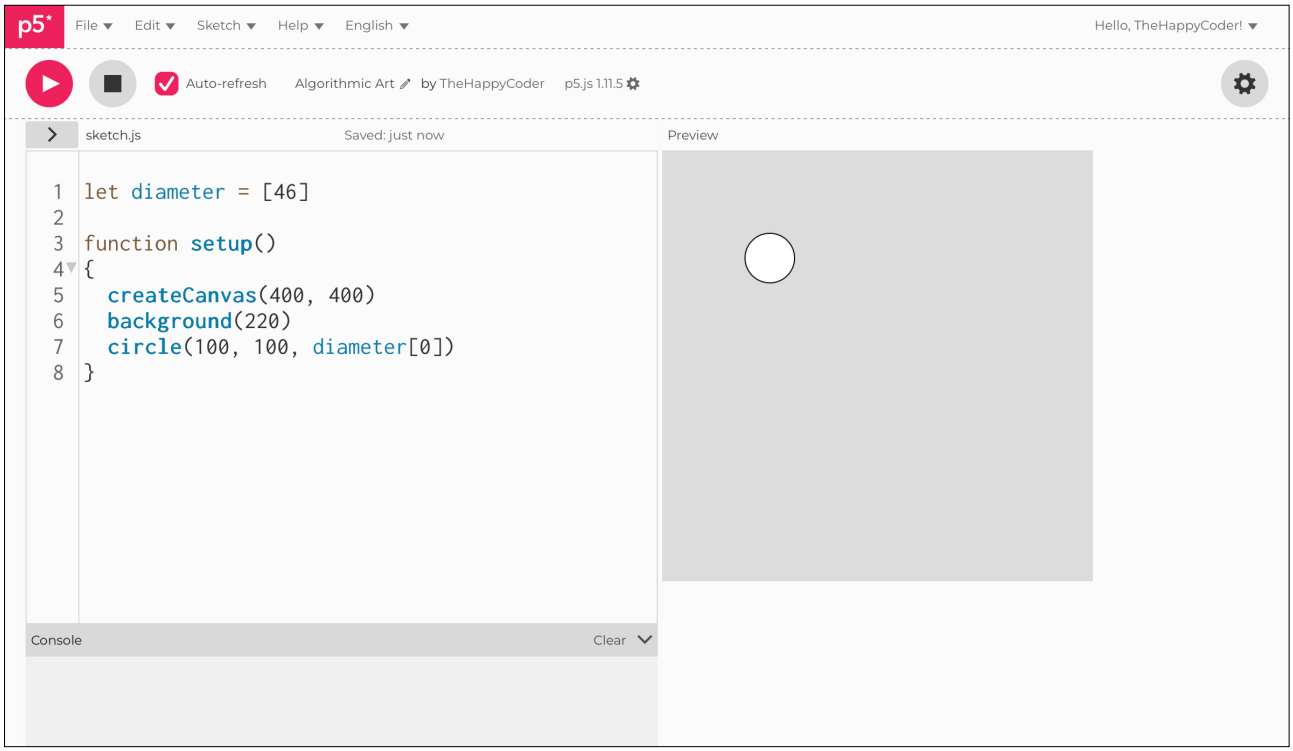
Challenge

What happens if you put a `1` instead of `0` in the index (`diameter[1]`)?

Code Explanation

`let diameter = [46]`	To make an array, we use square brackets `[]`.
`circle(100, 100, diameter[0])`	The array has one element at index `[0]`.

Figure B7.2



Sketch B7.3 a bigger array

We can put more numbers (elements) in this array and then draw each circle, spacing them out.

```
let diameter = [46, 12, 33, 18, 27]


function setup()
{
  createCanvas(400, 400)
  background(220)
  circle(100, 100, diameter[0])
  circle(100, 150, diameter[1])
  circle(100, 200, diameter[2])
  circle(100, 250, diameter[3])
  circle(100, 300, diameter[4])
}
```

Notes

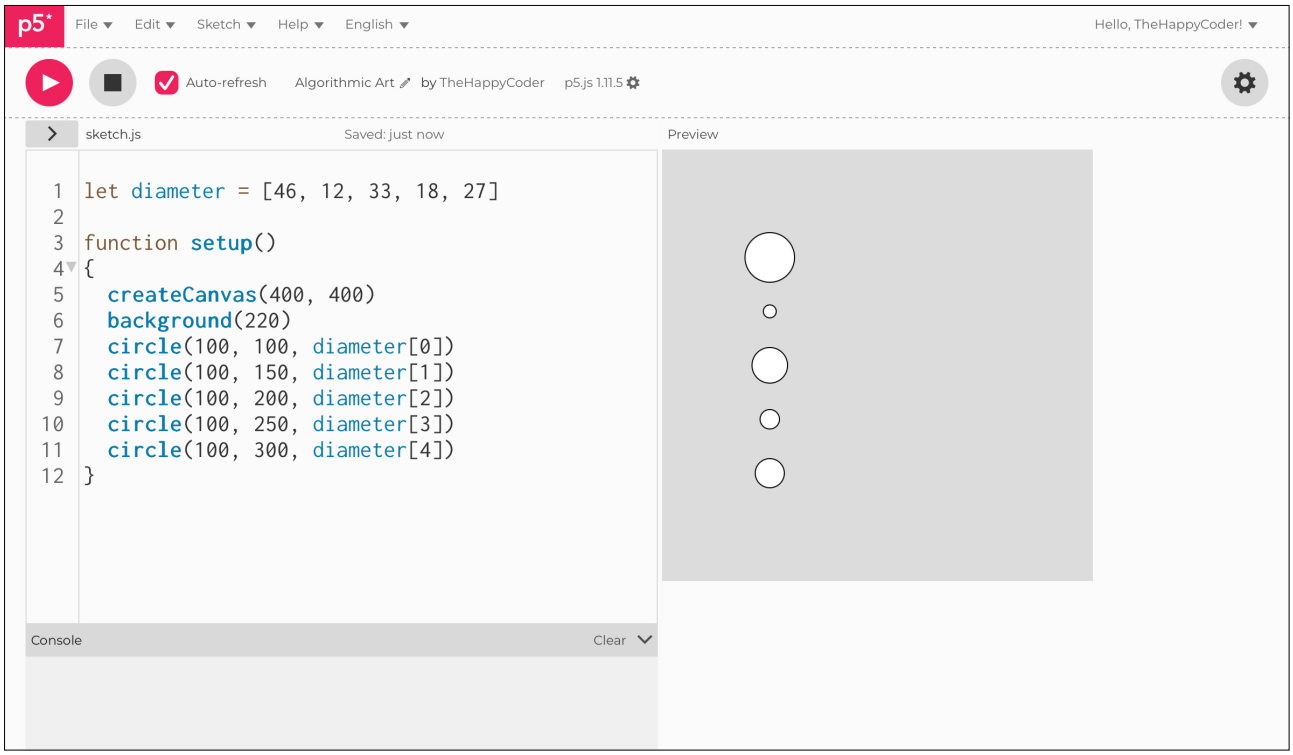
Notice we have to give an index number for each element of the array:

```
index [0] is 46
index [1] is 12
index [2] is 33
index [3] is 18
index [4] is 27
```

Code Explanation

`let diameter = [46, 12, 33, 18, 27]`	An array holding five elements.

Figure B7.3



Sketch B7.4 looping through the array

This isn't very efficient, so let's use a `for()` loop.

```
let diameter = [46, 12, 33, 18, 27]

function setup()
{
  createCanvas(400, 400)
  background(220)
  for (let i = 0; i < 5; i++)
  {
    circle(100, 100, diameter[i])
  }
}
```

Notes

We have them all on top of each other.

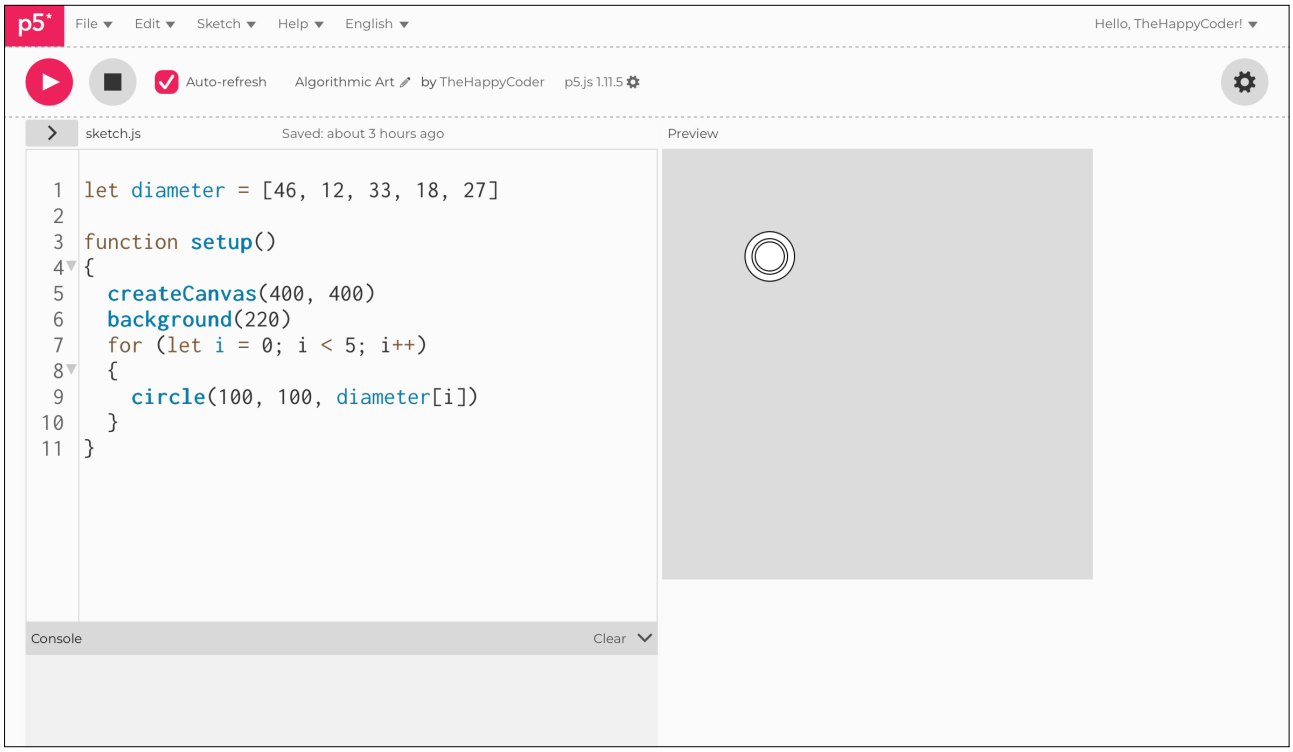
Challenge

How could we separate them?

Code Explanation

`circle(100, 100, diameter[i])`	We loop through each element one at a time, incrementing i by 1 each iteration.

Figure B7.4



Sketch B7.5 another array

We add another array for the y position of the circles. This allows us to space them back out again. We use the same principle for the y array.

```
let diameter = [46, 12, 33, 18, 27]
let y = [100, 150, 200, 250, 300]

function setup()
{
  createCanvas(400, 400)
  background(220)
  for (let i = 0; i < 5; i++)
  {
    circle(100, y[i], diameter[i])
  }
}
```

Notes

We are back where we were.

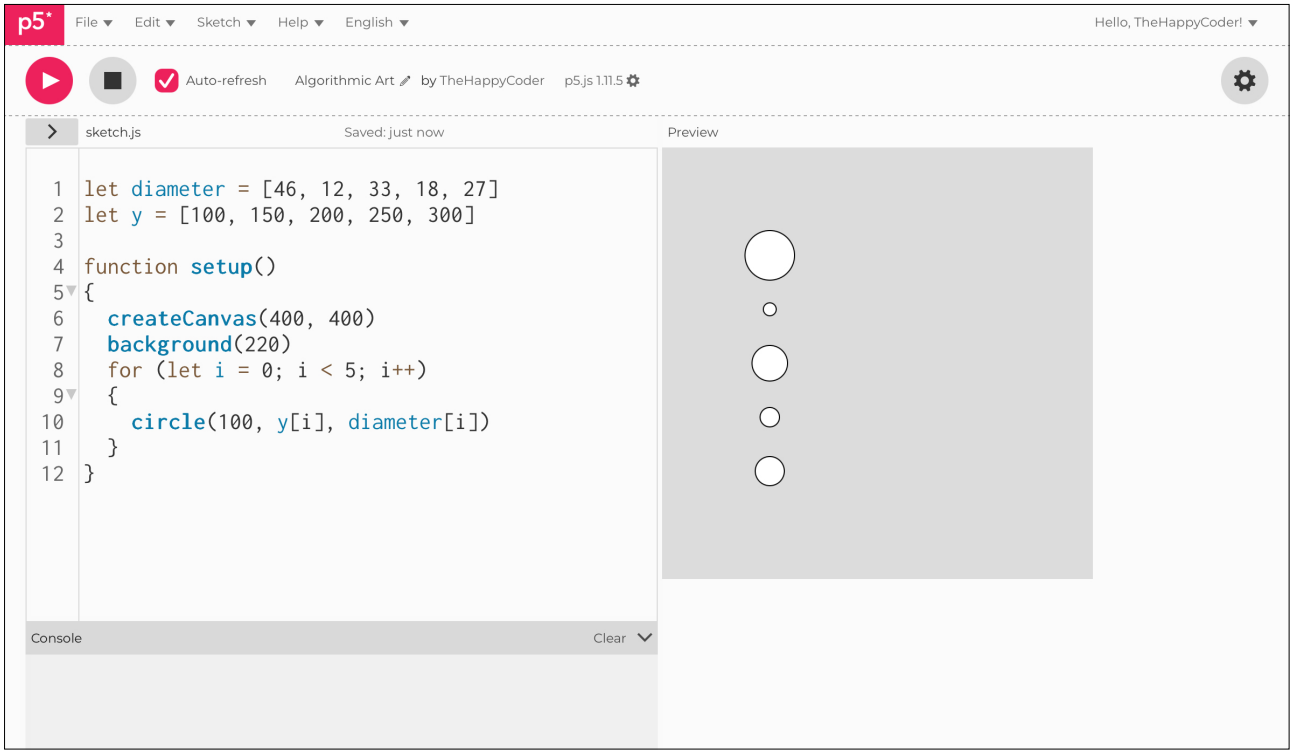
Challenge

Think of a better name for the array than y.

Code Explanation

`circle(100, y[i], diameter[i])`	We loop through two arrays, the diameter and the y position.

Figure B7.5



Sketch B7.6 an array of strings

Rather than numbers, we can also have strings of letters, words, or a combination of all three. Here, we will colour each circle with a corresponding colour from an array of named colours.

```
let diameter = [46, 12, 33, 18, 27]
let y = [100, 150, 200, 250, 300]
let colours = ['red', 'green', 'blue', 'orange', 'purple']

function setup()
{
  createCanvas(400, 400)
  background(220)
  for (let i = 0; i < 5; i++)
  {
    fill(colours[i])
    circle(100, y[i], diameter[i])
  }
}
```

Notes

A string is denoted by having speech (singular or double) marks.

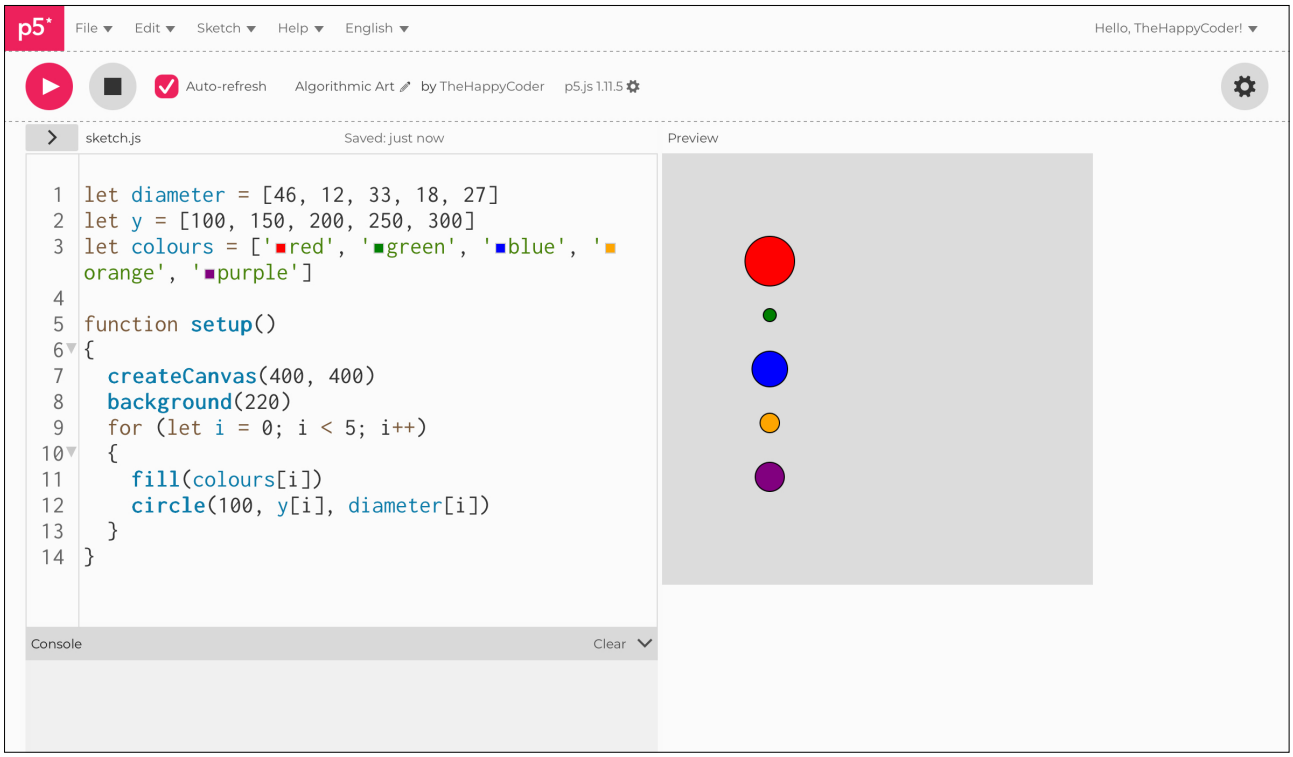
Challenge

Add your own colours.

Code Explanation

`fill(colours[i])`	Each element is the named colour.

Figure B7.6



Sketch B7.7 random selection

Another useful trick is random selection from an array; here, we will randomly select the colour of the circle.

```
let diameter = [46, 12, 33, 18, 27]
let y = [100, 150, 200, 250, 300]
let colours = ['red', 'green', 'blue', 'orange', 'purple']

function setup()
{
  createCanvas(400, 400)
  background(220)
  for (let i = 0; i < 5; i++)
  {
    fill(random(colours))
    circle(100, y[i], diameter[i])
  }
}
```

Notes

You may get the same colour chosen more than once.

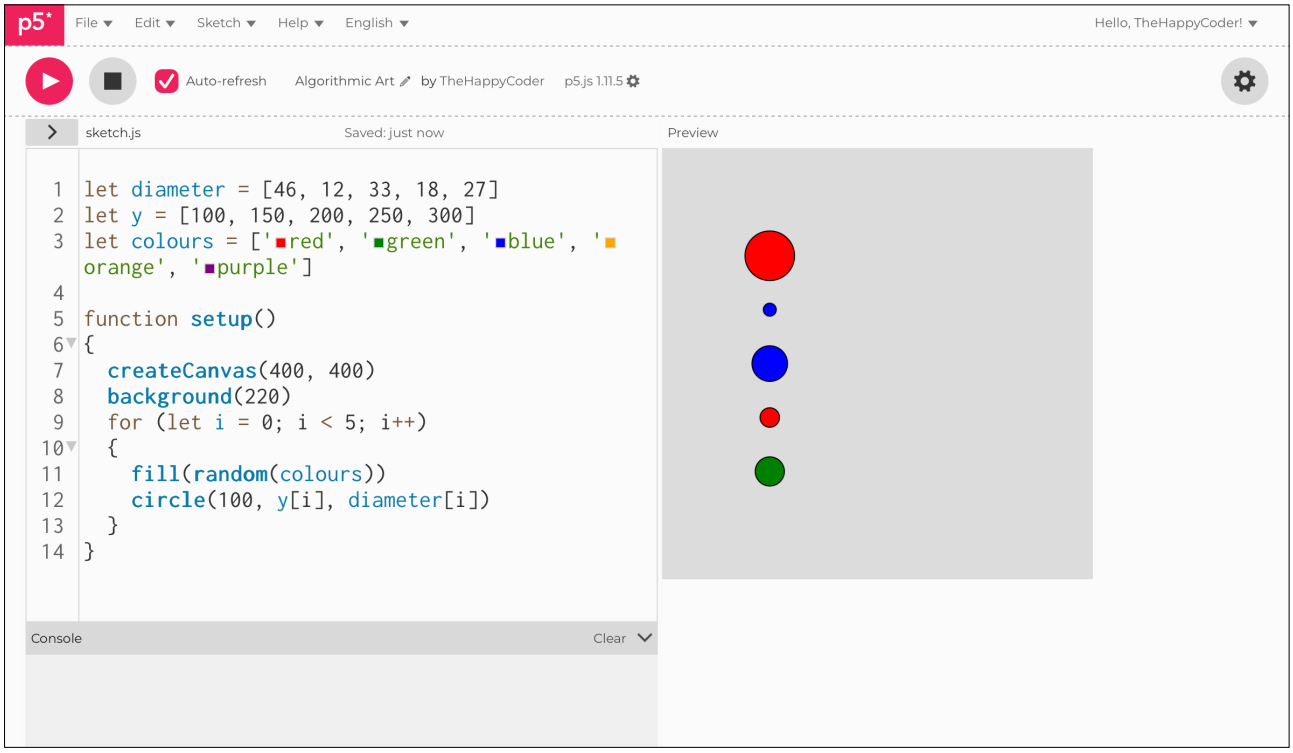
Challenges

1. Add more colours to the colour array.
2. Randomise the `y` position and the `diameter`.

Code Explanation

`fill(random(colours))`	The colours are randomly chosen for the array.

Figure B7.7



Sketch B7.8 array length

Rather than hardcoding the length of the array, we can use a function that already knows the length of the array.

```
let diameter = [46, 12, 33, 18, 27]
let y = [100, 150, 200, 250, 300]
let colours = ['red', 'green', 'blue', 'orange', 'purple']

function setup()
{
  createCanvas(400, 400)
  background(220)
  for (let i = 0; i < diameter.length; i++)
  {
    fill(random(colours))
    circle(100, y[i], diameter[i])
  }
}
```

Notes

Use `diameter.length` rather than hard-coding the value. This is useful if you have arrays that change in size because it is possible to run code that adds to the array (and removes elements).

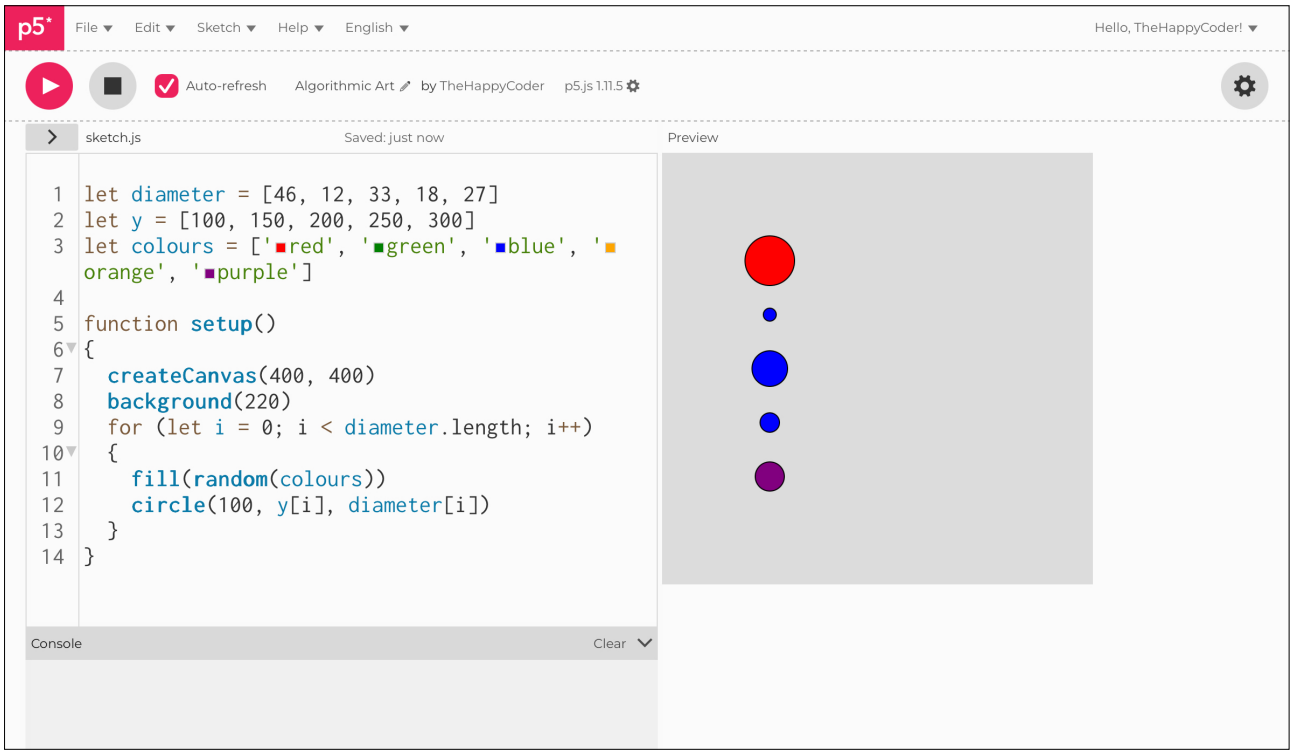
Challenge

What would happen if there were `six` elements in the `diameter` array but still only `five` in the `y` array?

Code Explanation

`for (let i = 0; i < diameter.length; i++)`	The diameter length looks at the array and works out how many elements it has.

Figure B7.8



Sketch B7.9 empty array

! Start a new sketch

We have created two empty arrays, one for x and one for y.

```
let x = []
let y = []

function setup()
{
  createCanvas(400, 400)
  background(220)
}
```

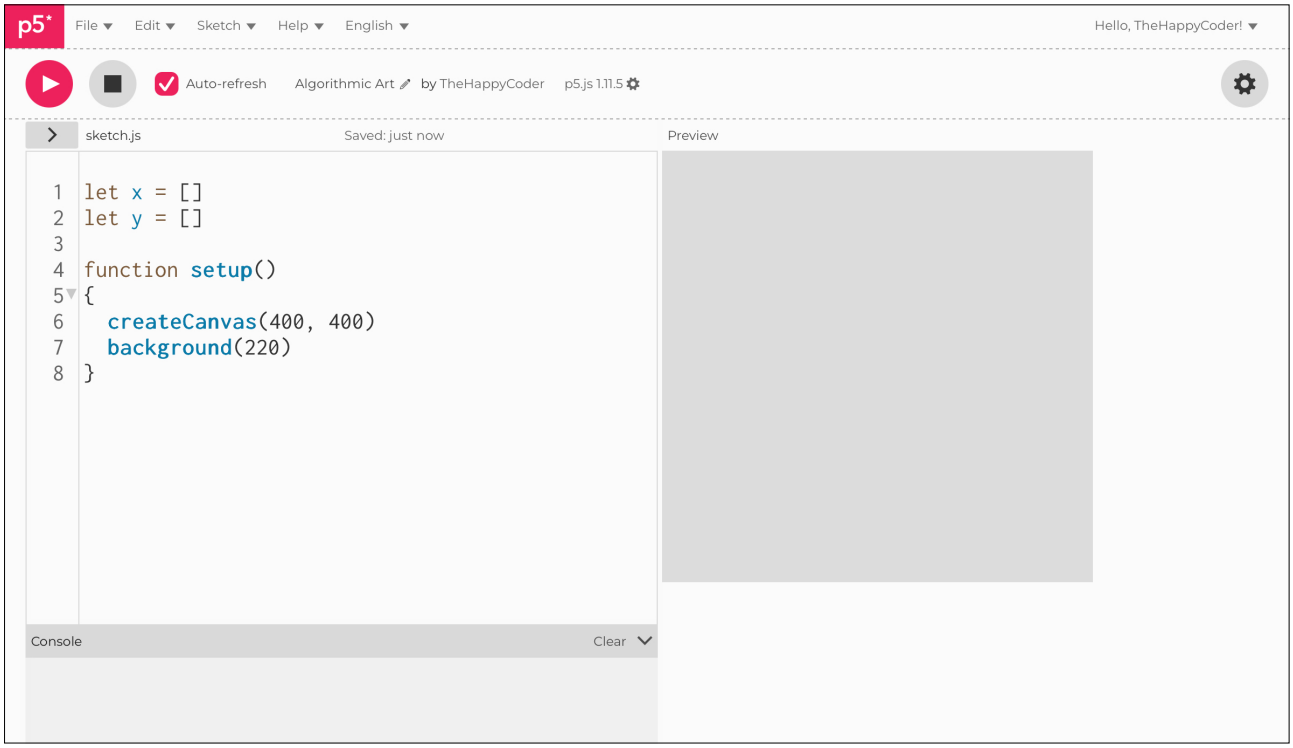
Notes

Empty arrays created.

Code Explanation

`let x = []`	An empty x array.
`let y = []`	An empty y array.

Figure B7.9



Sketch B7.10 filling the array

We can fill the array with random values; we will give each array ten numbers.

```
let x = []
let y = []

function setup()
{
  createCanvas(400, 400)
  background(220)
  for (let i = 0; i < 10; i++)
  {
    x[i] = random(width)
    y[i] = random(height)
  }
}
```

Notes

This fills each array with ten random numbers.

Code Explanation

`x[i] = random(width)`	Ten random numbers between 0 and 400 stored in the x array at each index i.
`y[i] = random(height)`	Ten random numbers between 0 and 400 stored in the y array at each index i.

Sketch B7.11 drawing the circles

We can now draw these circles. Every time you run the sketch, it generates a new set of ten circles.

```
let x = []
let y = []

function setup()
{
  createCanvas(400, 400)
  background(220)
  for (let i = 0; i < 10; i++)
  {
    x[i] = random(width)
    y[i] = random(height)
  }
  for (let i = 0; i < x.length; i++)
  {
    circle(x[i], y[i], 20)
  }
}
```

Notes

This uses the length of the `x` array, `x.length`, so it is critical that the `y` array is either the same length or longer. If you have two arrays that you need to check, then you would need to build in some code to check and compare lengths and always use the smaller of the two.

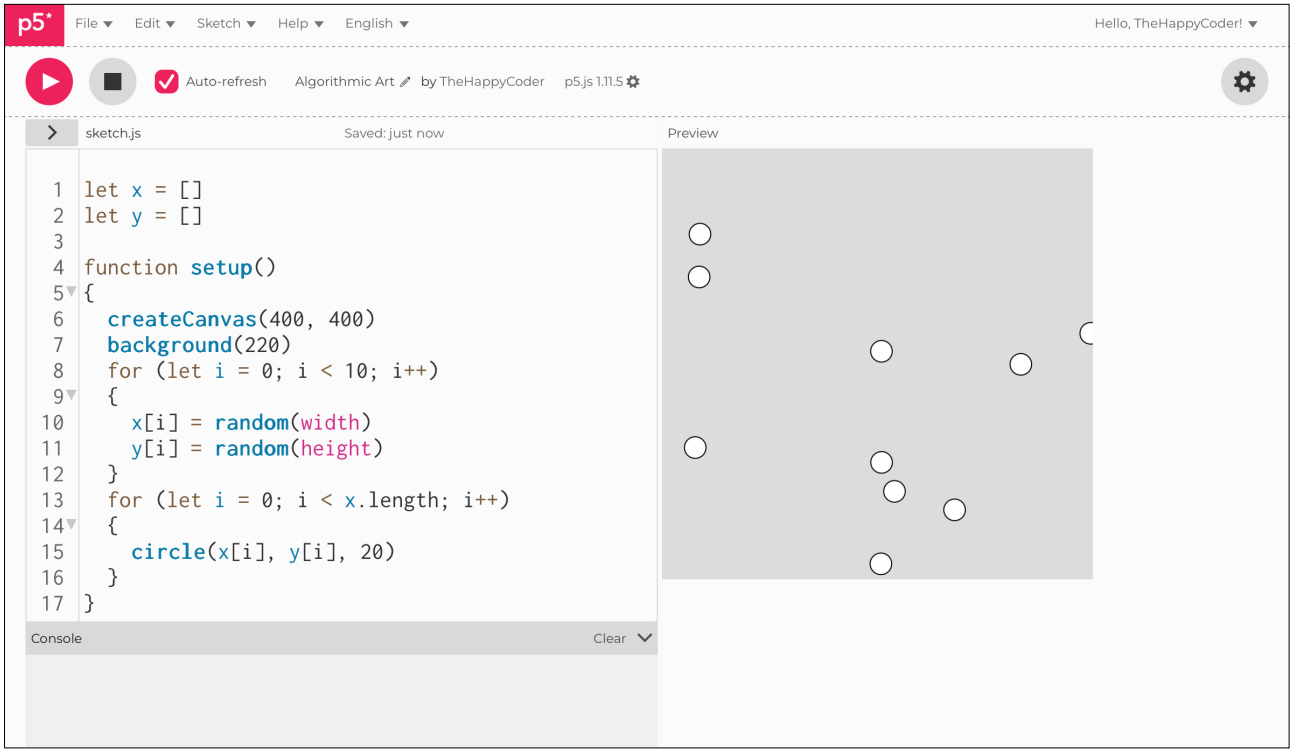
Challenge

Could you devise a way of checking the length of each array and comparing them?

Code Explanation

`for (let i = 0; i < x.length; i++)`	The x.length means that it stops when it gets to the end of the x array.
`circle(x[i], y[i], 20)`	Looks at each array (x and y) and pulls each value in turn to collect the coordinates of the circles.

Figure B7.11



Sketch B7.12 clicking the mouse

! Start a new sketch

We can use a function called `mousePressed()`. It is a function that waits for the mouse to be clicked (pressed) and will execute any code you give it. Here we simply draw a circle every time we click on the canvas.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
}
```

```
function mousePressed()
{
  let x = mouseX
  let y = mouseY
  circle(x, y, 20)
}
```

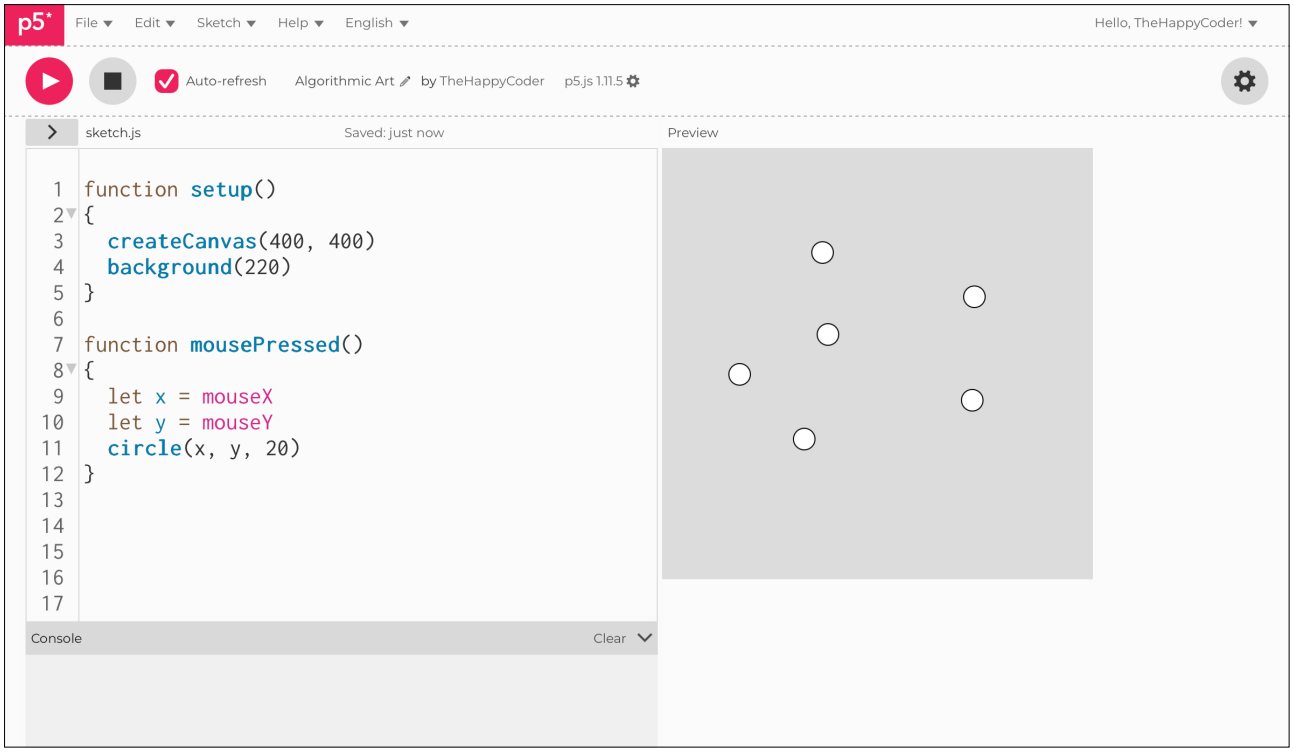
Notes

You get a circle every time you click on the canvas.

Code Explanation

`function mousePressed()`	This function is only executed when the mouse is pressed.

Figure B7.12



Sketch B7.13 pushing the array

Instead, we can store the positions of the circles (their `x` and `y` coordinates) when we click on the canvas. The function now pushes the `x` and `y` coordinates into an array called `bubbles` using the `push()` function.

```
let bubbles = []

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function mousePressed()
{
  let x = mouseX
  let y = mouseY
  circle(x, y, 20)
  bubbles.push(x, y)
}
```

Notes

The `push()` function does just that, pushing elements into an array.

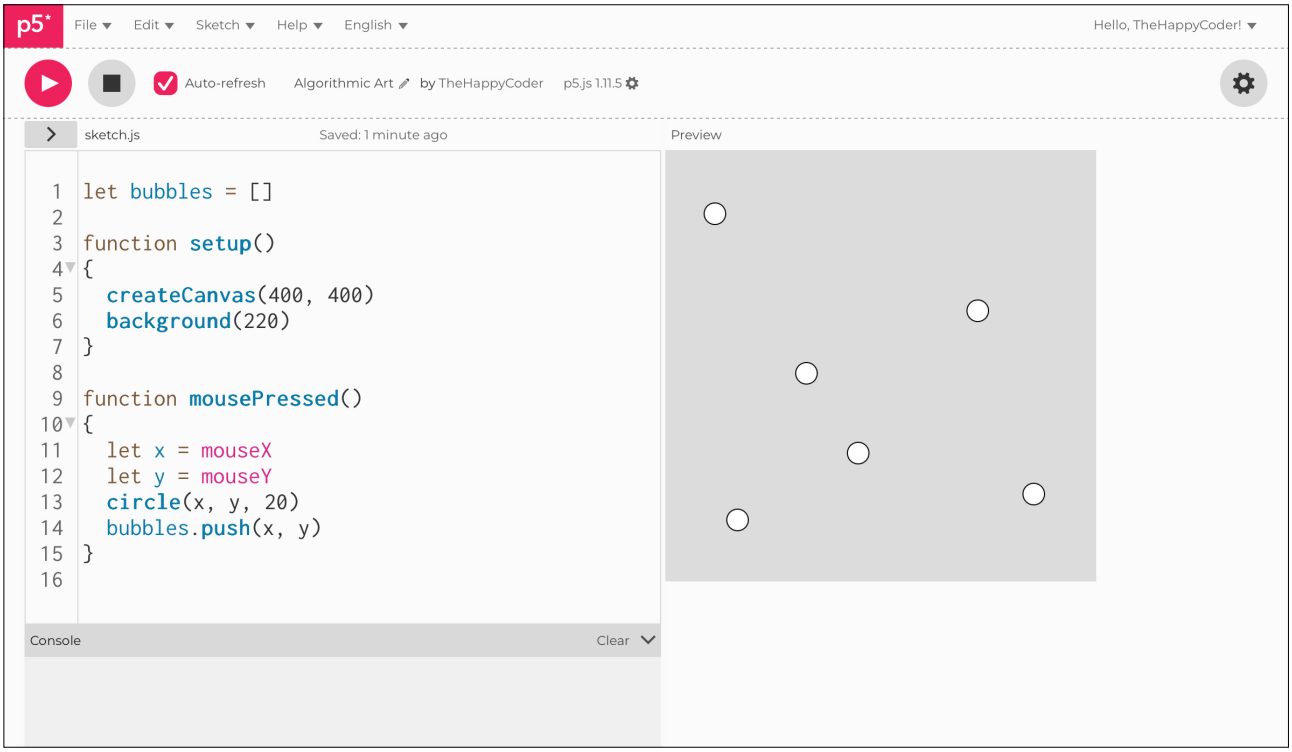
Challenge

Could we push a third element, for instance, the diameter?

Code Explanation

`bubbles.push(x, y)`	This pushes two elements into a single array called bubbles.

Figure B7.13



Sketch B7.14 console log

We can see inside the array by using something called `console.log()`. This sends information to the console.

```
let bubbles = []

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function mousePressed()
{
  let x = mouseX
  let y = mouseY
  circle(x, y, 20)
  bubbles.push(x, y)
  console.log(bubbles)
}
```

Notes

You can see the results in the console below the code panel. When you click once, you get two elements in the array: the first circle's (bubble) x and y coordinates, then when you click a second time, you get another pair of coordinates for the second circle, and so on.

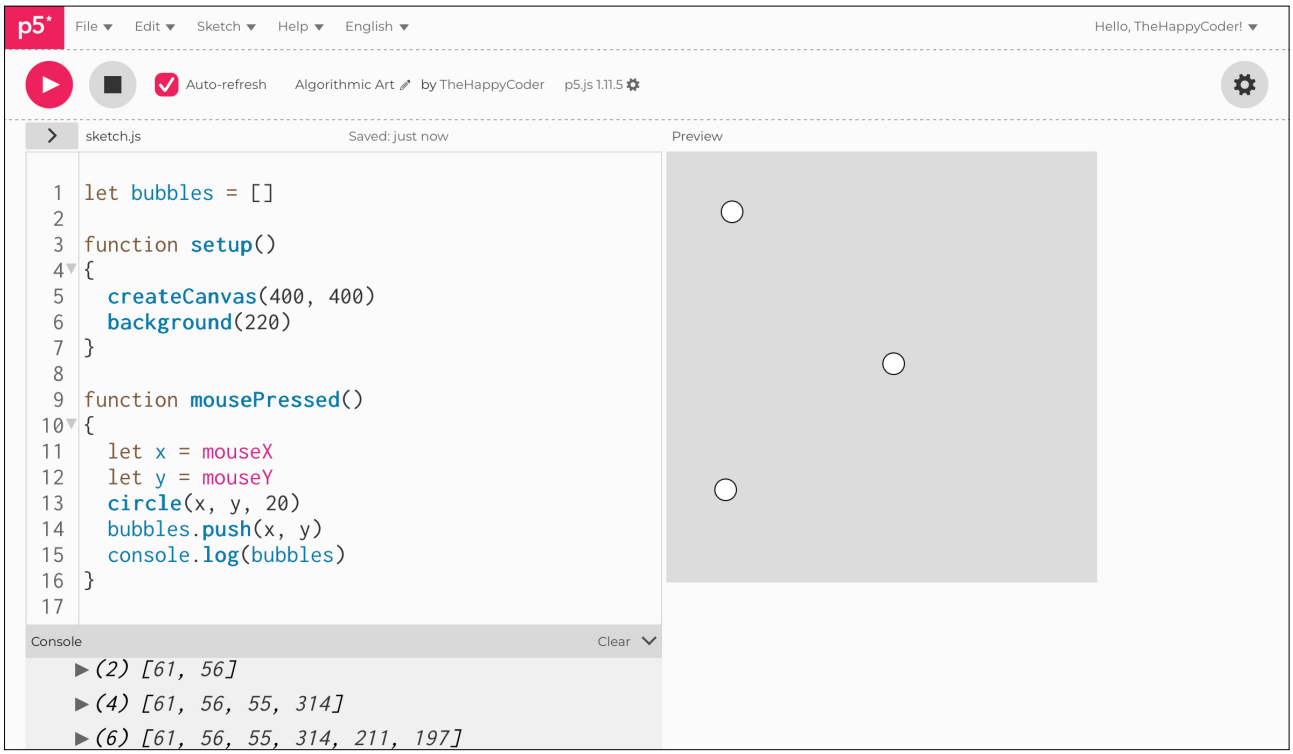
Challenge

Try `console.log(bubbles.length)`.

Code Explanation

`let bubbles = []`	We define an empty array.
`let x = mouseX`	Our x value is the mouseX position when we click.
`let y = mouseY`	Our y value is the mouseY position when we click.
`bubbles.push(x, y)`	The x and y values are pushed into the bubbles array.
`console.log(bubbles)`	We can see inside the bubble array.

Figure B7.14



Adding and Splicing

Arrays can often be pre-populated and are therefore quite static. But you can also add elements to them with new strings or integers, or even change an element. Also, you are able to remove an element or add a whole new database to the array. This makes it a powerful tool if you want to use data that is dynamic.

You can also start with an empty array and fill it with newly created data. Useful when creating an array or database with new data.

Sketch B7.15 appending using concat

! A new sketch

The `concat()` function allows you to join two arrays together to make one. We can see inside the array to check if they have been combined.

```
let colours1 = ['red', 'green', 'blue']
let colours2 = ['yellow', 'orange', 'purple']
let colours3 = []

function setup()
{
  createCanvas(400, 400)
  background(220)
  colours3 = colours1.concat(colours2)
  console.log(colours3)
}
```

Notes

This works with numbers and with objects.

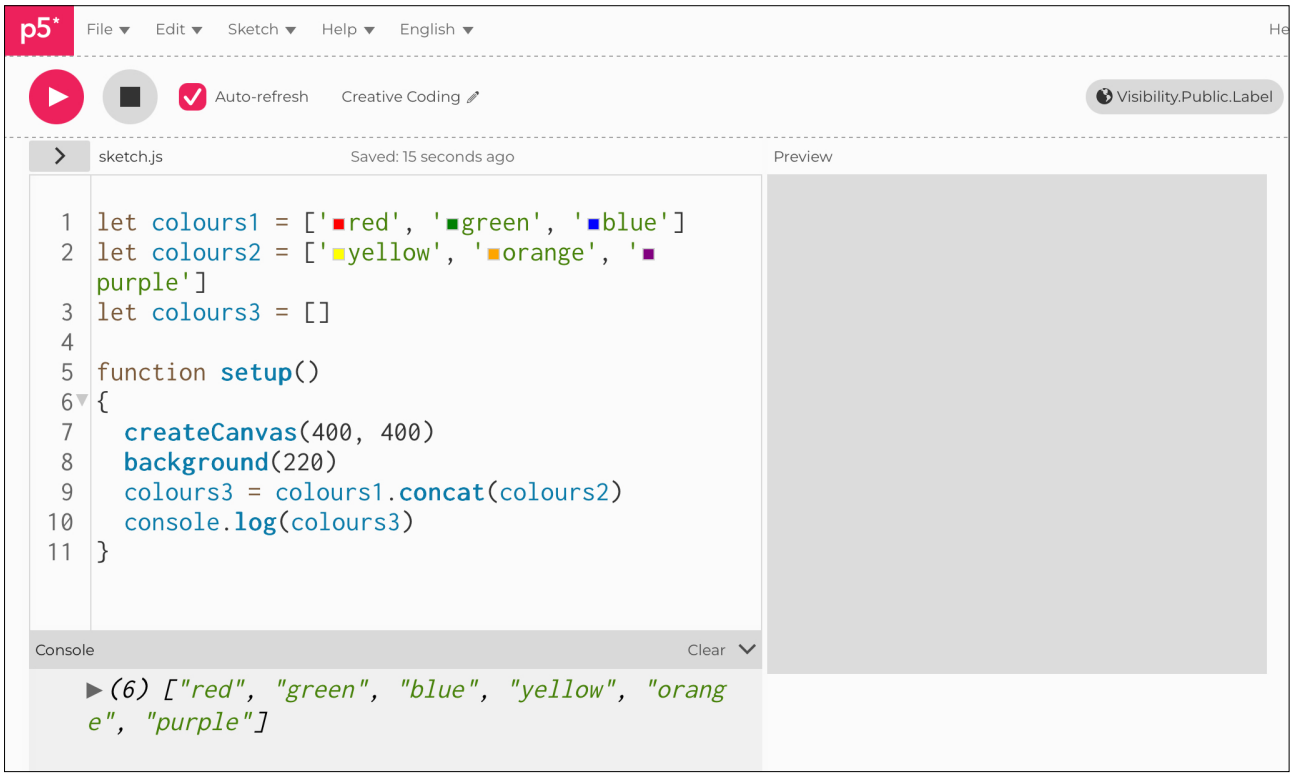
Challenge

Try with three arrays. Does it work?

Code Explanation

`colours3 = colours1.concat(colours2)`	Adds two (colours1 and colours2) arrays together to form one new one (colours3).

Figure B7.15



Sketch B7.16 adding by splicing

! A newish sketch

You can add new elements to the array using `splice()`. Here we add an extra colour (yellow) at the end. There are three arguments to the `splice()` function. The first argument (`3`) tells you where you want to add the extra element, the second argument (`0`) means you are adding it, and the third argument (`'yellow'`) is what you are adding.

```
let colours = ['red', 'green', 'blue']


function setup()
{
  createCanvas(400, 400)
  background(220)
  colours.splice(3, 0, 'yellow')
  console.log(colours)
}
```

Notes

You get an extra element added to the array. If you want to add something to the end of the array, giving it a hardcoded index position is not going to work if the array keeps growing. A better way is shown in the next sketch.

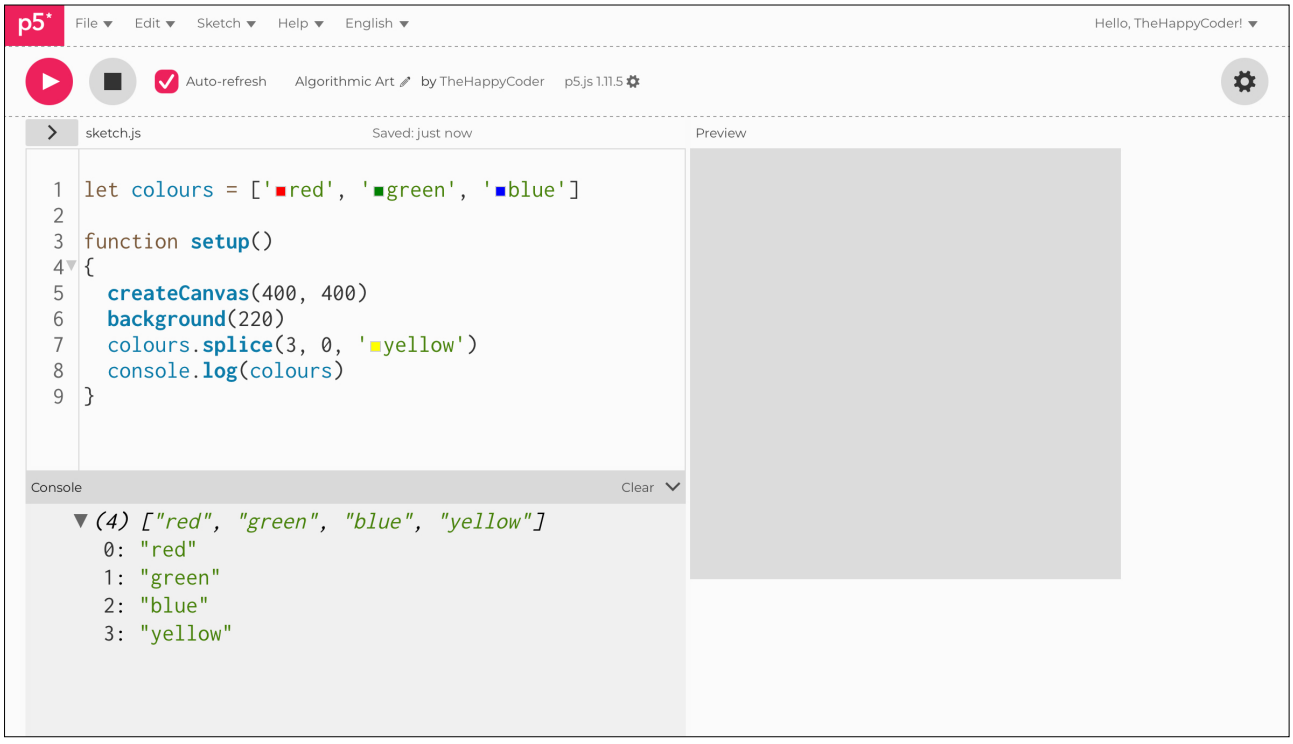
Challenges

1. Put 'yellow' in other positions in the array.
2. What happens if you give it a position, say, 13?

Code Explanation

`colours.splice(3, 0, 'yellow')`	Putting yellow at index[3], which is the fourth (end) element in the array.

Figure B7.16



Sketch B7.17 a better way

We can use the length of the array as our way of keeping track of the number of elements in the array.

```
let colours = ['red', 'green', 'blue']


function setup()
{
  createCanvas(400, 400)
  background(220)
  colours.splice(colours.length, 0, 'yellow')
  console.log(colours)
}
```

Notes

This is a stronger way of adding to an array.

Code Explanation

`colours.splice(colours.length, 0, 'yellow')`	Adds to the end of the array, whatever the length.

Sketch B7.18 in a different place

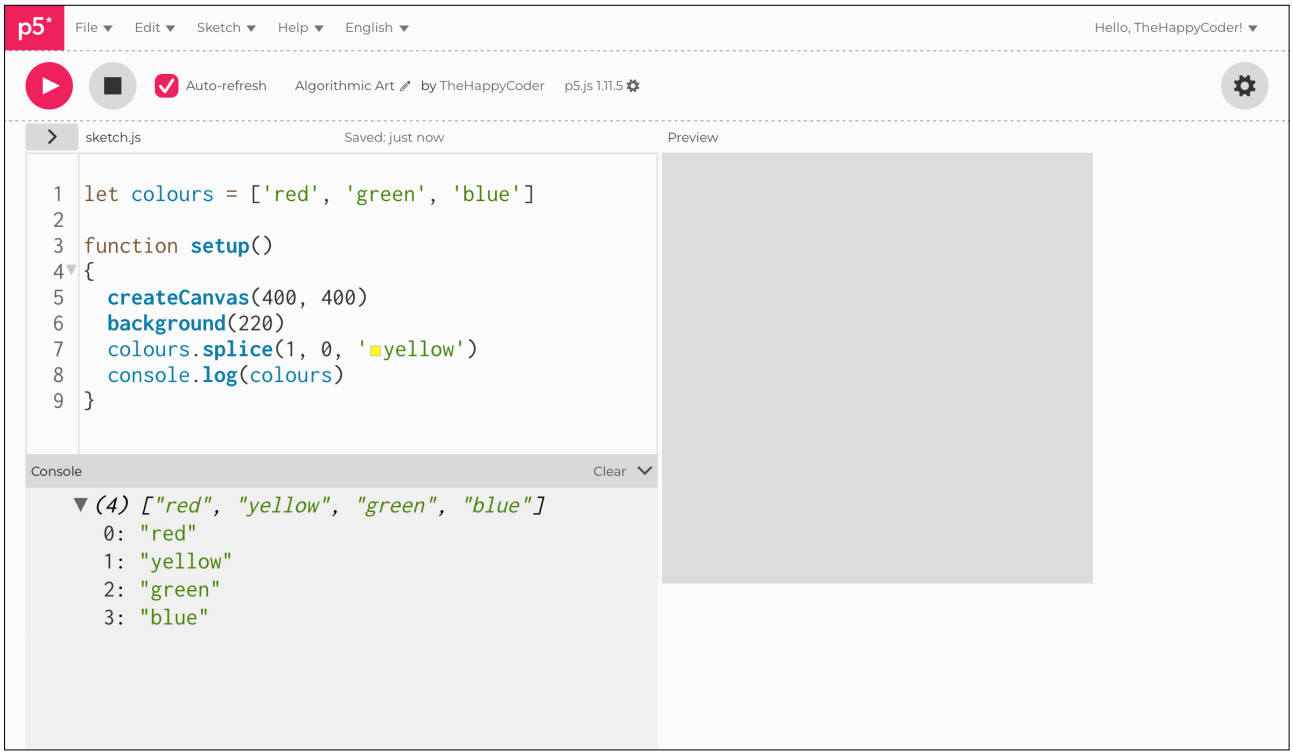
This time we place the yellow in spot `index [1]`, just in case you haven't already tried.

```
let colours = ['red', 'green', 'blue']


function setup()
{
  createCanvas(400, 400)
  background(220)
  colours.splice(1, 0, 'yellow')
  console.log(colours)
}
```

Notes

It is now sandwiched between the red and the green.

Figure B7.18



Sketch B7.19 replacing an element using splice

As well as adding, we can replace an element in an array. We do this by replacing the 0 with a 1 as the second argument.

```
let colours = ['red', 'green', 'blue']


function setup()
{
  createCanvas(400, 400)
  background(220)
  colours.splice(1, 1, 'yellow')
  console.log(colours)
}
```

Notes

The first argument tells you where to put it. It replaces the green with yellow.

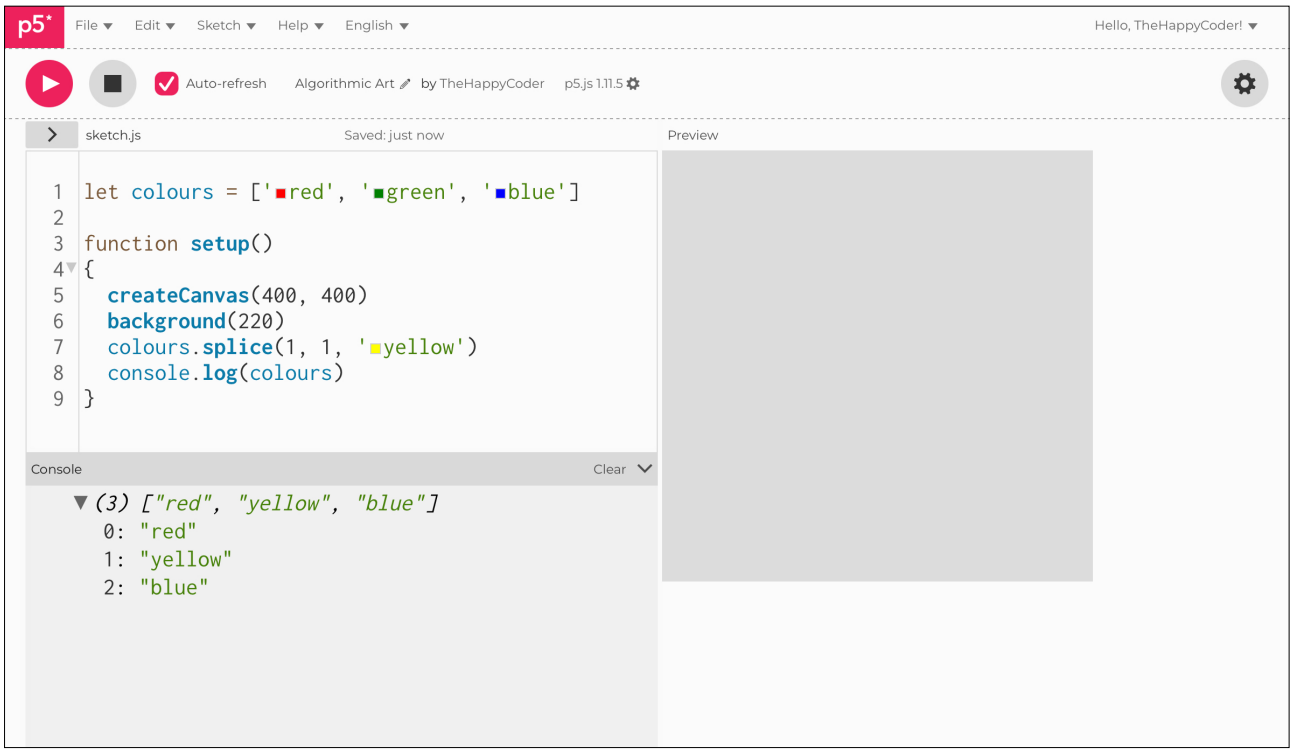
Challenge

Replace other colours.

Code Explanation

`colours.splice(1, 1, 'yellow')`	Replaces the 'green' with 'yellow'.

Figure B7.19



Sketch B7.20 deleting an element using splice

You can delete an element from an array; here we delete two elements starting at index 1.

```
let colours = ['red', 'green', 'blue']


function setup()
{
  createCanvas(400, 400)
  background(220)
  colours.splice(1, 2)
  console.log(colours)
}
```

Notes

We have removed both the green and blue colour elements from the array.

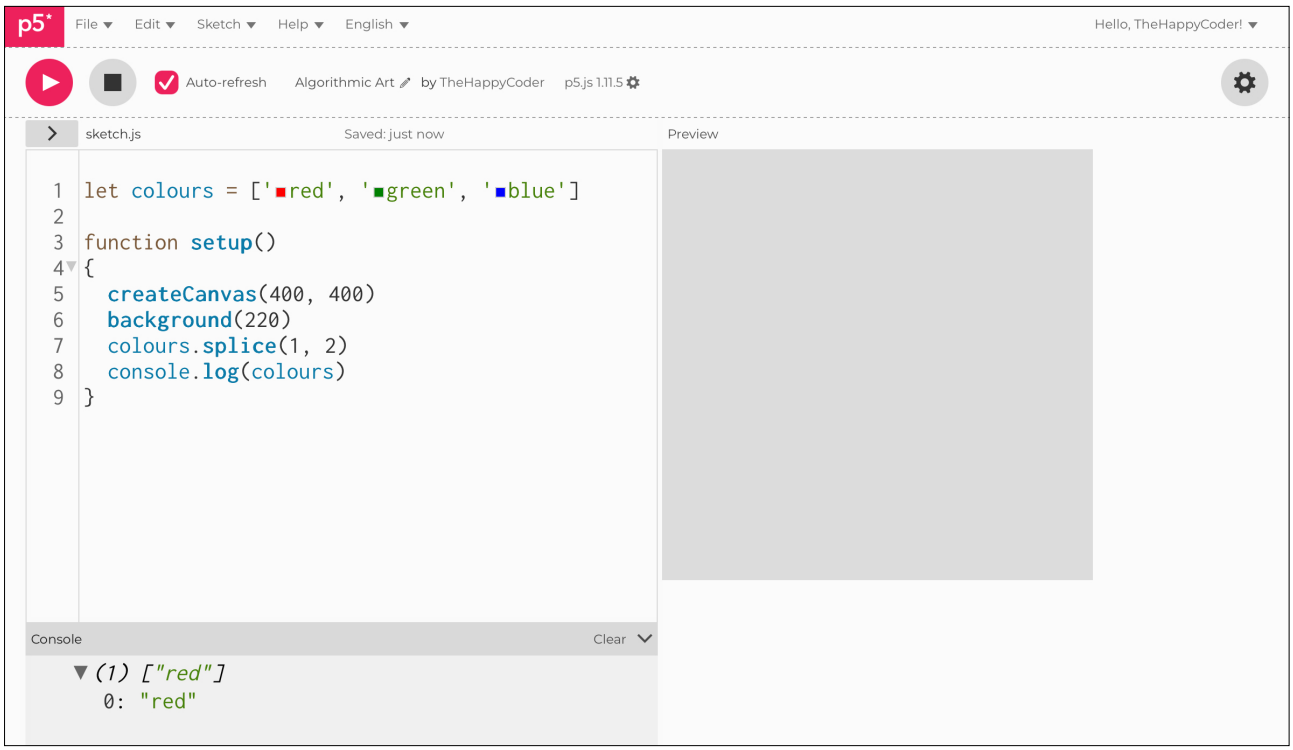
Challenge

Try: `colours.splice(0, 3)`. You should get an empty array.

Code Explanation

`colours.splice(1, 2)`	Deleting two elements starting at index [1].

Figure B7.20



Sketch B7.21 an array of objects

! Start a new sketch and call the array `data` for a change (mousePressed not draw!)

Another way to use an array is to have objects. In this example, an object is one circle. Each object has an `x` and `y` value; they are recognisable as objects because of the colons (`:`). We will use `console.log(data)` as before to see the difference. Each object is called `inputs` to collect the `x` and `y` co-ordinates when you click on the canvas.

```
let data = []

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  circle(inputs.x, inputs.y, 30)
  data.push(inputs)
  console.log(data)
}
```

Notes

We use `console.log(data)` to see inside the array. It gives you a list of objects, each one representing a circle. Inside each object are the `x` and `y` inputs. Click on the arrow to reveal the contents of the array (fig. 1.21b). Please note that in version 2 you will get a float not an integer.

Challenge

Change the variable names of `x` and `y` to, say, `flower` and `animal`. See where else you have to change them.

Code Explanation

`let inputs = {...}`	We give the collective name for the x and y values as inputs.
`x: mouseX`	For the x value, we use a colon, then the input value or variable.
`y: mouseY`	For the y value, we use a colon, then the input value or variable.
`circle(inputs.x, inputs.y, 50)`	The x co-ordinate of the object (inputs) is inputs.x, repeated for the y component.

Figure B7.21a

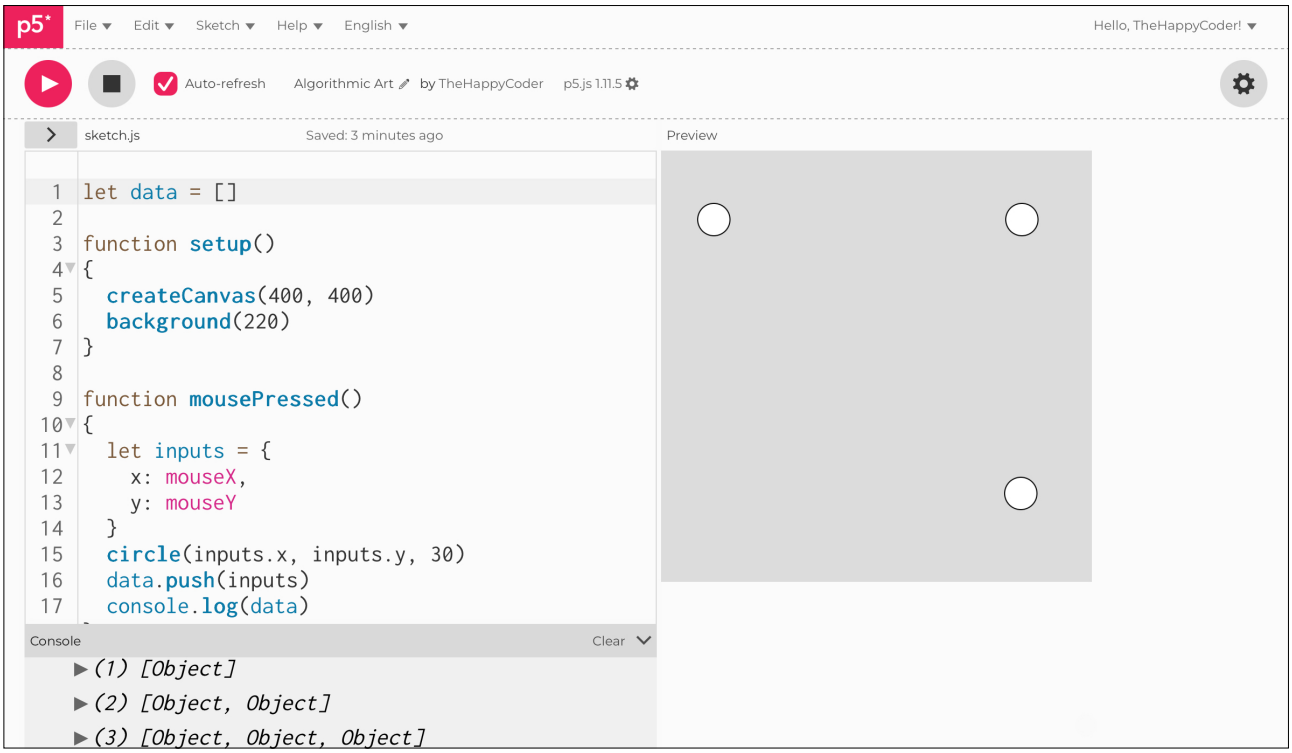


Figure B7.21b: opening the object in the console

The Joy of Coding Algorithmic Art Module B Unit #8 irregular shapes

Module B Unit #8: irregular shapes

It is one thing to draw regular shapes using the functions circle, square, rectangle, and triangle, etc., but what if you want to draw something that isn't a regular shape? Then you can use the `vertex()` function to draw any shape. This is great for drawing random patterns or shapes and manipulating them. The `bezier()` function allows us to draw curves.

We start with `beginShape()` and add vertex co-ordinates for `x` and `y`, and then add `endShape()` at the end.

There are many types; here are two:

```
vertex()
bezier()
```

Vertex Lines

A vertex is a point in space; here we are using it in 2D, but you can also do this in 3D (next module). It needs the x and y coordinates. You can have as many vertices as you would like. To draw the lines between the vertices, you need to use the functions (commands) `beginShape()` and `endShape()`.

Sketch B8.1 drawing a vertex line

We are going to start very simply by drawing a line. We specify the end coordinates of the line, and it draws a line between them; there is no line function. Although this seems a little protracted, it has great value when you use the coordinates in creative ways, drawing complex shapes and patterns.

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

function draw()
{
  background(220)
  beginShape()
  vertex(100, 100)
  vertex(300, 300)
  endShape()
}
```

Notes

Using the `vertex()` function is a useful tool to draw irregular polygons, in other words, shapes other than rectangles and triangles. A vertex simply draws a dot (pixel) at the co-ordinate given. The `beginShape()` and `endShape()` functions join the dots up. In this example, to get you started, two dots are used, so it gives you one line.

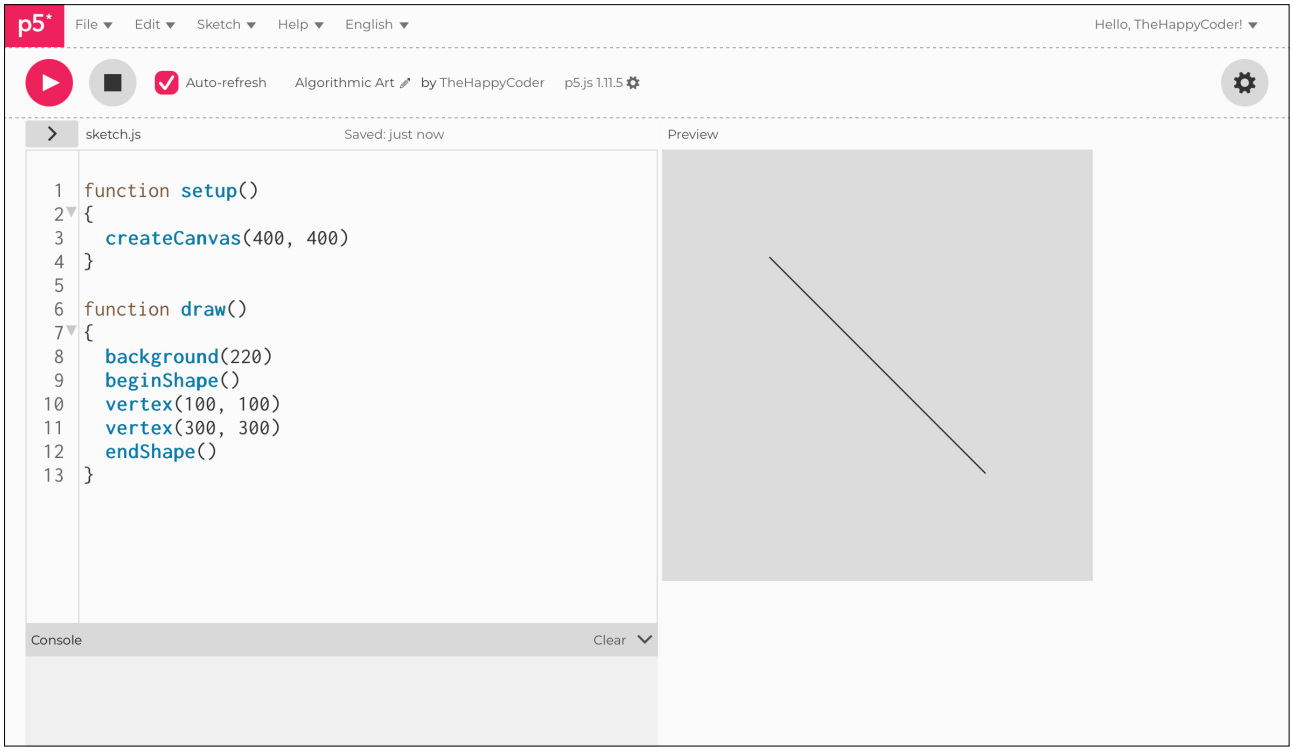
Challenge

Try it without the `beginShape()` and `endShape()`.

Code Explanation

`beginShape()`	Starts adding vertices to an irregular shape.
`vertex(100, 100)`	Defines a vertex with x and y coordinates.
`endShape()`	Stops adding vertices to an irregular shape.

Figure B8.1



Sketch B8.2 drawing a square using vertex

Drawing a simple square using the vertex coordinates, the order is critical. It draws a line from one vertex to the next one, in order, on the list. So don't get them mixed up. Notice that we have five vertices, whereas a square should have only four; do you know why?

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

function draw()
{
  background(220)
  beginShape()
  vertex(100, 100)
  vertex(300, 100)
  vertex(300, 300)
  vertex(100, 300)
  vertex(100, 100)
  endShape()
}
```

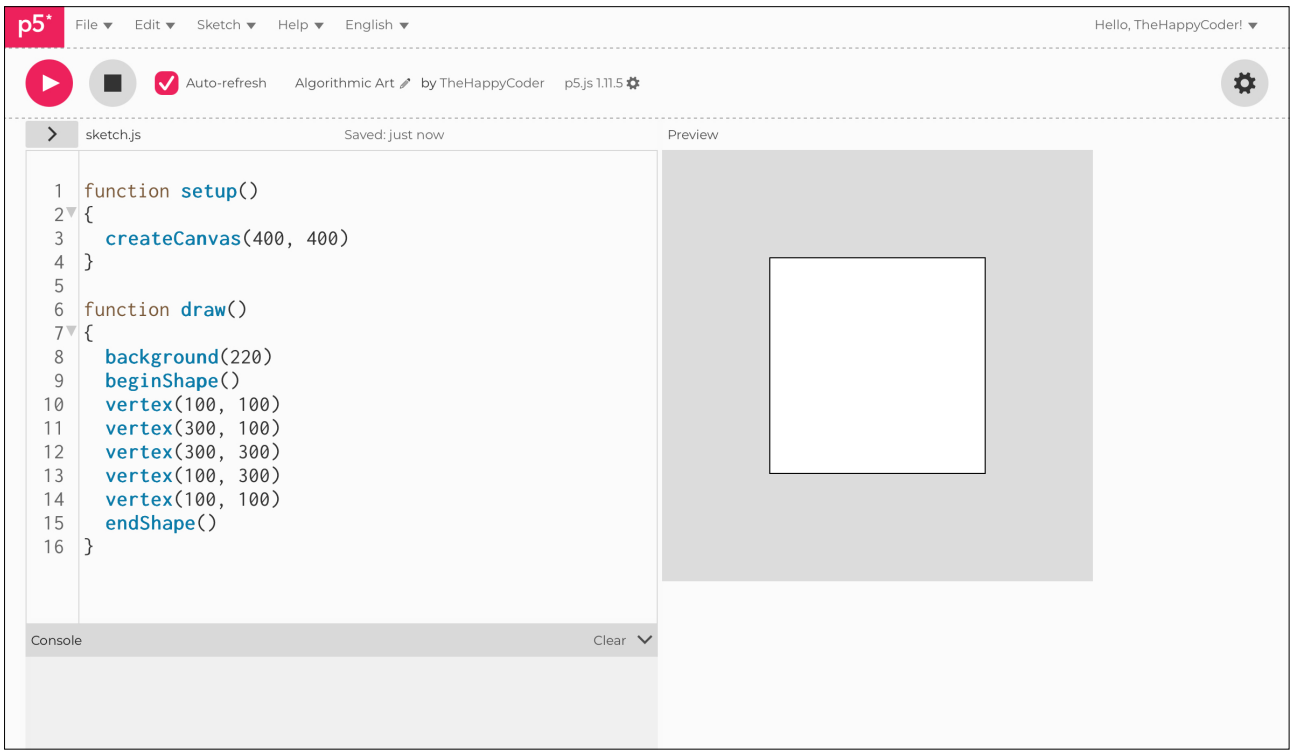
Notes

Now to draw a square using more dots and therefore more lines. For this to work, you need to close the shape by drawing the last dot where the first one was drawn. In the next example, there is a shortcut to this. Notice that it fills the shape automatically.

Challenges

1. Reorder the vertices and see what happens.
2. Try to draw a couple of rectangles.

Figure B8.2



Sketch B8.3 that was CLOSE

! Remove the last vertex point

This joins up the first and last vertices automatically. We can remove the last vertex so that we only have four and add the CLOSE command. Although we have four vertices for the square, it doesn't draw the final line until you add the word CLOSE to the endShape().

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

function draw()
{
  background(220)
  beginShape()
  vertex(100, 100)
  vertex(300, 100)
  vertex(300, 300)
  vertex(100, 300)
  endShape(CLOSE)
}
```

Notes

Using `endShape(CLOSE)` means you don't have to draw the last dot to join it all up. Notice it is in capital letters.

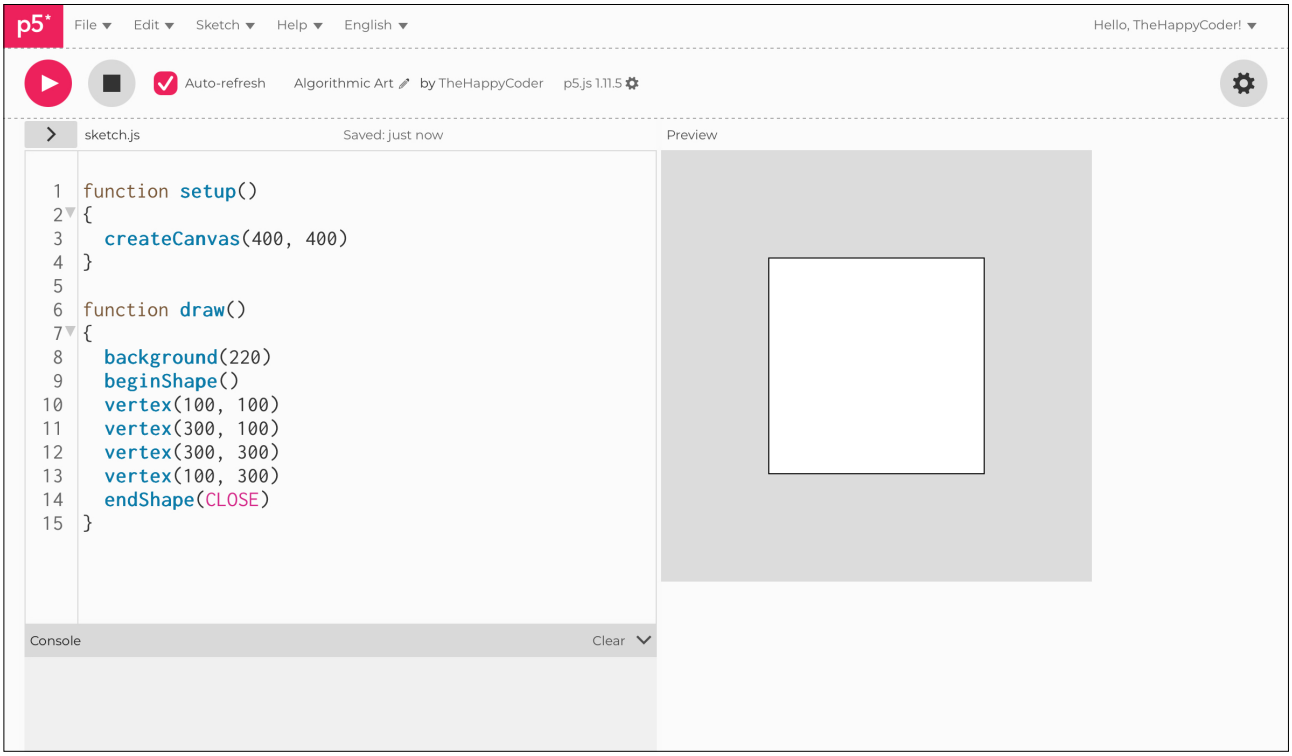
Challenge

Try the above without the `CLOSE` argument and see the difference.

Code Explanation

`endShape(CLOSE)`	Draws the final line between the first and last vertices.

Figure B8.3



Sketch B8.4 making a more irregular shape

Adding in a couple of vertices.

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

function draw()
{
  background(220)
  beginShape()
  vertex(100, 100)
  vertex(300, 100)
  vertex(200, 150)
  vertex(300, 300)
  vertex(100, 300)
  vertex(200, 250)
  endShape(CLOSE)
}
```

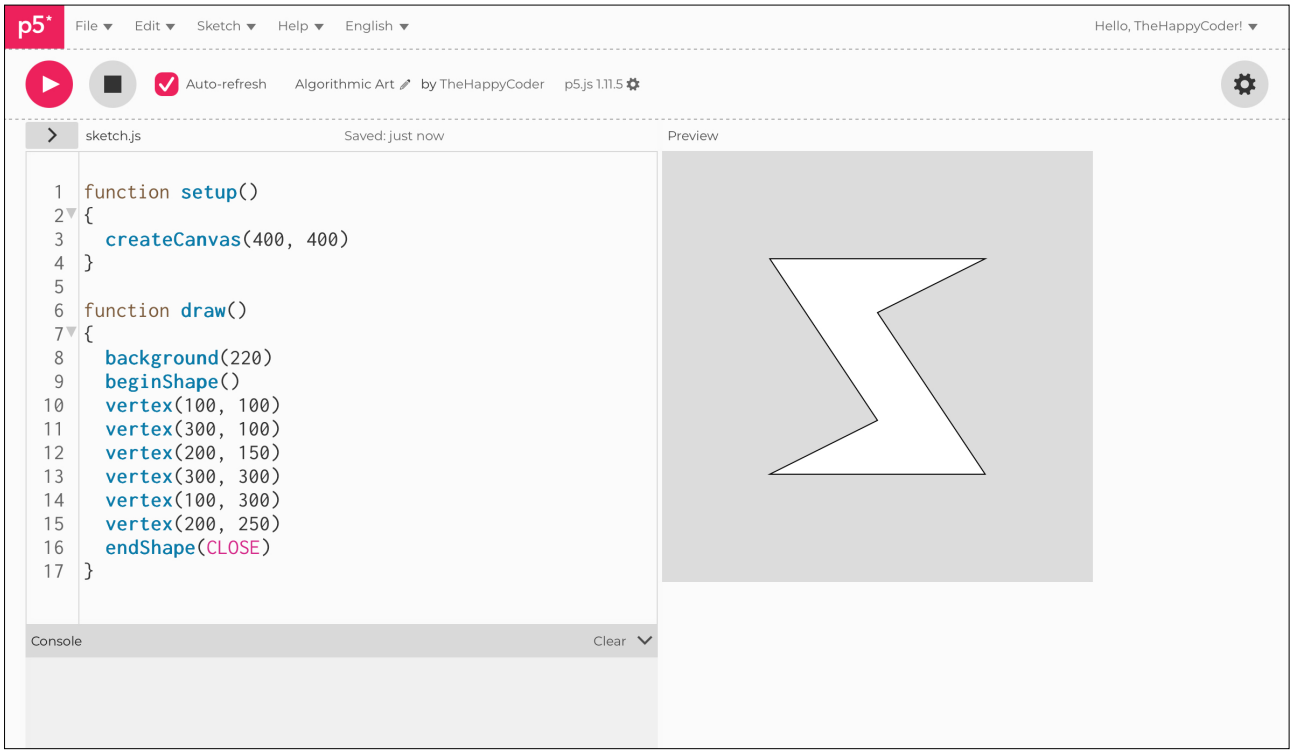
Notes

A slightly more interesting shape.

Challenge

Make your own shape, a star perhaps!

Figure B8.4



Sketch B8.5 translating

We will now translate to the centre of the canvas. There is going to be some reorganising of the coordinates, as you might expect. Have a go yourself first and see if you can do it. Using pen and paper can help to work out the new coordinates.

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

function draw()
{
  background(220)
  translate(width/2, height/2)
  beginShape()
  vertex(-100, -100)
  vertex(100, -100)
  vertex(0, -50)
  vertex(100, 100)
  vertex(-100, 100)
  vertex(0, 50)
  endShape(CLOSE)
}
```

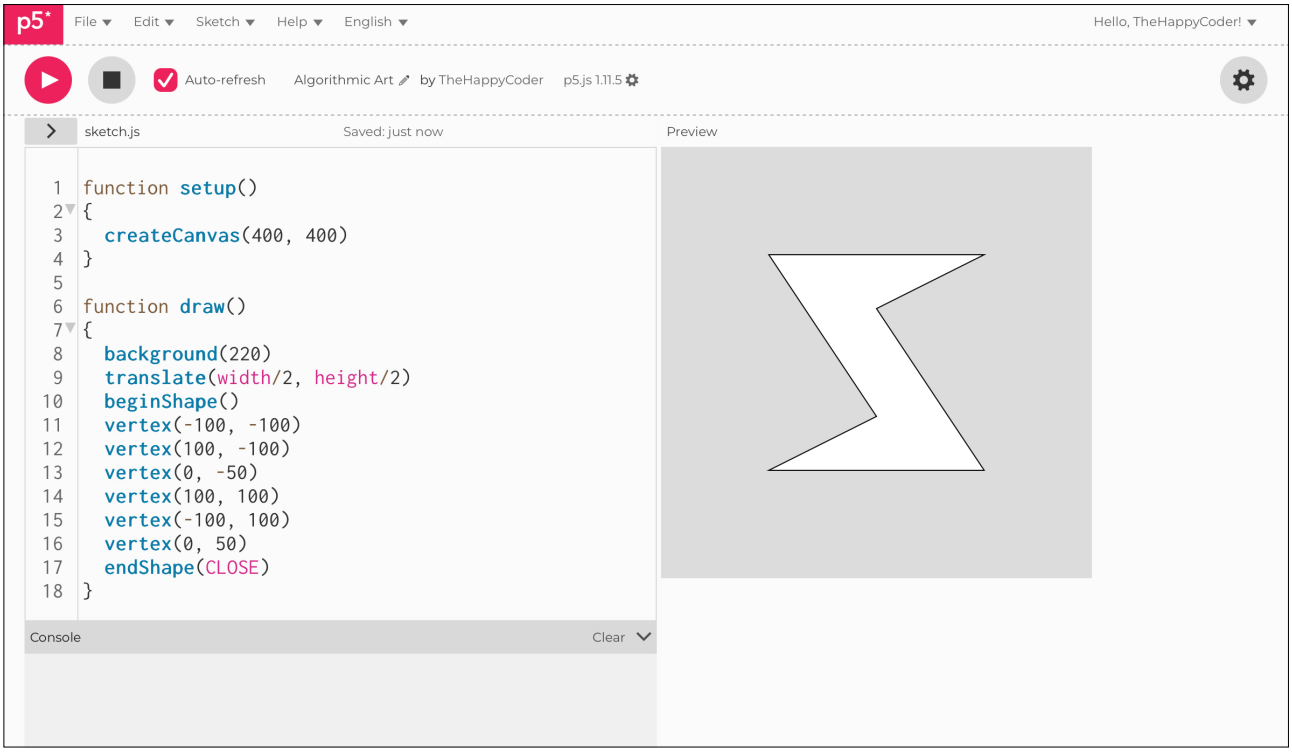
Notes

We just subtracted 200.

Challenge

Create a variable to subtract.

Figure B8.5



Sketch B8.6 rotating

We can now rotate the shape.

```
let angle = 0

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

function draw()
{
  background(220)
  translate(width/2, height/2)
  rotate(angle)
  beginShape()
  vertex(-100, -100)
  vertex(100, -100)
  vertex(0, -50)
  vertex(100, 100)
  vertex(-100, 100)
  vertex(0, 50)
  endShape(CLOSE)
  angle += 0.05
}
```

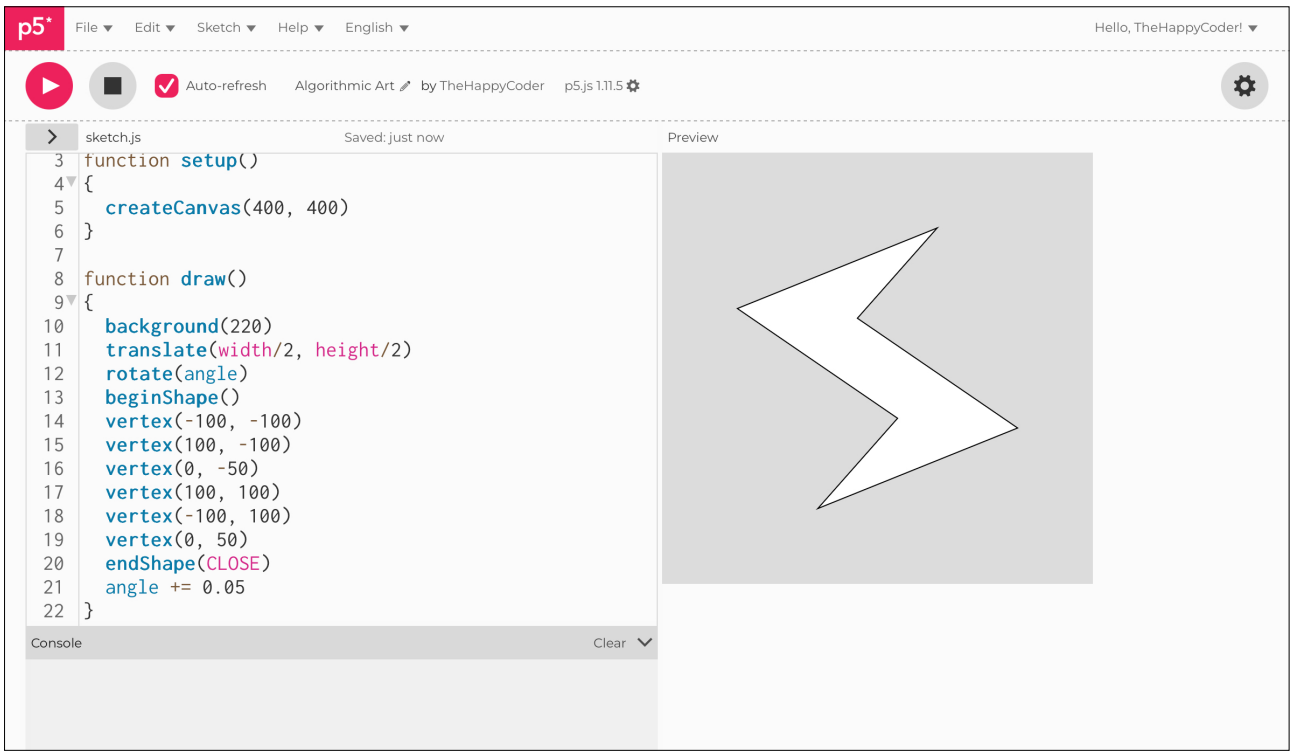
Notes

It rotates about the centre of the canvas; also, we are using radians for the angle.

Challenge

Change to degrees.

Figure B8.6



Sketch B8.7 many sided shape

! Starting a new sketch

Using a `for()` loop and the `x` and `y` coordinates for a circle, we can draw any-sided shape we want. Here, we are drawing a `10-sided` shape.

```
let radius = 150
let sides = 10
let x = 0
let y = 0

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background(220)
  translate(width/2, height/2)
  beginShape()
  for (let angle = 0; angle < 360; angle += 360/sides)
  {
    x = radius * sin(angle)
    y = radius * cos(angle)
    vertex(x, y)
  }
  endShape(CLOSE)
}
```

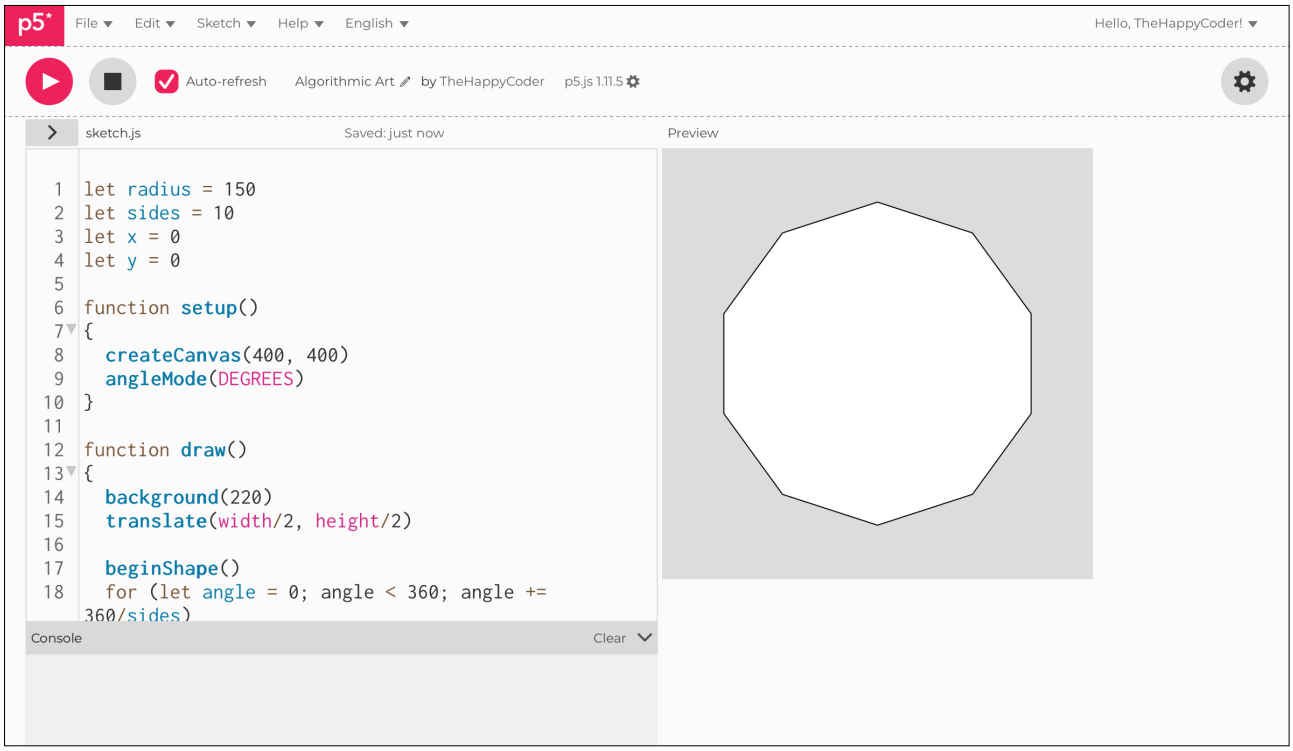
Notes

We specify the number of sides and divide 360 by the number of sides we want.

Challenges

1. Try other numbers of sides.
2. What happens if you put in 4.5 rather than a whole number?
3. Have a slider to change the number of sides.

Figure B8.7



Sketch B8.8 a vertex doodle

! New sketch

A bit of a doodle.

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

function draw()
{
  background(255)
  beginShape()
  for (let y = 50; y < 350; y += 5)
  {
    strokeWeight(random(2))
    for (let x = 50; x < 355; x += 5)
    {
      vertex(x, y)
      y += random(-2, 2)
      fill(random(50), 10)
      endShape()
    }
    beginShape()
  }
  endShape()
  noLoop()
}
```

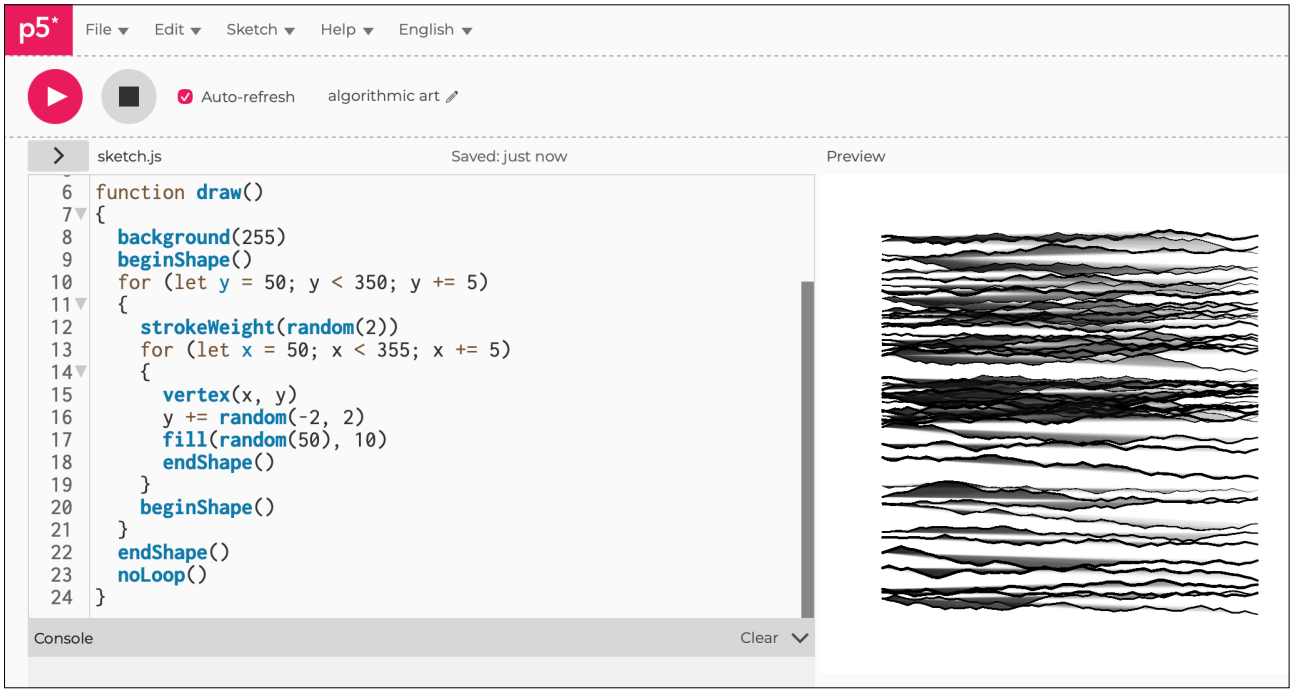
Notes

Just using some of the lessons we have learned. This is a matter of playing and experimenting.

Challenges

1. Play with the values, try colours; HSB might be nice.
2. Animated seascape?

Figure B8.8



Bezier Curves

Not only can we draw straight lines with a `vertex()` function, but we can also do curved lines with bezier. The `bezier()` function has four arguments. The first and last are the fixed end points, think `line()`. The middle two affect the line by drawing it towards those points. They don't join the line but pull it.

There is maths behind all of this, but at this stage, it is not necessary to understand it, just use it. If you want to know more, just search for it.

Sketch B8.9 a Bézier curve

! Starting a brand new sketch

We fix four points (pairs of co-ordinates) that serve as the arguments for the `bezier()` curve function.

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

function draw()
{
  background(220)
  noFill()
  strokeWeight(3)
  stroke('blue')
  let x1 = 50
  let y1 = 50
  let x2 = 50
  let y2 = 350
  let x3 = 350
  let y3 = 350
  let x4 = 350
  let y4 = 50
  bezier(x1, y1, x2, y2, x3, y3, x4, y4)
  strokeWeight(10)
  stroke('darkred')
  point(x1, y1)
  point(x2, y2)
  point(x3, y3)
  point(x4, y4)
}
```

Notes

The first and last are fixed, and the middle two cause the line to curve. I have coloured the points and the line so that you can see their positions and influence.

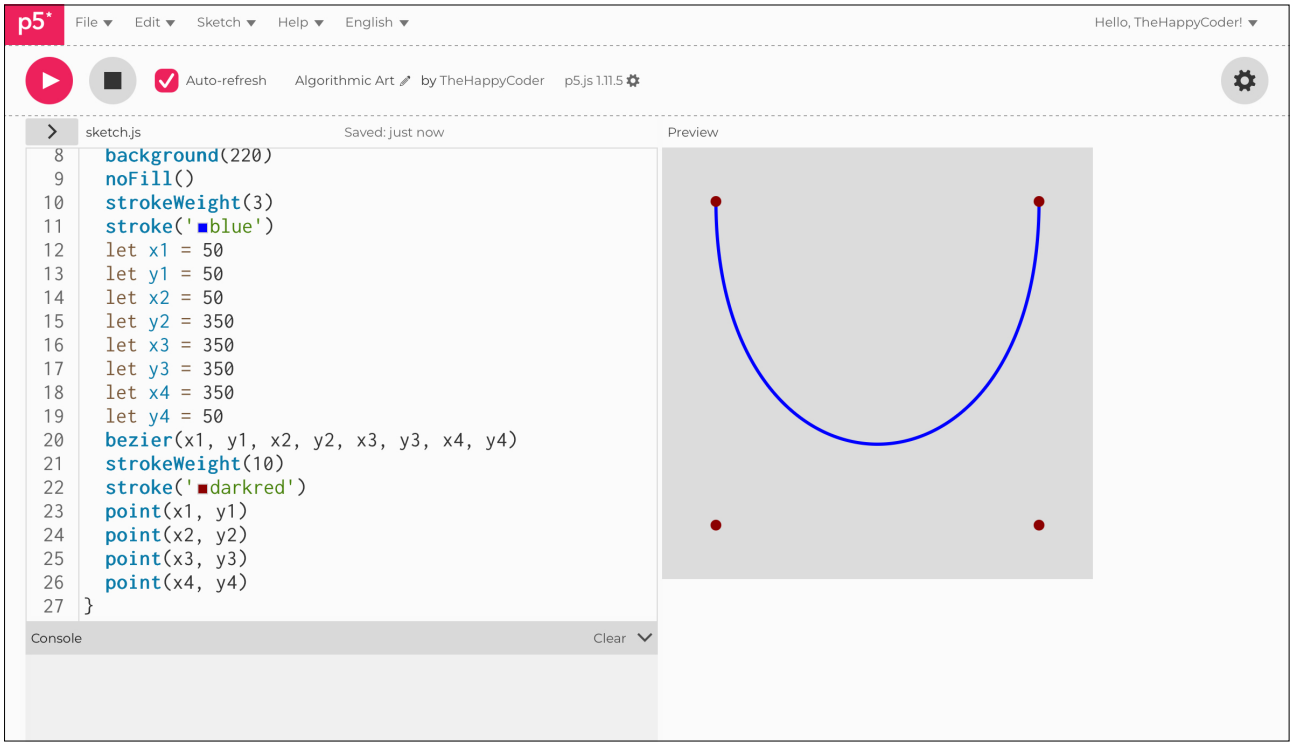
Challenge

Try different positions for the middle two points.

Code Explanation

`bezier(x1, y1, x2, y2, x3, y3, x4, y4)`	The bezier() curve function with the four sets of vertices.

Figure B8.9



Sketch B8.10 the dancing mouse

As you move your mouse, it affects the curve of the line. We have also changed the last y co-ordinate.

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

function draw()
{
  background(220)
  noFill()
  strokeWeight(3)
  stroke('blue')
  let x1 = 50
  let y1 = 50
  let x2 = 50
  let y2 = 350
  let x3 = mouseX
  let y3 = mouseY
  let x4 = 350
  let y4 = 350
  bezier(x1, y1, x2, y2, x3, y3, x4, y4)
  strokeWeight(10)
  stroke('darkred')
  point(x1, y1)
  point(x2, y2)
  point(x3, y3)
  point(x4, y4)
}
```

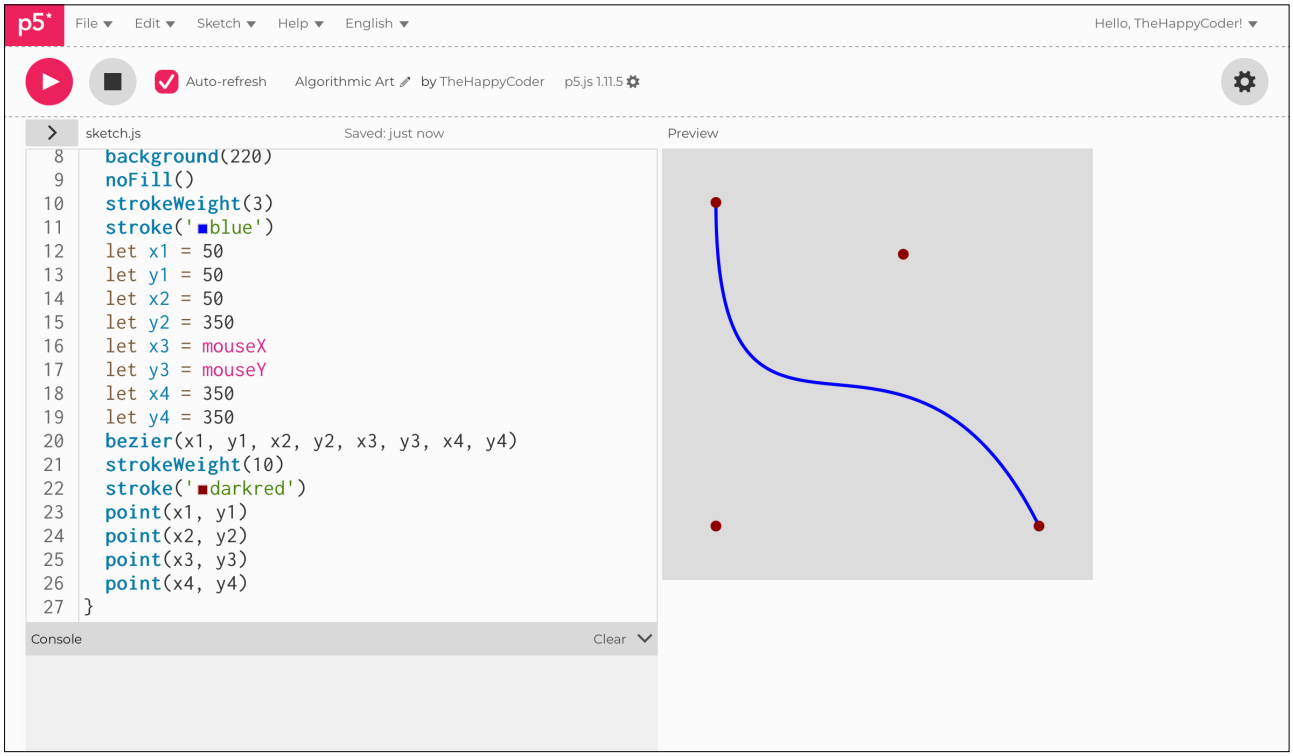
Notes

Gives you a sense of how one of the middle points may affect the curve.

Challenges

1. They'll see the other points and the difference.
2. Have both the middle points be controlled by the mouse.
3. You could code the values straight into the Bézier curve function.

Figure B8.10



Sketch B8.11 bezier doodle part 1

! New sketch

We are going to create a simple pattern.

```
function setup()
{
  createCanvas(400, 400)
  noFill()
}

function draw()
{
  background(220)
  bezier(50, 50, 50, 600, 600, 50, 50, 50)
}
```

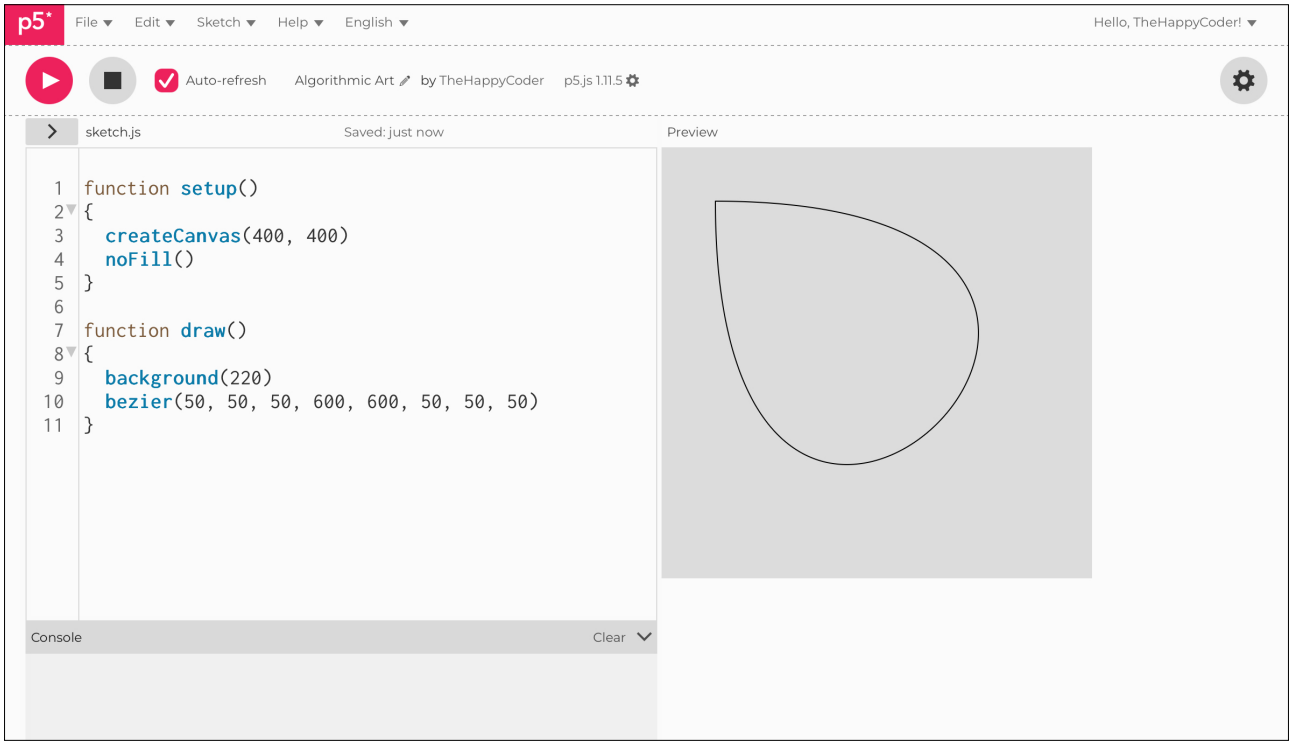
Notes

We will code the coordinates straight into the function.

Challenge

Remove the `noFill()`.

Figure B8.11



Sketch B8.12 bezier doodle part 2

We create a `for()` loop that increments in steps of `10`; this is added to the `bezier()` function, which is inside the loop.

```
function setup()
{
  createCanvas(400, 400)
  noFill()
}

function draw()
{
  background(220)
  for(let i = 0; i < 100; i += 10)
  {
  bezier(50 + i, 50 + i, 50 + i, 600 + i, 600 + i, 50 + i, 50 + i, 50 + i)
  }
  noLoop()
}
```

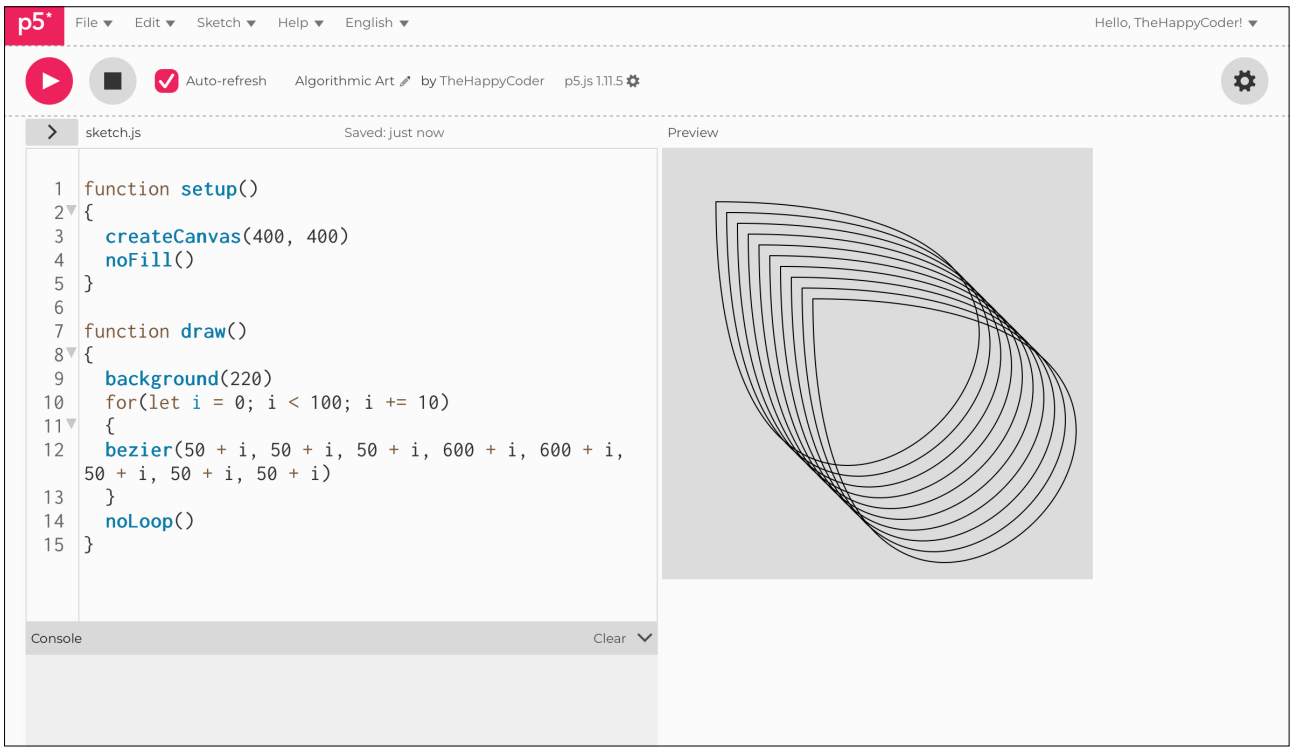
Notes

Creates a repeated pattern; the `noLoop()` isn't entirely necessary.

Challenge

Try something else.

Figure B8.12



Sketch B8.13 another doodle with bezier

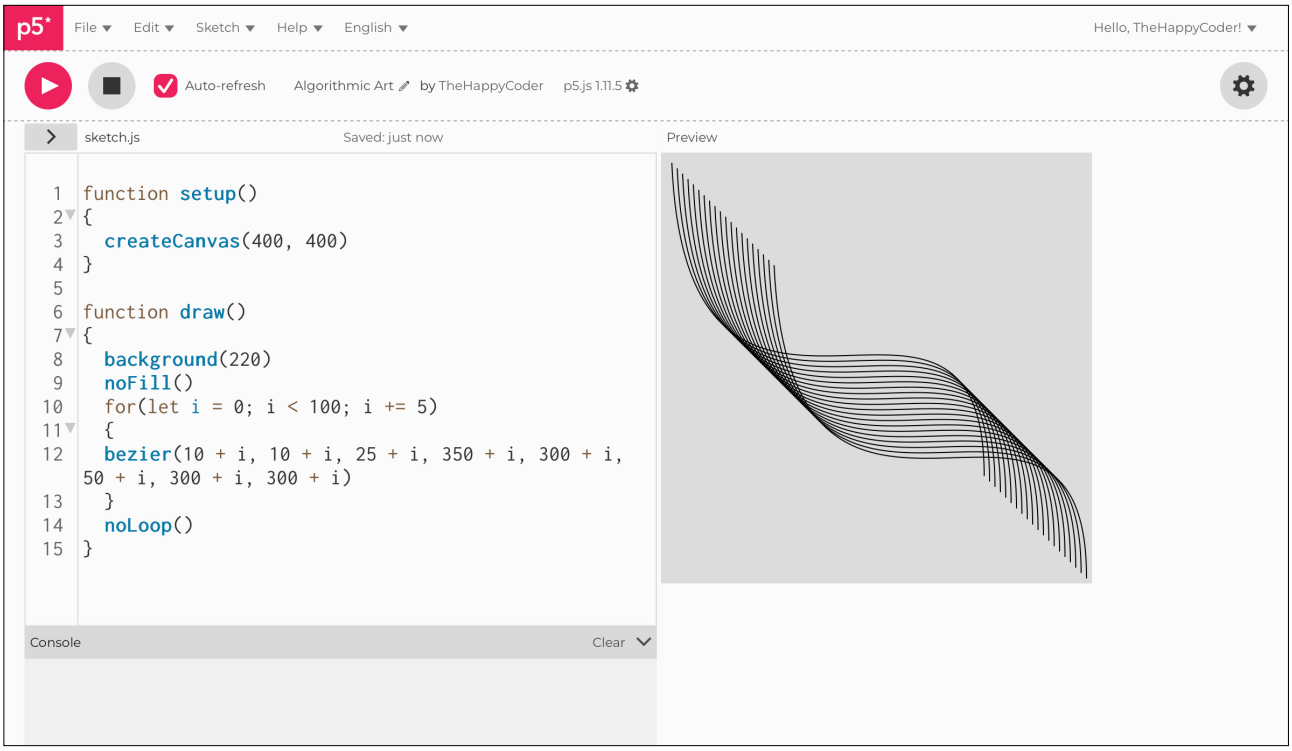
Changing this slightly, we can create the following pattern:

```
function setup()
{
  createCanvas(400, 400)
  noFill()
}

function draw()
{
  background(220)
  for (let i = 0; i < 100; i += 5)
  {
  bezier(10 + i, 10 + i, 25 + i, 350 + i, 300 + i, 50 + i, 300 + i, 300 + i)
  }
  noLoop()
}
```

Notes

Another nice pattern.

Figure B8.13



Sketch B8.14 lovely jubbly

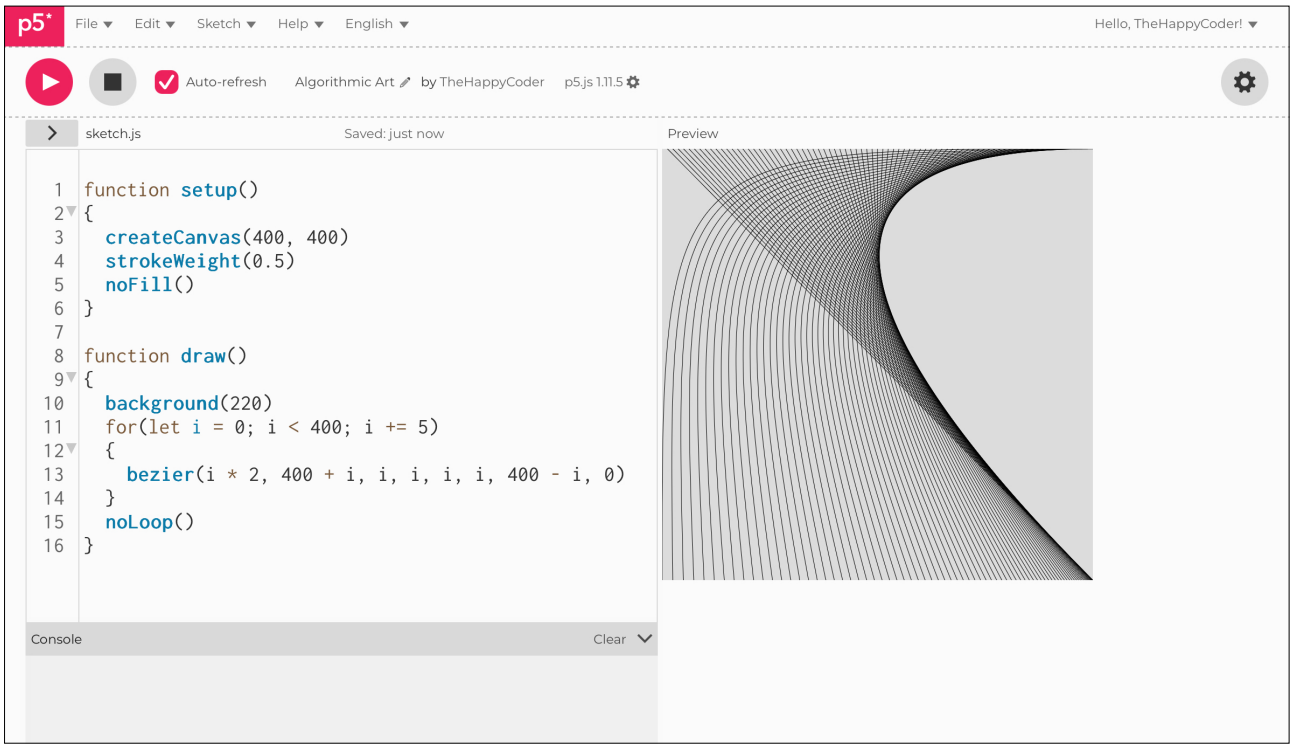
Another little doodle.

```
function setup()
{
  createCanvas(400, 400)
  strokeWeight(0.5)
  noFill()
}

function draw()
{
  background(220)
  for(let i = 0; i < 400; i += 5)
  {
    bezier(i * 2, 400 + i, i, i, i, i, 400 - i, 0)
  }
  noLoop()
}
```

Notes

Just have fun playing.

Figure B8.14



Sketch B8.15 a bit of a splash

Another (final) Bezier doodle.

```
function setup()
{
  createCanvas(400, 400)
  noFill()
  strokeWeight(3)
}

function draw()
{
  background(255)
  for(let i = 0; i < 100; i += 5)
  {
    let x1 = random(10, 390)
    let y1 = random(10, 390)

    let x2 = random(100, 200)
    let y2 = random(100, 200)

    let x3 = random(200, 300)
    let y3 = random(200, 300)

    let x4 = random(10, 390)
    let y4 = random(10, 390)
    stroke(random(255), 0, 0)
    bezier(x1, y1, x2, y2, x3, y3, x4, y4)
  }
  noLoop()
  rect(0, 0, width, height)
}
```

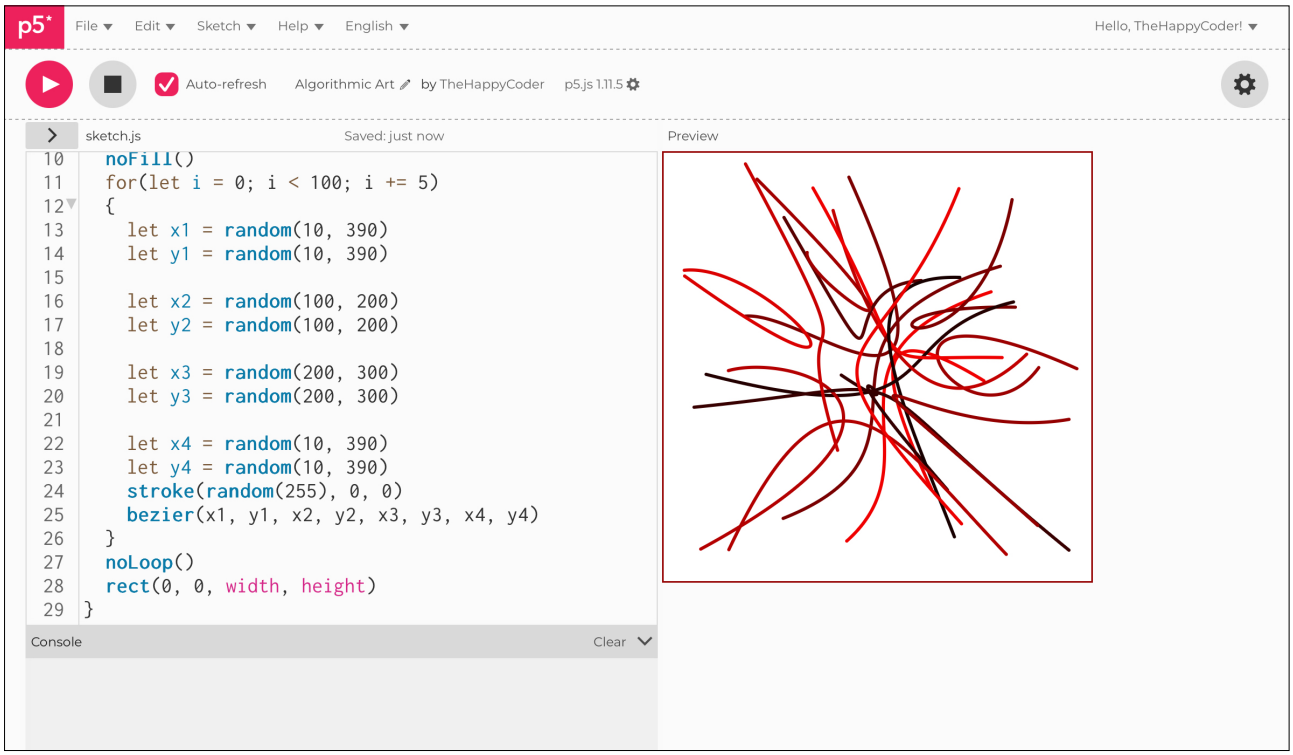
Notes

White background with a simple frame around it.

Challenge

Play with colours, alpha, strokeWeight(), etc.

Figure B8.15



The Joy of Coding Algorithmic Art Module B Unit #9 arcs and mapping

Module B Unit #9: arcs and mapping

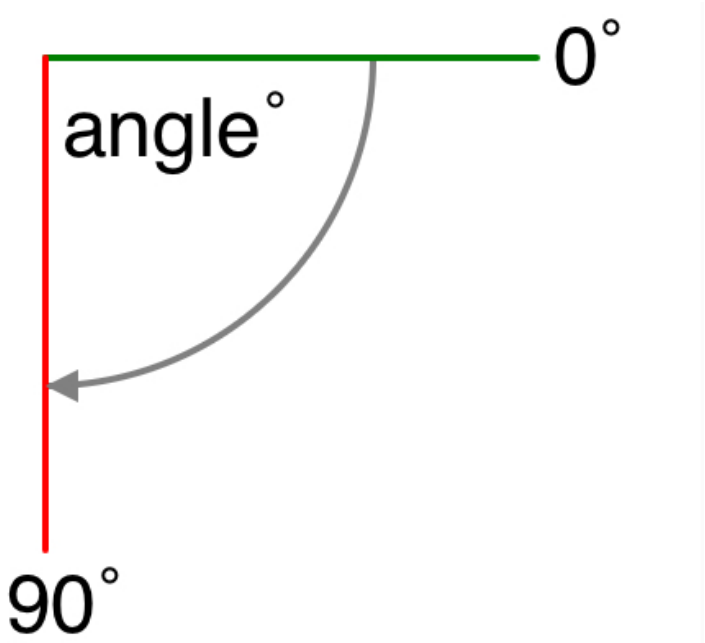
The arc() function has six arguments (three pairs), so if we consider this example:

```
arc(100, 200, 150, 300, 0, 180)
```

The first two arguments are the x and y coordinates (x is 100, y is 200).
The second two are the dimensions of the ellipse (150 wide, 300 tall).
The third pair of arguments are the angles from and to (starts at 0° and stops at 180°).

The angle is described clockwise from the horizontal 0°, to the new angle 90° (see Fig.1). It is measured in radians by default; hence, we will be using the angleMode() to change to degrees to make it more intuitive.

Figure 1: arc angle orientation



Key concepts:

arcs
ellipseMode()
mapping

Sketch B9.1 the 90° arc

A simple 90° arc.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  arc(200, 200, 200, 200, 0, 90)
}
```

Notes

Because the width and height are the same, it scribes a circle, or a quarter of one in this case.

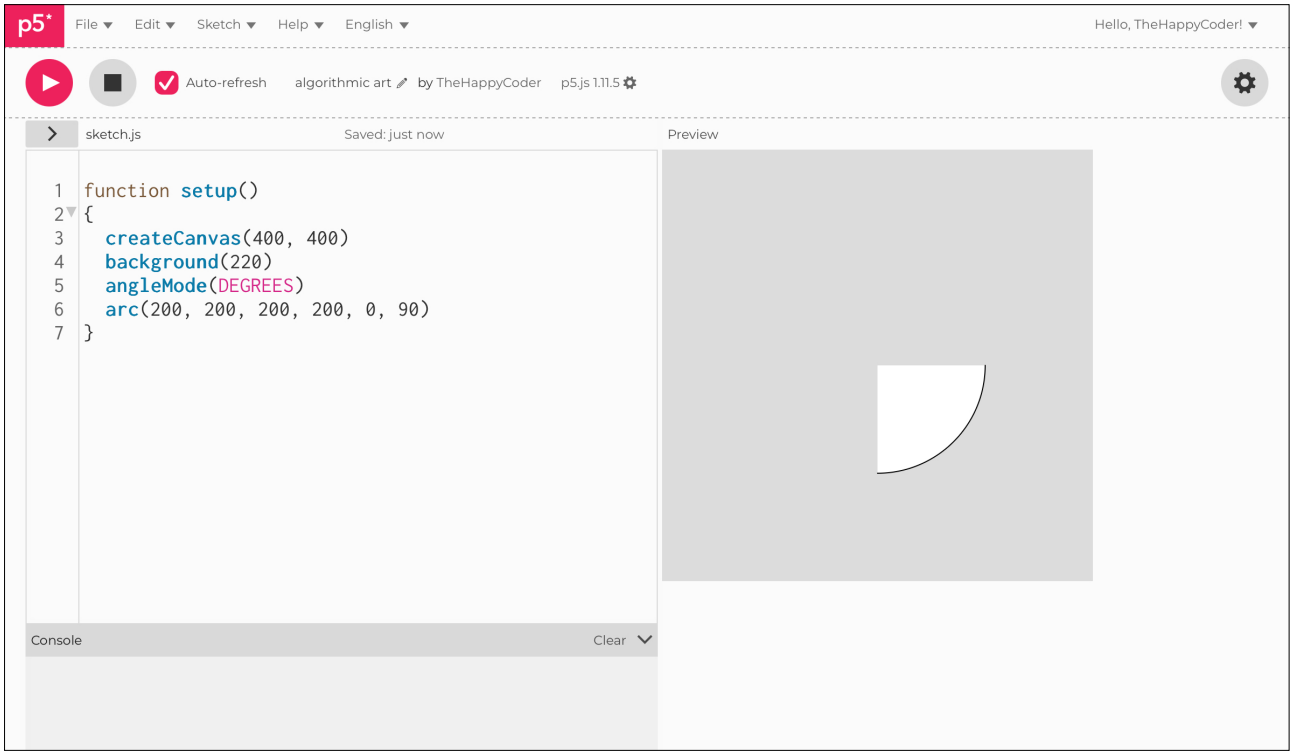
Challenge

Try other angles, starting and stopping.

Code Explanation

`arc(200, 200, 200, 200, 0, 90)`	Arc drawn at position (200, 200) with a width and height of 200, starting at 0° and finishing at 90°.

Figure B9.1



Sketch B9.2 the negative 90° arc

We can start anywhere; here we start at −90° and arc round to 0°.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  arc(200, 200, 200, 200, -90, 0)
}
```

Notes

A different way of using the angles.

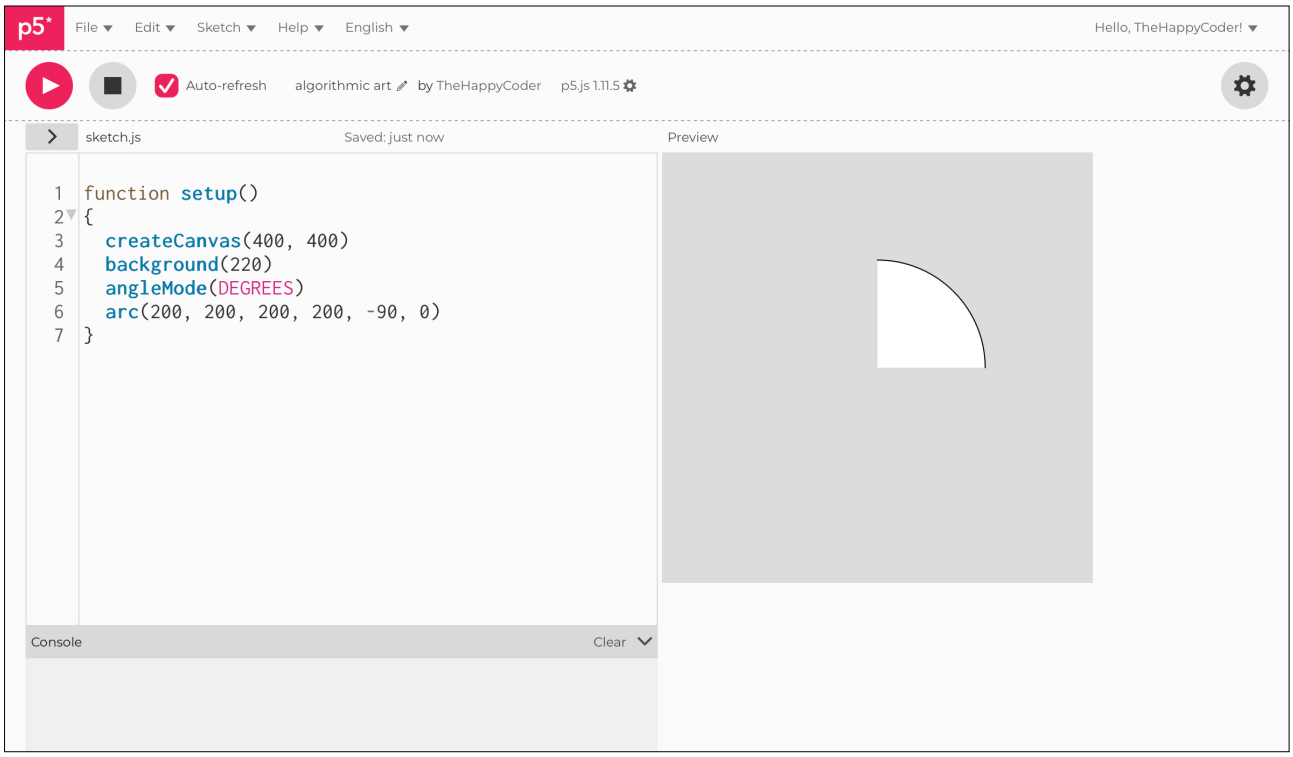
Challenge

Try other negative values.

Code Explanation

`arc(200, 200, 200, 200, -90, 0)`	Starts at −90° and moves round to 0° clockwise.

Figure B9.2



Sketch B9.3 drawing a complete circle

Although there are a number of easy ways to draw a circle, we can do the same with arcs; here we increase the final angle by one in a `for()` loop.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  for (let i = 0; i < 360; i++)
  {
    arc(200, 200, 200, 200, i - 1, i)
  }
}
```

Notes

We start with `i - 1` because we want to draw the arc at each increment; otherwise, we get to the end of the loop and just draw the final arc from `0°` to `360°`.

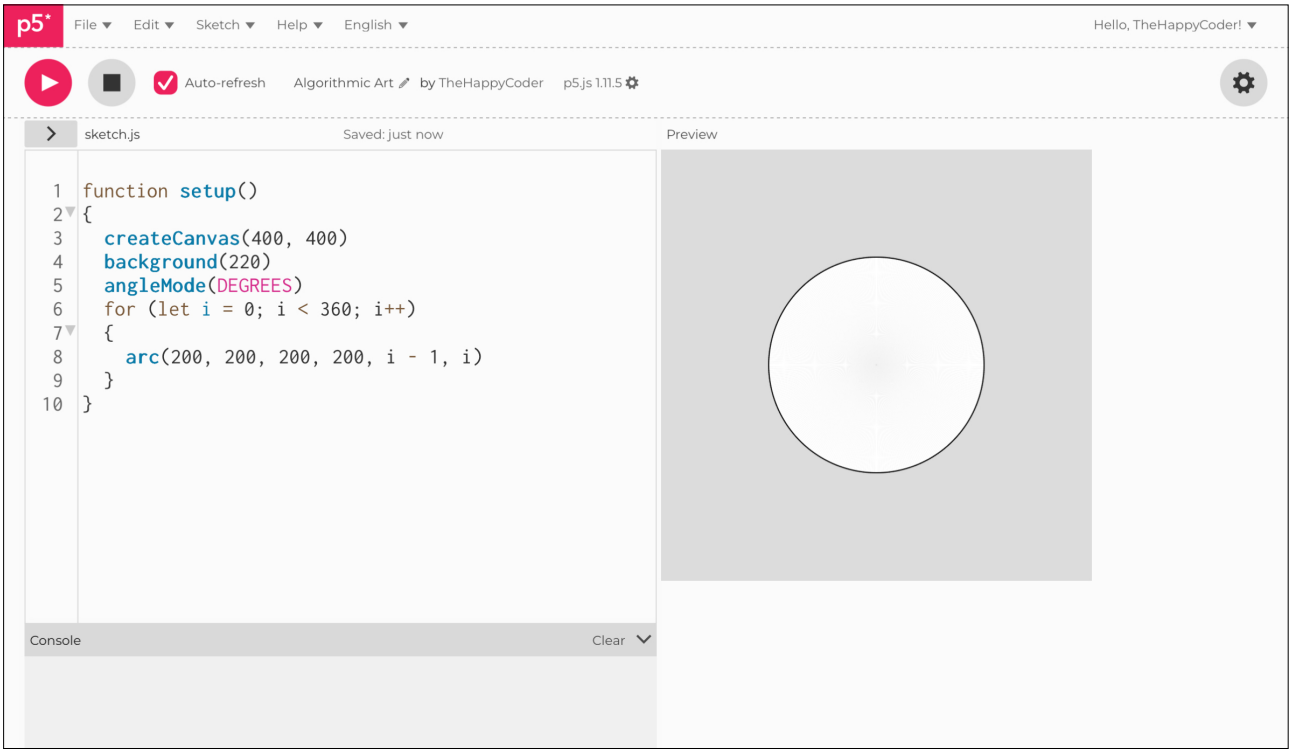
Challenge

What happens if you start with `0°`?

Code Explanation

`for ( let i = 0; i < 360; i++)`	A for() loop starting at 0° and finishing at 360° in increments of 1°.

Figure B9.3



Sketch B9.4 random

We can now add some randomness to the circle so that each arc has a slightly different value.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  for (let i = 0; i < 360; i++)
  {
    arc(200, 200, 200 + random(-10, 10), 200 + random(-10, 10), i - 1, i)
  }
}
```

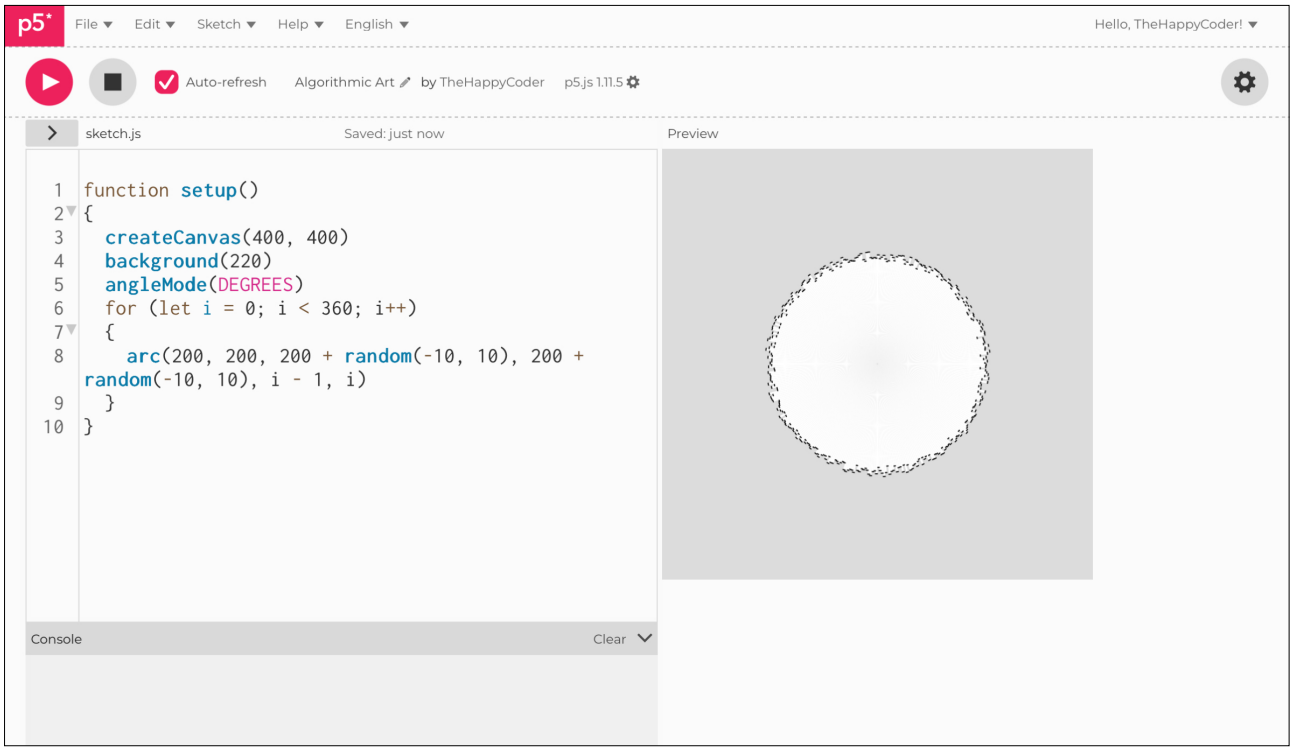
Notes

Each diameter (width and height) will vary randomly between `-10` and `+10`.

Challenge

1. Add in a `noFill()`.
2. Try `i - 5` (or more). Why do you think you get that effect?

Figure B9.4



Sketch B9.5 accentuating the randomness

This just pushes the boat out.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  noFill()
  for (let i = 0; i < 360; i++)
  {
    strokeWeight(2)
    arc(200, 200, 250 + random(-30, 30), 250 + random(-30, 30), i - 15, i)
  }
}
```

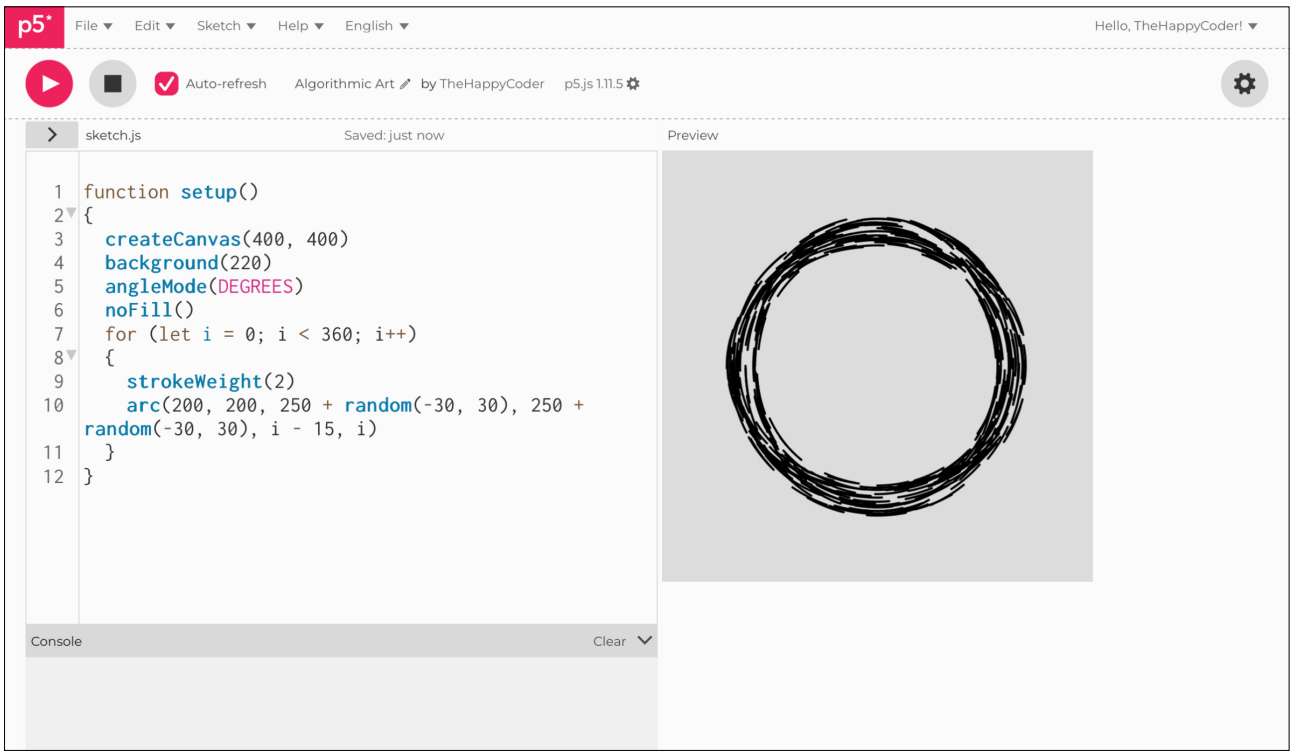
Notes

Added `noFill()` and `strokeWeight()`.

Challenge

Add some colour and play with the values.

Figure B9.5



Sketch B9.6 spiral

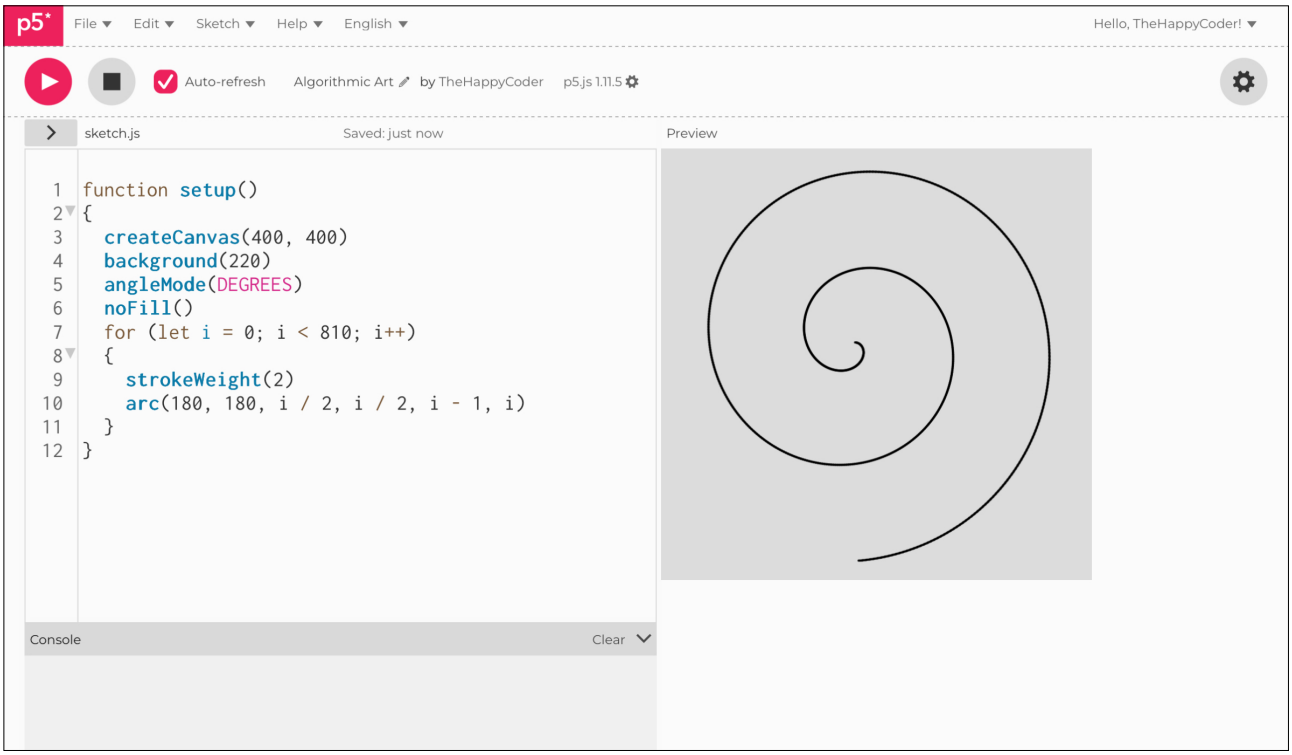
We can make it do a spiral.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  noFill()
  for (let i = 0; i < 810; i++)
  {
    strokeWeight(2)
    arc(180, 180, i / 2, i / 2, i - 1, i)
  }
}
```

Notes

Moved the centre of the spiral so it fits on the canvas.

Figure B9.6



Sketch B9.7 being a bit more OPEN

! Start with a new sketch

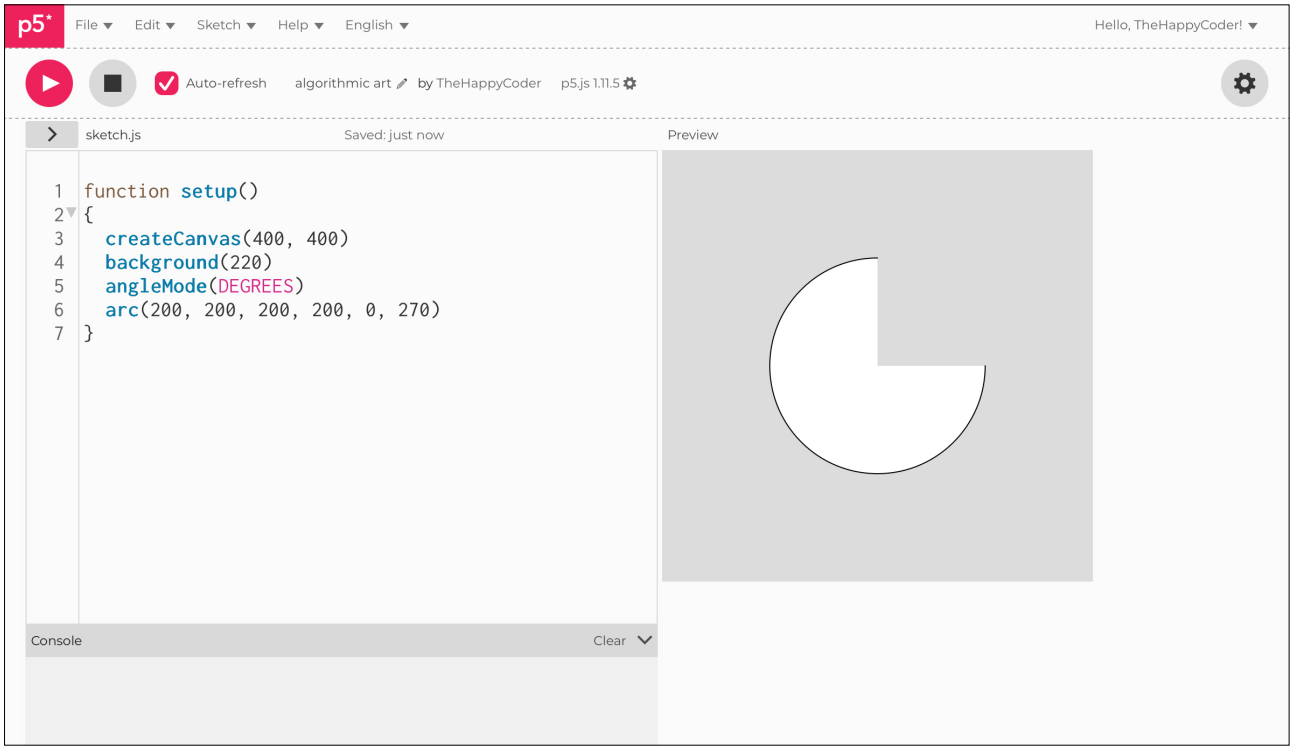
There are a number of attributes that you can use that change the appearance of an arc. The first one we will look at is called OPEN.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  arc(200, 200, 200, 200, 0, 270)
}
```

Notes

This is an arc through to 270°.

Figure B9.7



Sketch B9.8 OPEN sesame

Adding the word OPEN at the end of the function (7th argument).

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  arc(200, 200, 200, 200, 0, 270, OPEN)
}
```

Notes

It has a dramatic effect.

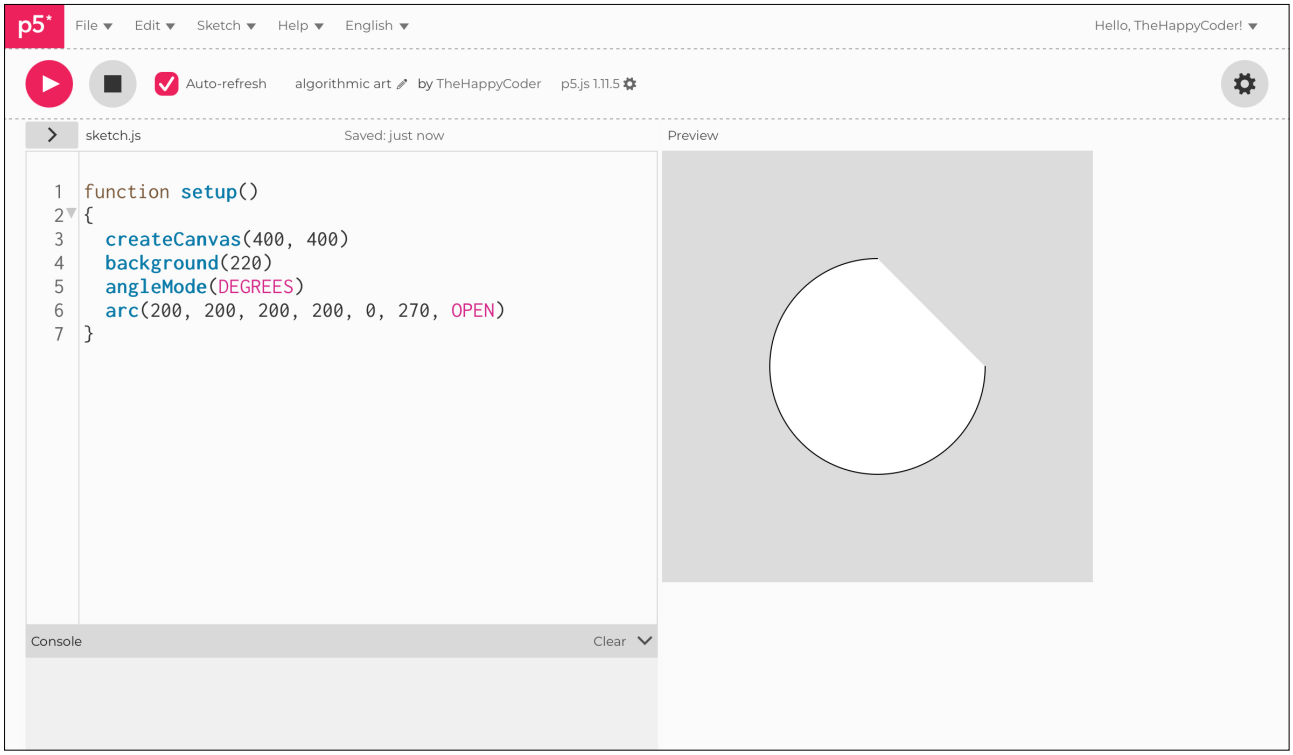
Challenge

Try it with other angles.

Code Explanation

`arc(200, 200, 200, 200, 0, 270, OPEN)`	Adding the attribute OPEN as the seventh argument.

Figure B9.8



Sketch B9.9 strikes a CHORD

We will try a different attribute, the CHORD.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  arc(200, 200, 200, 200, 0, 270, CHORD)
}
```

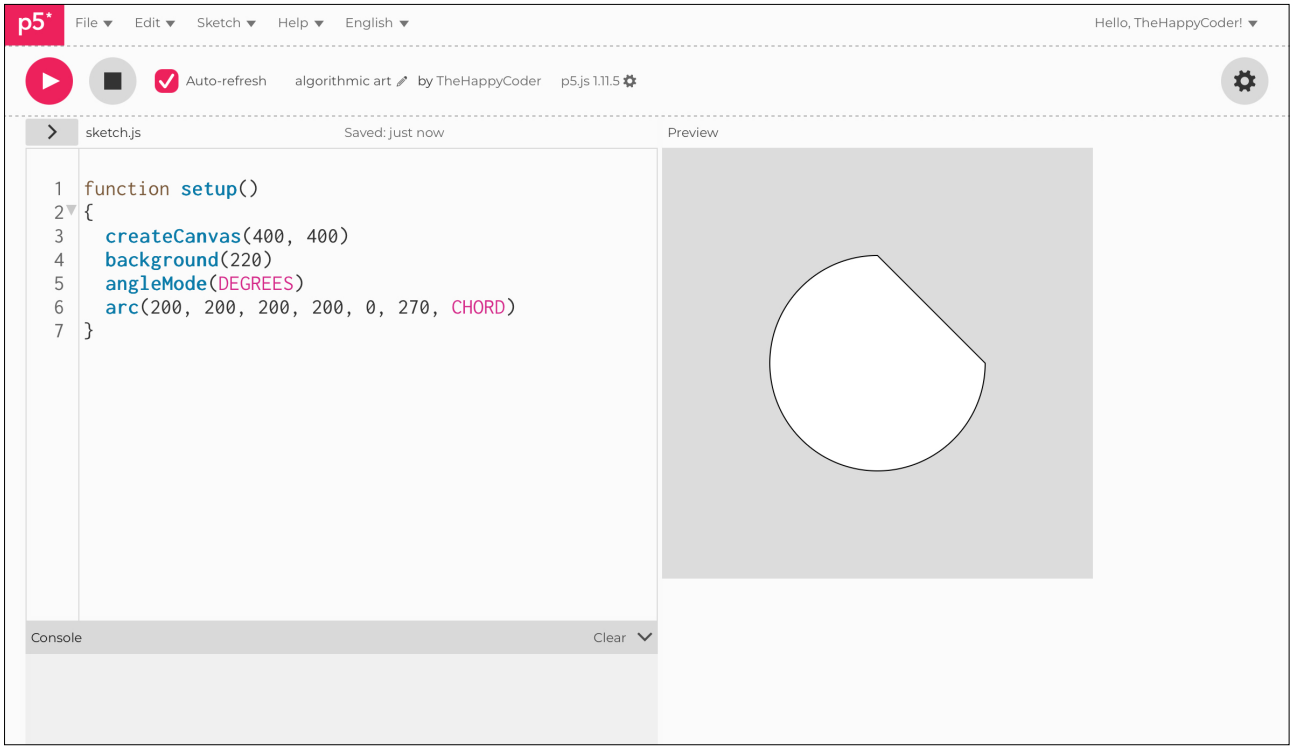
Notes

It has the same effect but also draws the outline.

Code Explanation

`arc(200, 200, 200, 200, 0, 270, CHORD)`	Adding the attribute CHORD as the seventh argument.

Figure B9.9



Sketch B9.10 a piece of the PIE

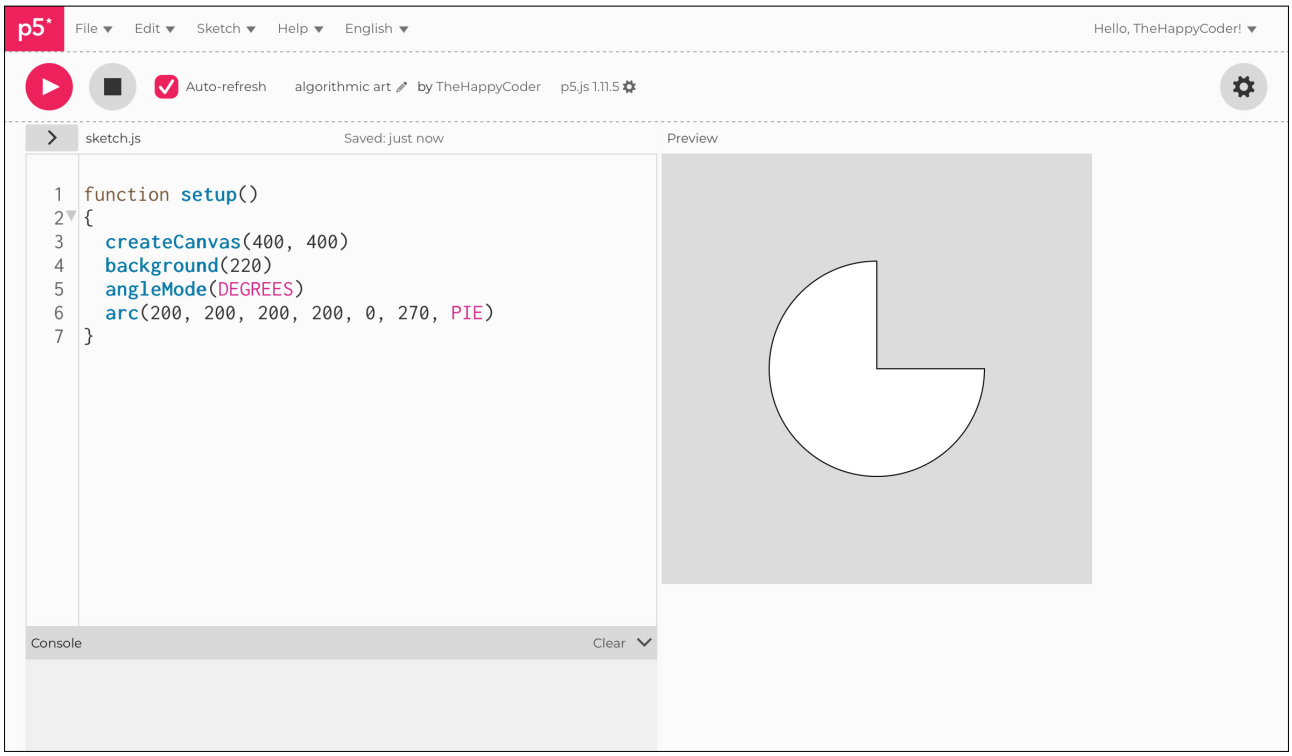
Another attribute, this time PIE.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  arc(200, 200, 200, 200, 0, 270, PIE)
}
```

Notes

More like the original but with an outline.

Figure B9.10



Sketch B9.11 the making of a PAC-MAN

I will leave you to work out the logic, just a bit of fun; art can be anything.

```
let bite = 10
let angleA
let angleB

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background('darkblue')
  fill('yellow')
  angleA = bite * sin(millis()/2) + bite + 0.01
  angleB = -angleA
  arc(200, 200, 200, 200, angleA, angleB, PIE)
  fill('darkblue')
  circle(225, 150, 25)
}
```

Notes

The sine of an angle fluctuates between `-1` and `+1`, that is why we multiply by the bit and add it; it doubles and then cancels itself out. We have a small addition (`0.01`) so that it doesn't close all the way to `zero`.

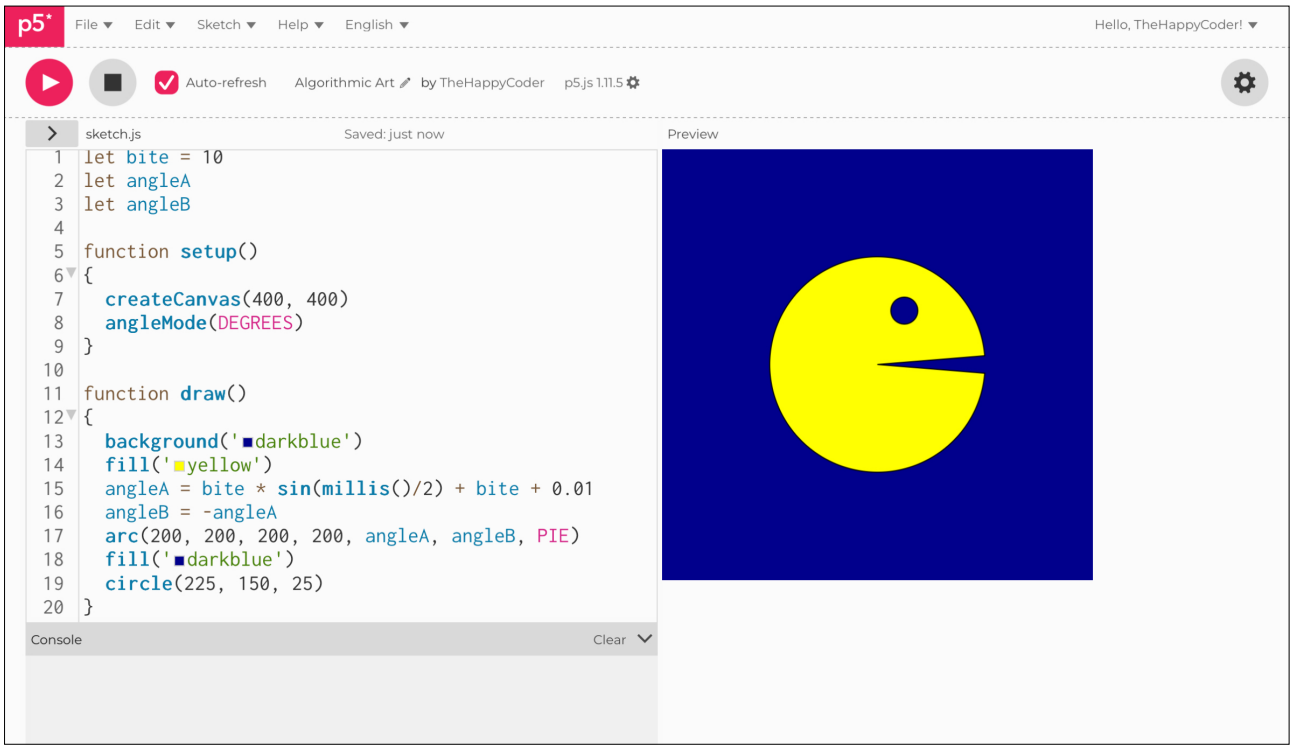
Challenge

You could use `frameCount` which also adds up very quickly but you may need to multiply by `15` or more/less

Code Explanation

`millis()`	Counts the number of milliseconds since the sketch started running.

Figure B9.11



These are our four arcs.

Arc 1: `arc(x, y, spacing, spacing, 0, 270)`
Arc 2: `arc(x, y, spacing, spacing, 90, 0)`
Arc 3: `arc(x, y, spacing, spacing, 270, 180)`
Arc 4: `arc(x, y, spacing, spacing, 180, 90)`

They start and finish in different places.

Figure 2: our four arcs

We are going to randomly select them, where a quarter of the time any single one is selected. As a basis, we will use the `10PRINT` code from a previous unit, which I have adapted slightly (rather than going through the whole thing again).

We will create a random number (between `0` and `1`), then divide the probability into four equal parts:

A) less than `0.25`
B) between `0.25` and `0.5`
C) between `0.5` and `0.75`
D) more than `0.75`

Sketch B9.12 10PRINT arcs

I have highlighted the main elements that are very different.

```
let x = 0
let y = 0
let spacing = 25

function setup()
{
  createCanvas(400, 400)
  ellipseMode(CORNER)
  angleMode(DEGREES)
  background(220)
  noFill()
  x = spacing
  y = spacing
}

function draw()
{
  if (random(1) <= 0.25)
  {
    arc(x, y, spacing, spacing, 0, 270)
  }
  else if (random(1) >= 0.25 && random(1) <= 0.5)
  {
    arc(x, y, spacing, spacing, 90, 0)
  }
  else if (random(1) >= 0.5 && random(1) <= 0.75)
  {
    arc(x, y, spacing, spacing, 270, 180)
  }
  else
  {
    arc(x, y, spacing, spacing, 180, 90)
  }
  x += spacing
  if (x >= width - spacing)
```

```
  {
    x = spacing
    y += spacing
  }
  if (y >= height - spacing)
  {
    noLoop()
  }
}
```

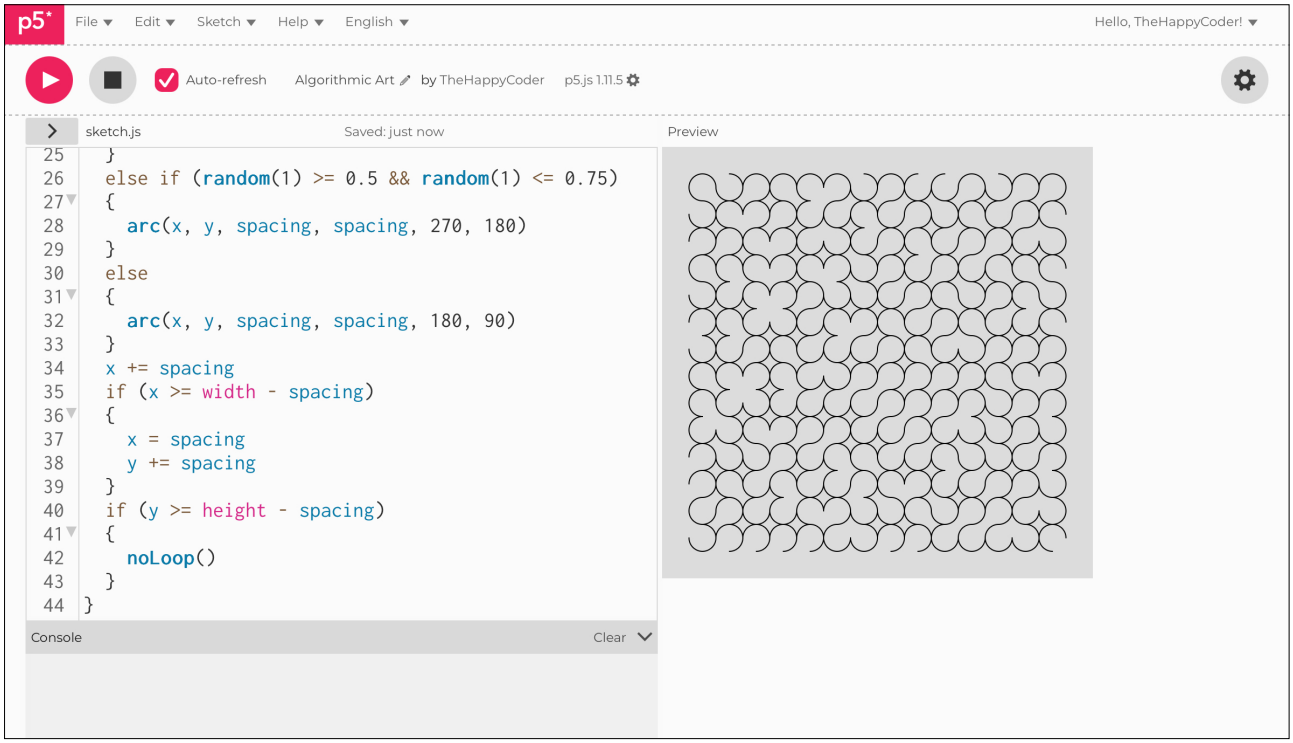
Notes

The `ellipseMode()` function allows you to move the centre of the circle (in terms of coordinates).

Code Explanation

`ellipseMode(CORNER)`	Puts the coordinates of the circle in the top left-hand rather than in the centre.
`if (random(1) <= 0.25)`	If less than `0.25`, draw the first arc.
`else if (random(1) >= 0.25 && random(1) <= 0.5)`	If between than `0.25` and `0.5` draw the second arc.
`else if (random(1) >= 0.5 && random(1) <= 0.75)`	If between `0.5` and `0.75`, draw a third arc.
`else`	If none of the above, then draw the fourth arc.

Figure B9.12



Sketch B9.13 mapping

We can use a function called `map()` to map the horizontal (`mouseX`) position of the mouse across the canvas to the angle of the arc. The `map()` function takes five arguments.

The first one is the input variable you want to map, in this case it is the `mouseX` value which will range from `0` to `400`. The second two are start and end values for that input value (`0` and `400`), and the final two values are the values you want to map to, in this case the `angle` of the arc which we want in a range of `0` to `360`.

In effect, we are mapping `0` to `400` onto `0` to `360`.

```
let angle = 0

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
}

function draw()
{
  background(220)
  fill(255)
  angle = map(mouseX, 0, width, 0, 360)
  if (angle > 360)
  {
    angle = 360
  }
  if (angle < 0)
  {
    angle = 0
  }
  arc(200, 200, 200, 200, 0, angle, PIE)
  fill(0)
  textSize(32)
  text(floor(angle) + '°', 100, 100)
}
```

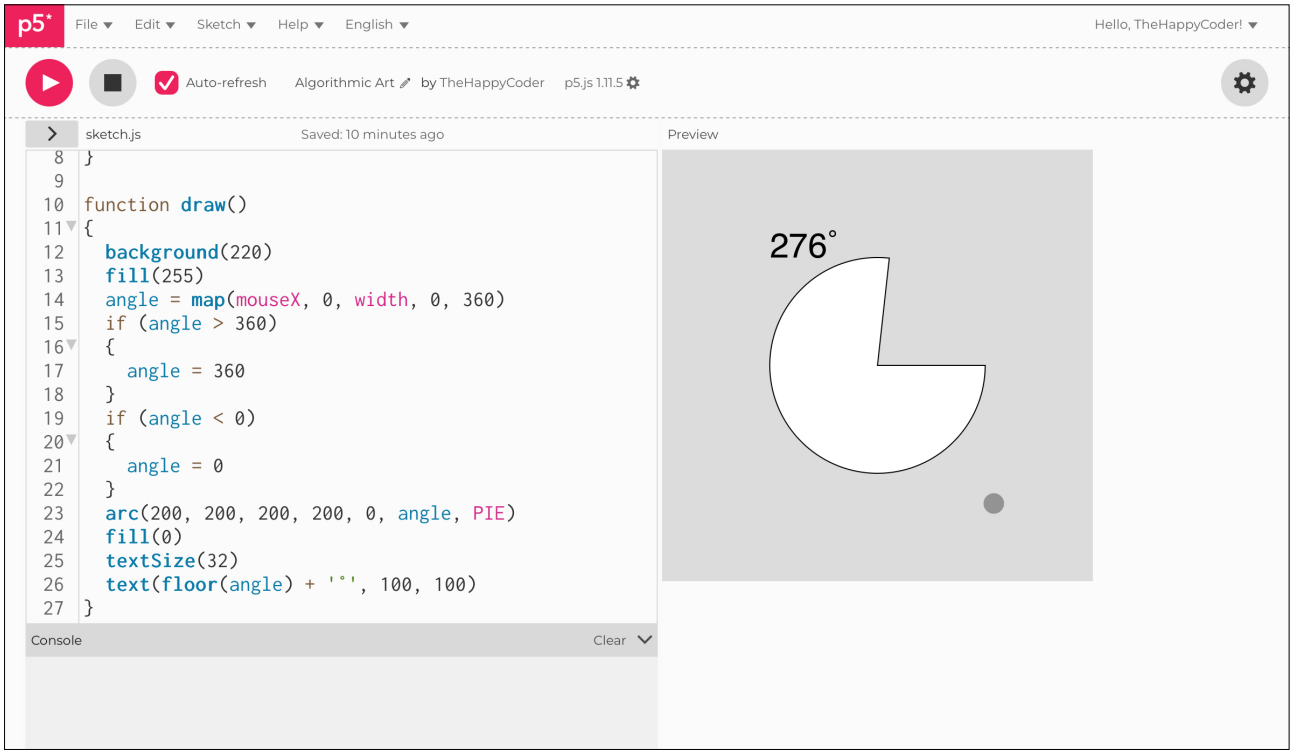
Notes

We have also put limitations on how far to the left and right of the canvas we can go; otherwise, you will have values bigger than 360° and negative numbers.

Code Explanation

`angle = map(mouseX, 0, width, 0, 360)`	Mapping the mouse position (0–400) onto the angle (0–360).

Figure B9.13



The Joy of Coding Algorithmic Art Module B Unit #10 phyllotaxis

Module B Unit #10: phyllotaxis

In nature, there is something called the golden ratio and the golden angle, which is a mathematical explanation for the pattern on, say, a sunflower. We are going to recreate the pattern that you can see in nature. If you want to know more, I suggest that you Google it and read up about it for yourself.

It will probably cover a lot of maths, but it is simple maths that can be coded, as you will see towards the end of this unit. But first, let's draw some circles and spirals to get us started.

Sketch B10.1 start with a point

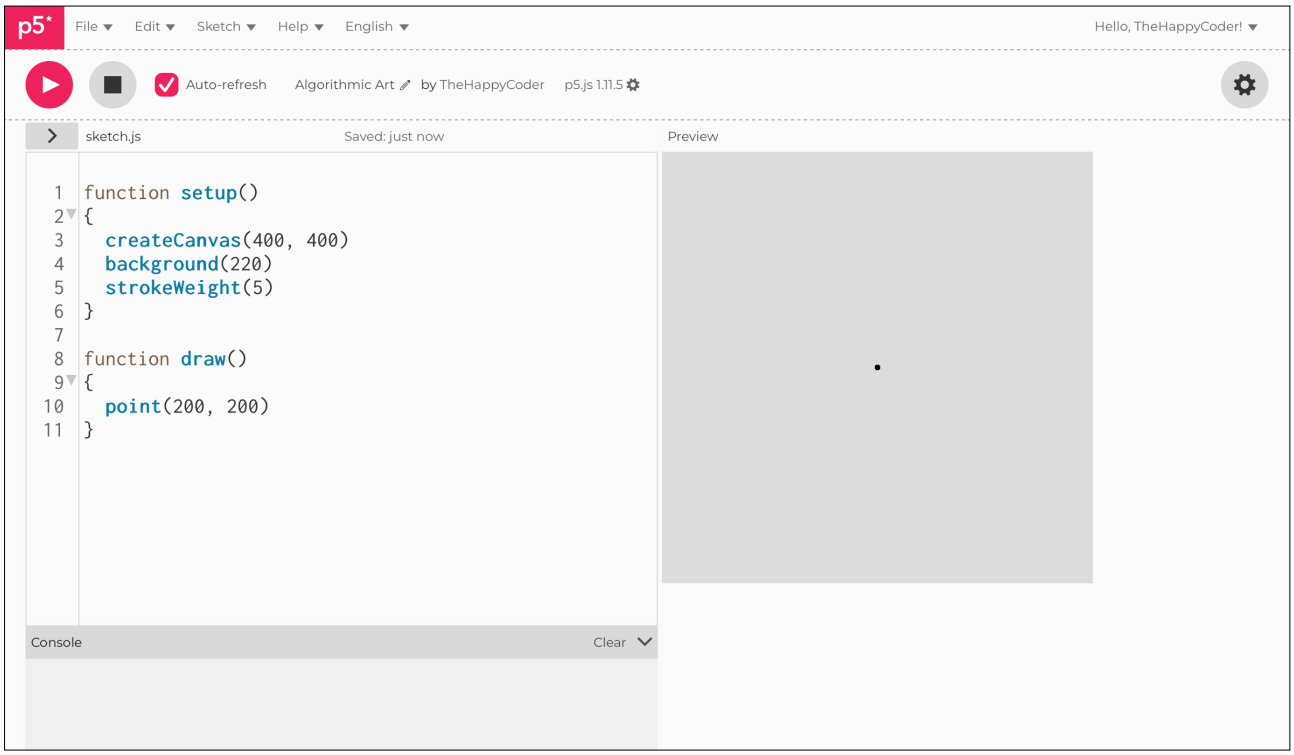
A simple point in the centre of the canvas.

```
function setup()
{
  createCanvas(400, 400)
  background(220)
  strokeWeight(5)
}

function draw()
{
  point(200, 200)
}
```

Notes

Giving it a heavy stroke weight so we can see it.

Figure B10.1



Sketch B10.2 the x and y variables

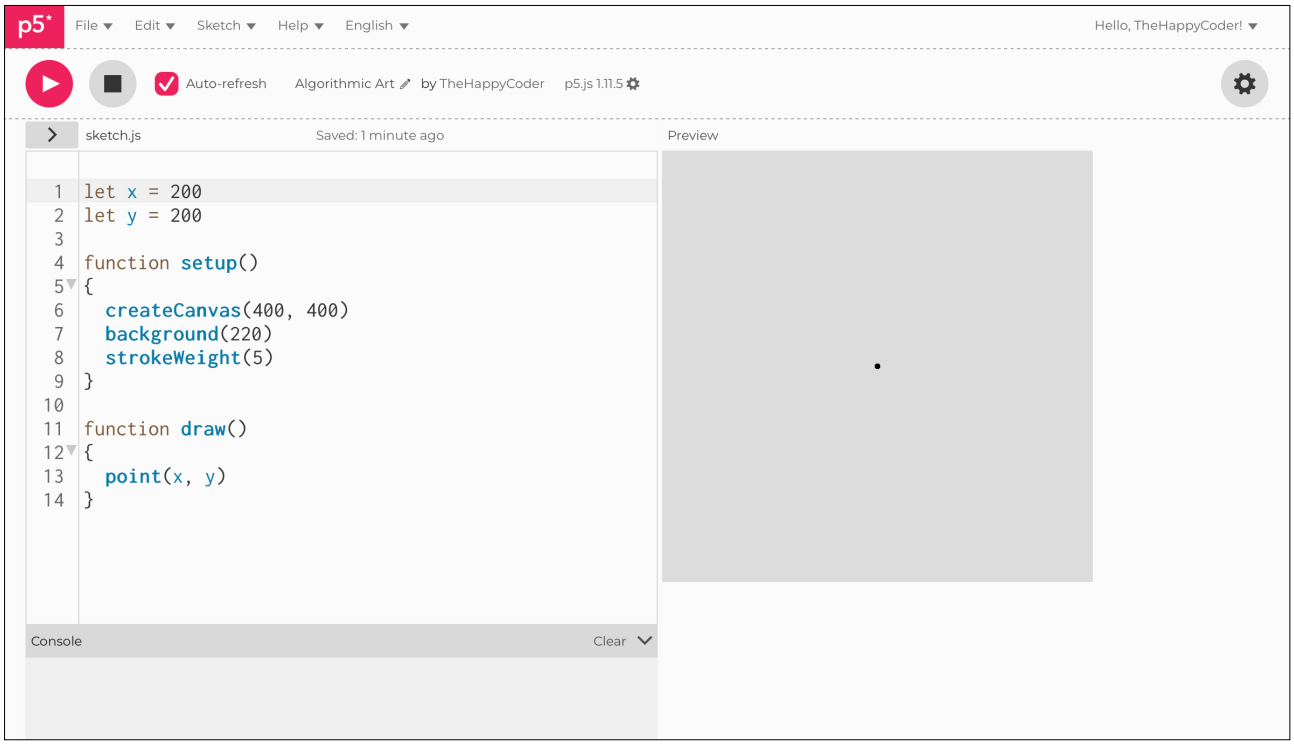
We will give it an x and y variable but still draw it in the centre.

```
let x = 200
let y = 200

function setup()
{
  createCanvas(400, 400)
  background(220)
  strokeWeight(5)
}

function draw()
{
  point(x, y)
}
```

Notes

You should still have the point in the centre, just as before.

Figure B10.2



Sketch B10.3 translate

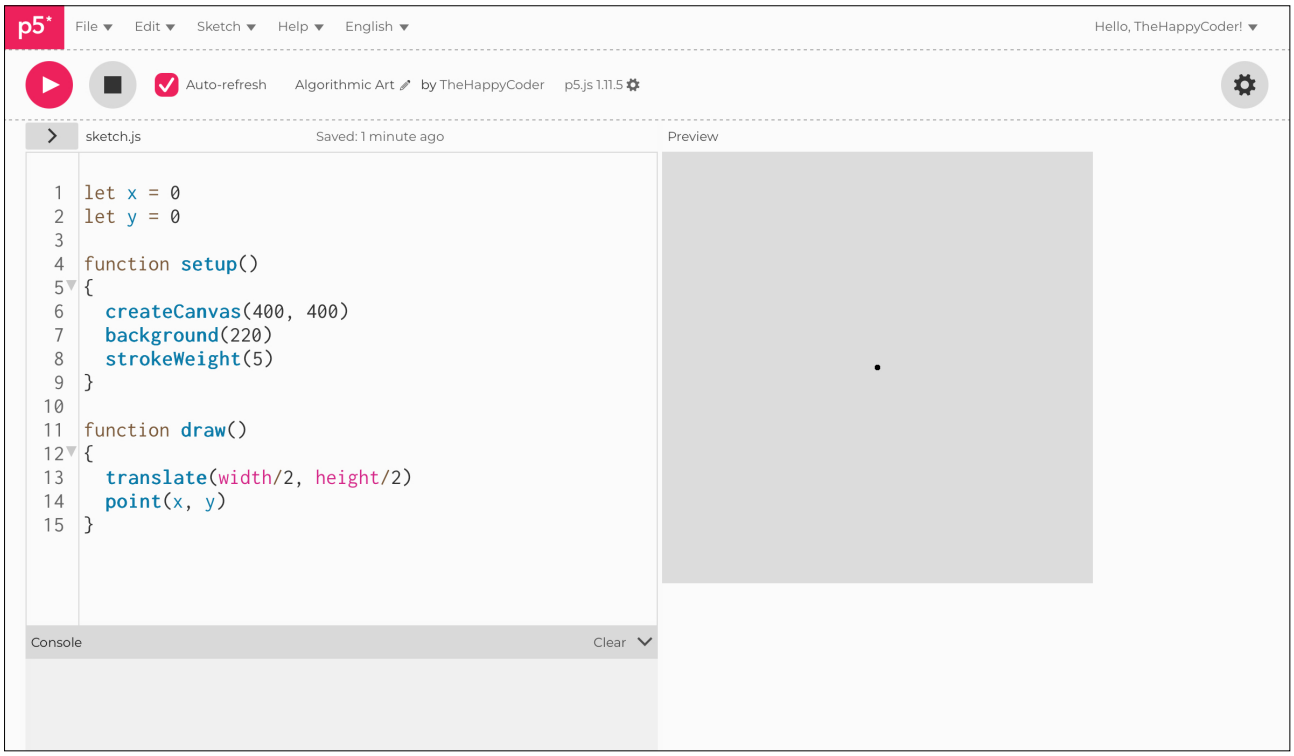
We will translate it into the centre using the x, y variables.

```
let x = 0
let y = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  strokeWeight(5)
}

function draw()
{
  translate(width/2, height/2)
  point(x, y)
}
```

Notes

Exactly as before.

Figure B10.3



Sketch B10.4 background

! Comment out `background()` in `setup()`.

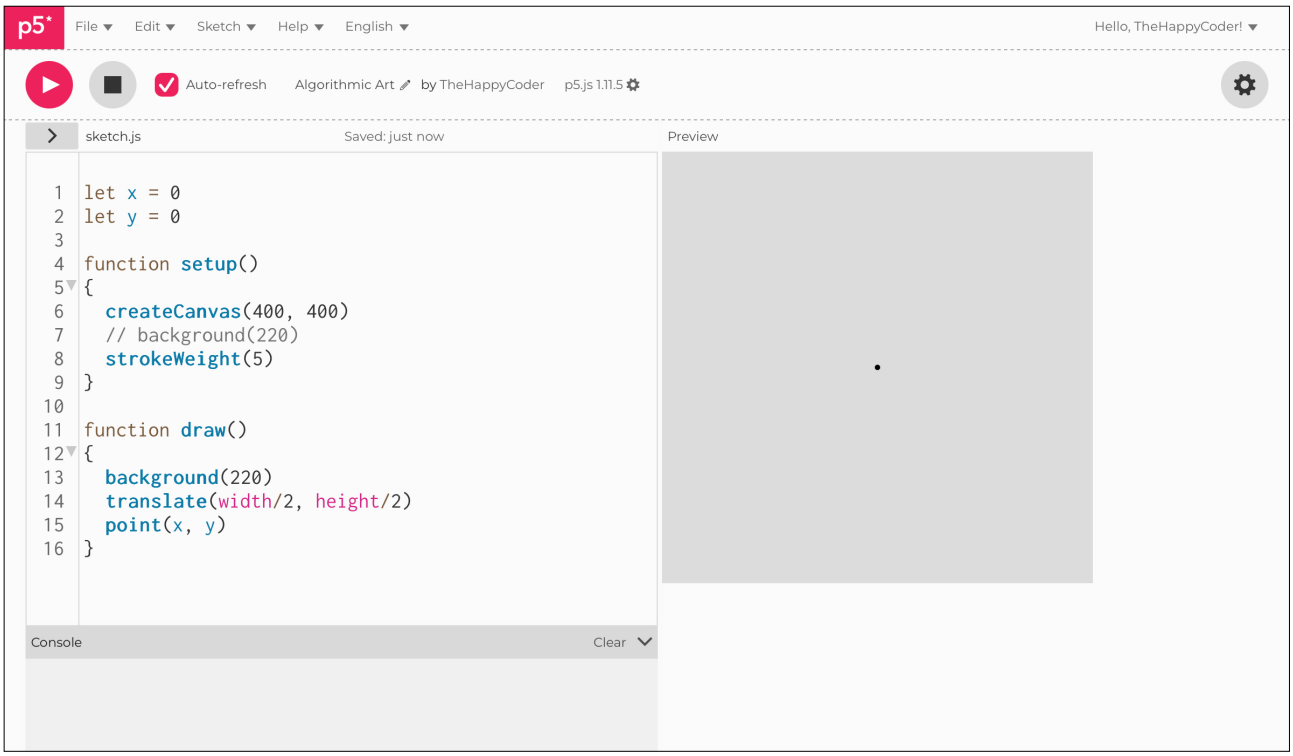
We are going to put the background in the `draw()` function so that it draws the point moving.

```
let x = 0
let y = 0

function setup()
{
  createCanvas(400, 400)
  // background(220)
  strokeWeight(5)
}

function draw()
{
  background(220)
  translate(width/2, height/2)
  point(x, y)
}
```

Notes

One small step at a time, nothing you haven't already seen.

Figure B10.4



Sketch B10.5 angle in radians

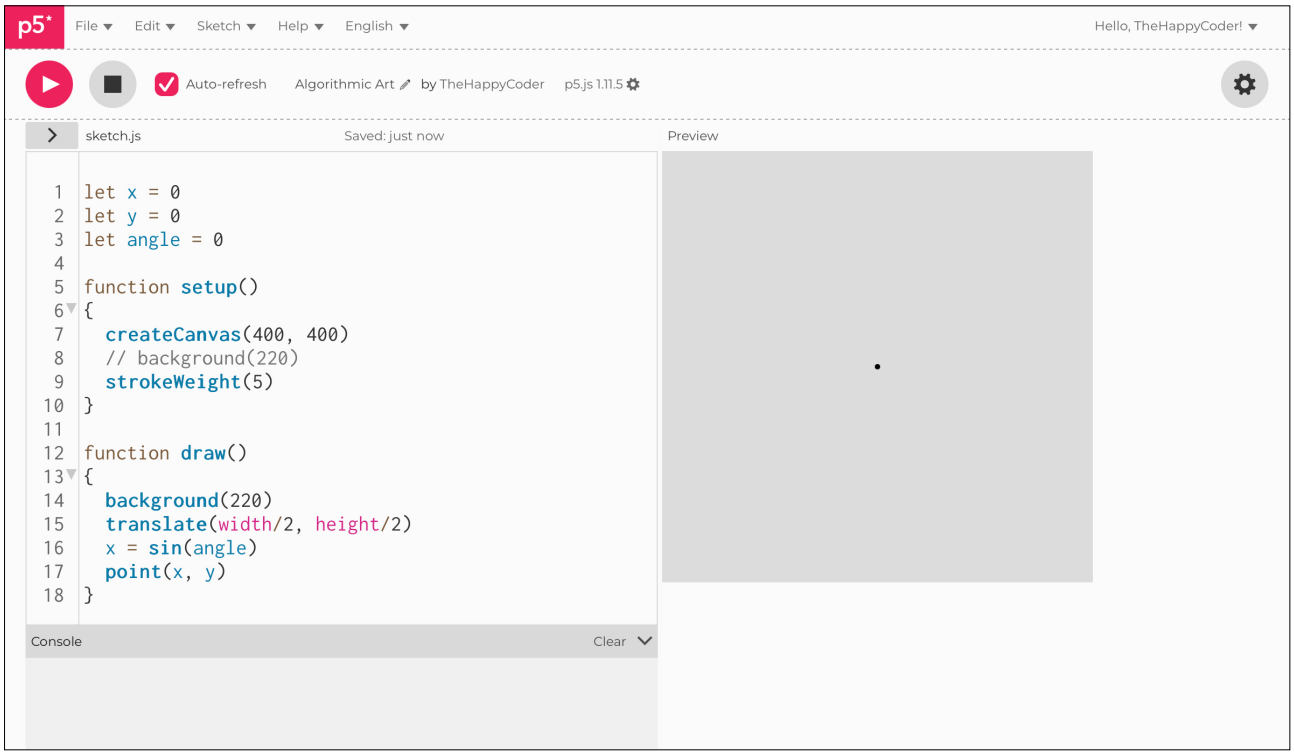
We want to oscillate the point about the centre using a sine wave motion. We first need to create a variable called angle. That angle is in radians, and the sine of those radians returns values between −1 and +1.

```
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  // background(220)
  strokeWeight(5)
}

function draw()
{
  background(220)
  translate(width/2, height/2)
  x = sin(angle)
  point(x, y)
}
```

Notes

A lot of coding just to draw a dot on the canvas.

Figure B10.5



Sketch B10.6 incremental angle

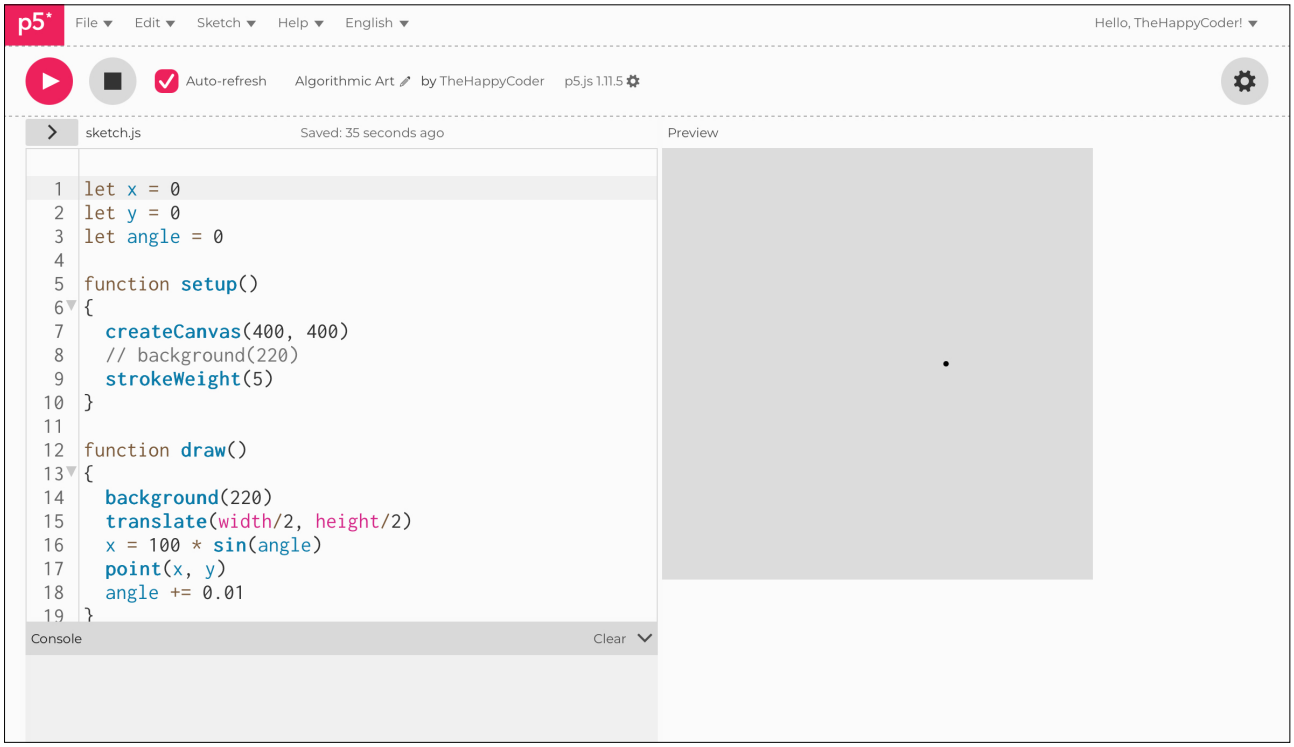
What we need to do is move it by increasing the angle in every loop of draw. We will do it by a very small amount each time. To add a number to a variable, we can use a shortcut method using `+=`. In this case, we are going to be adding `0.01` to the value of the angle each iteration of the loop. We start with `0`, and then it keeps on increasing. The sine of the angle always stays between `-1` and `+1`.

```
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  // background(220)
  strokeWeight(5)
}

function draw()
{
  background(220)
  translate(width/2, height/2)
  x = 100 * sin(angle)
  point(x, y)
  angle += 0.01
}
```

Notes

We multiply by 100 (the * symbol means multiply) because otherwise the point will only move one pixel! What you should see is it oscillating in a line from around 100 to 300 halfway down the canvas.

Figure B10.6



Sketch B10.7 circular motion

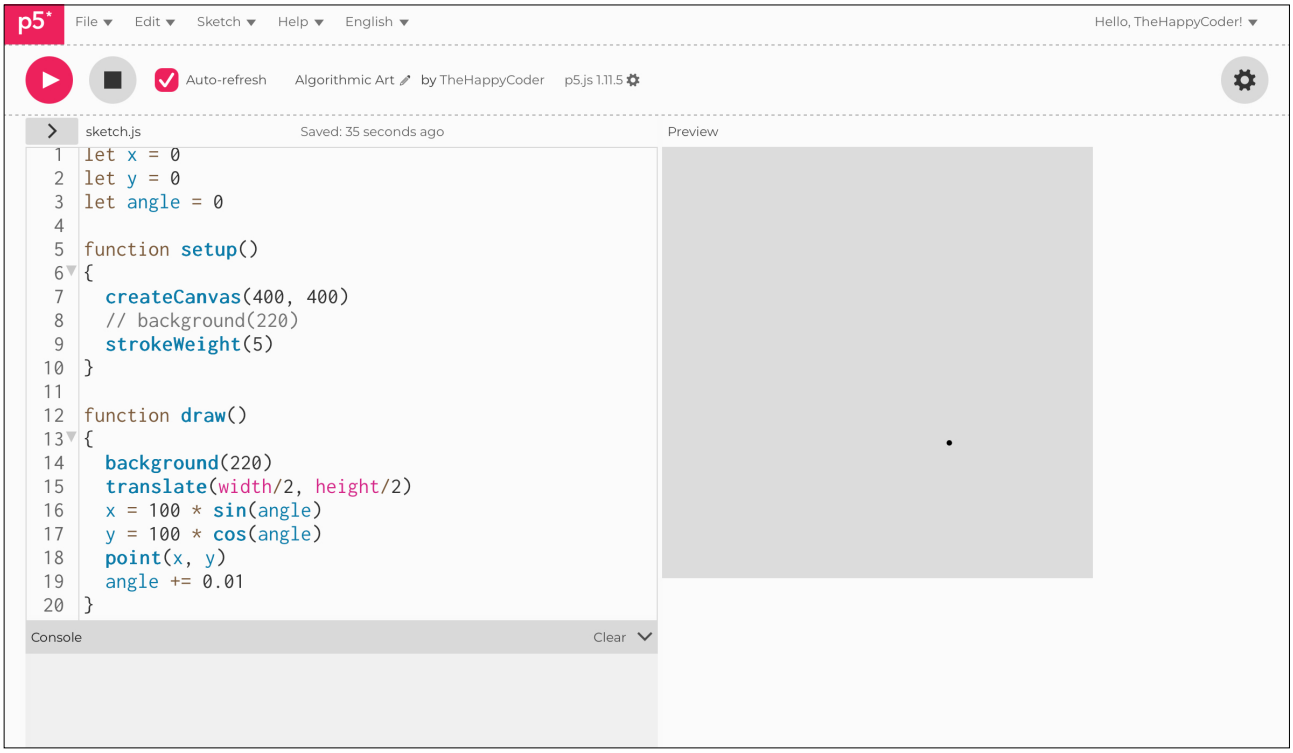
Next, we try to get it to move in a circle.

```
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  // background(220)
  strokeWeight(5)
}

function draw()
{
  background(220)
  translate(width/2, height/2)
  x = 100 * sin(angle)
  y = 100 * cos(angle)
  point(x, y)
  angle += 0.01
}
```

Notes

You should see the dot moving in a circle with a radius of 100.

Figure B10.7



Sketch B10.8 drawing a circle

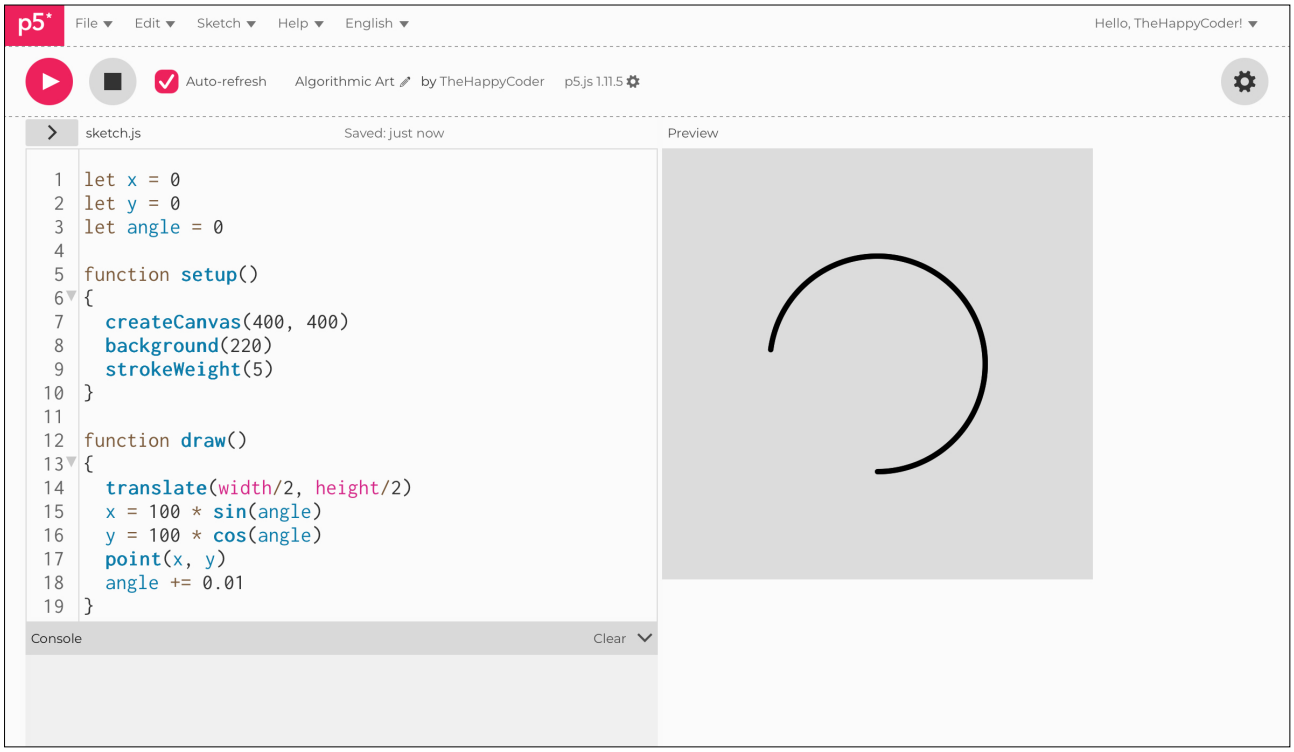
! Remove the `background()` function in `draw()` and reinstate it in `setup()` as before, and we will see the dot scribe a permanent circle on the canvas.

```
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  strokeWeight(5)
}

function draw()
{
  // background(220)
  translate(width/2, height/2)
  x = 100 * sin(angle)
  y = 100 * cos(angle)
  point(x, y)
  angle += 0.01
}
```

Notes

You will get a circle drawn on the canvas and it will keep on drawing.

Figure B10.8



Sketch B10.9 vertex circle

Going to make quite a few changes.

- The variable `r` for the radius
- The `for()` loop for the angle
- The `vertex()` instead of a `point()`
- The need to `beginShape()` and `endShape()`

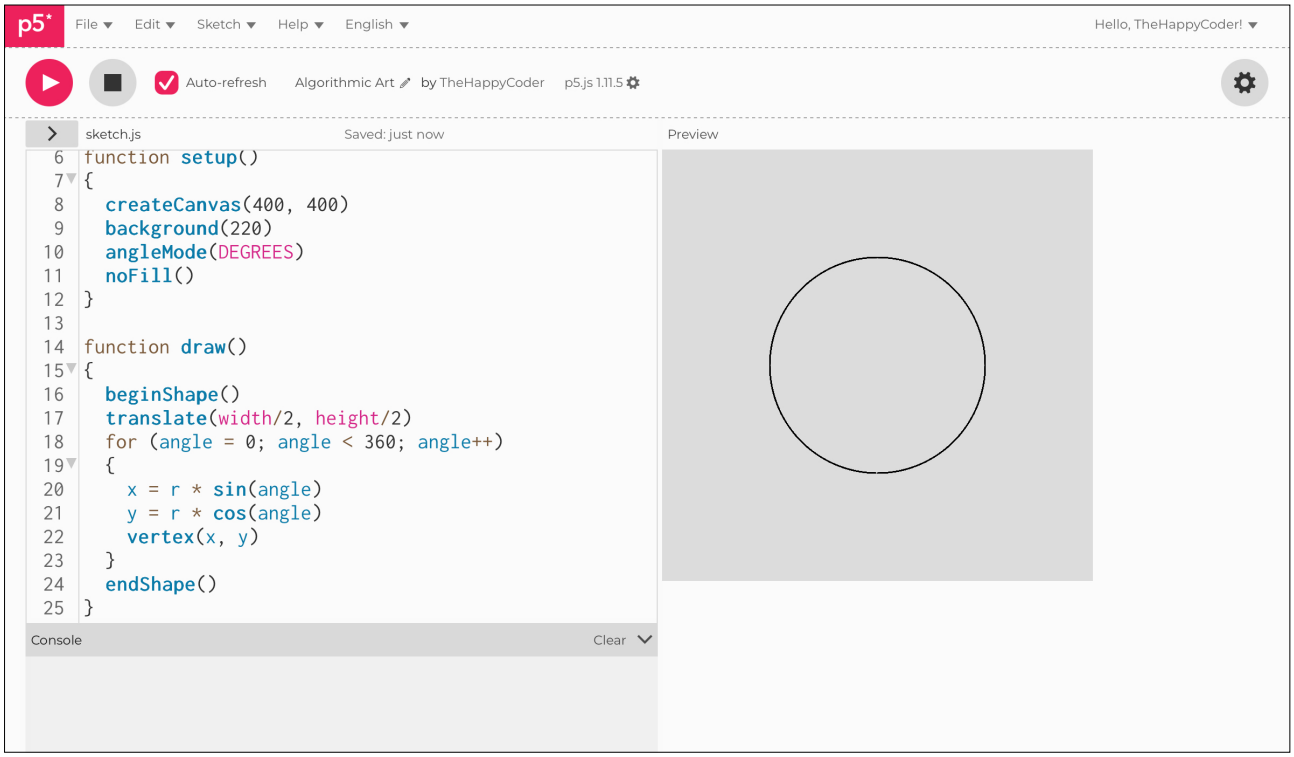
! remove `strokeWeight(5)`, and other bits of code.

```
let x = 0
let y = 0
let r = 100
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  noFill()
}

function draw()
{
  beginShape()
  translate(width/2, height/2)
  for (angle = 0; angle < 360; angle++)
  {
    x = r * sin(angle)
    y = r * cos(angle)
    vertex(x, y)
  }
  endShape()
}
```

Notes

A few changes, you might say, all fairly obvious.

Figure B10.9



Sketch B10.10 random loop

Let us try something a bit different, making the radius random. We need a `noLoop()` at the end, though.

```
let x = 0
let y = 0
let r = 100
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  noFill()
}

function draw()
{
  beginShape()
  translate(width/2, height/2)
  for (angle = 0; angle < 360; angle++)
  {
    x = random(r) * sin(angle)
    y = r * cos(angle)
    vertex(x, y)
  }
  endShape()
}
```

Notes

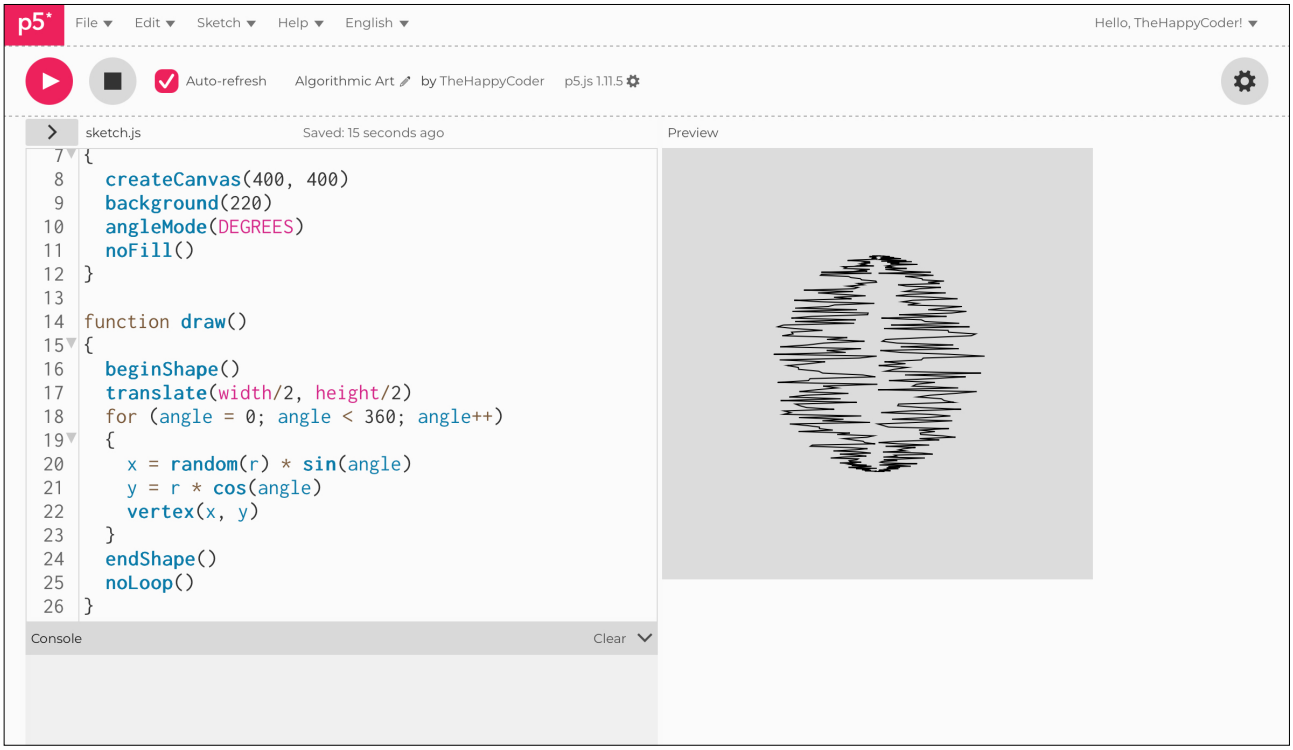
We have only changed one variable to be random.

Challenges

1. Change the `y` to a random value as well.
2. Change the random value for `x` to be between `50` and `100`.
3. Try the following:

```
x = random(-r, r) * sin(angle)
y = r*cos(angle)
```

Figure B10.10



Sketch B10.11 spiral

Take it back to a circle (remove the random). We can alter the radius so that it starts at zero and increases incrementally. We also need to decide how many times we want to go round the circle; here I have multiplied by 20 times.

```
let x = 0
let y = 0
let r = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  noFill()
}

function draw()
{
  beginShape()
  translate(width/2, height/2)
  for (angle = 0; angle < 360 * 20; angle++)
  {
    x = r * sin(angle)
    y = r * cos(angle)
    vertex(x, y)
    r += 0.02
  }
  endShape()
}
```

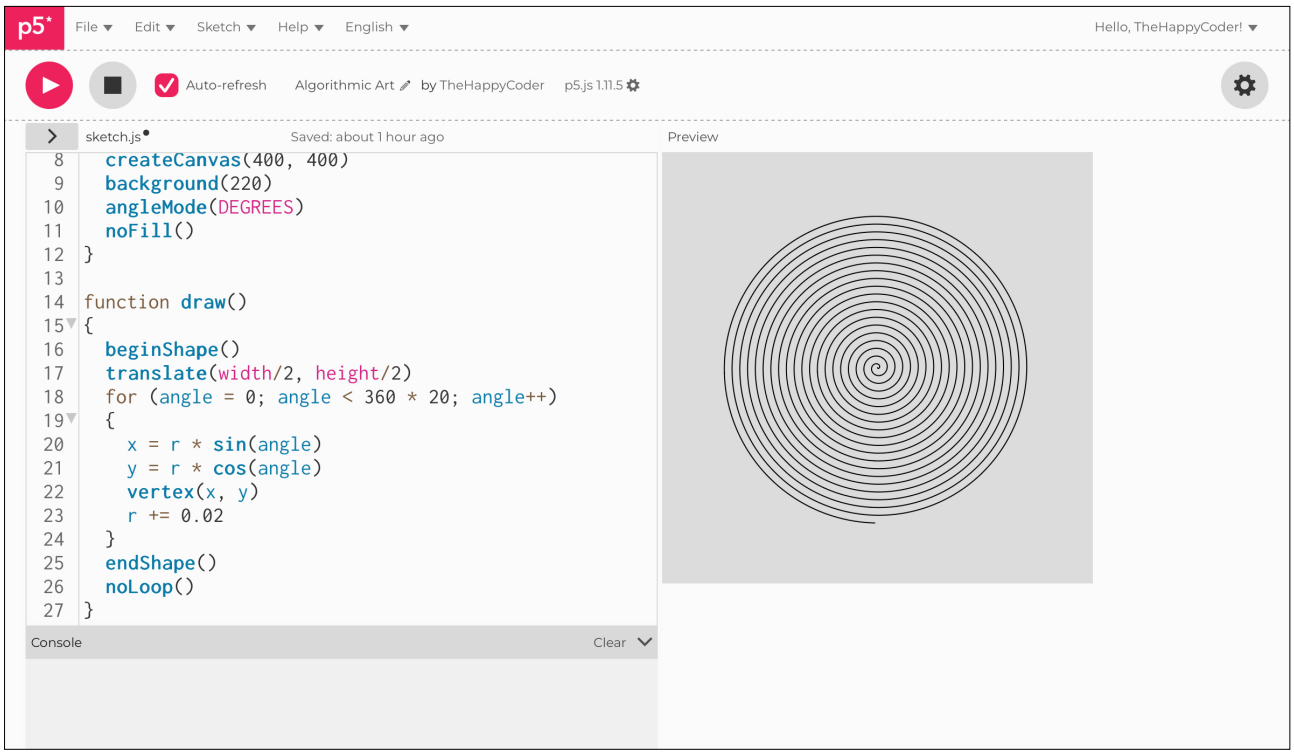
Notes

A very simple spiral. In version 2 it is nearly instant.

Challenges

1. Change the number of revolutions.
2. Change the increase in radius.
3. Add some randomness.

Figure B10.11



The Phyllotaxis

To replicate the patterns on a flower (such as a sunflower or something similar), we need to follow the maths. First, we identify the variables:

- `index`: each floret (circle) has an `index` (0, 1, 2, 3, etc.)
- `constant`: a scaling factor, something like `8`
- `radius`: is the `constant` times the square root of the `index`
- `angle`: is the `index` times the golden ratio (`137.508°`)

The angle `137.508°` is the golden angle, which is approximated by the ratios of Fibonacci numbers.

Sketch B10.12 constant variables

! Starting a brand new sketch

Let's keep it simple and build from there. First, we need the variables and constants. We will give them values of `zero` till we know what to do with them.

```
let constant = 0
let radius = 0
let index = 0
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{

}
```

Notes

This does nothing, and we have an empty `draw()` function for the moment.

Sketch B10.13 draw a circle

Now create a loop to draw the circles.

```
let constant = 0
let radius = 0
let index = 0
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
}

function draw()
{
  translate(width/2, height/2)
  for (let i = 0; i < index; i++)
  {
    x = radius * cos(angle)
    y = radius * sin(angle)
    circle(x, y, 10)
  }
}
```

Notes

Nothing to see because everything is zero.

Sketch B10.14 angleMode

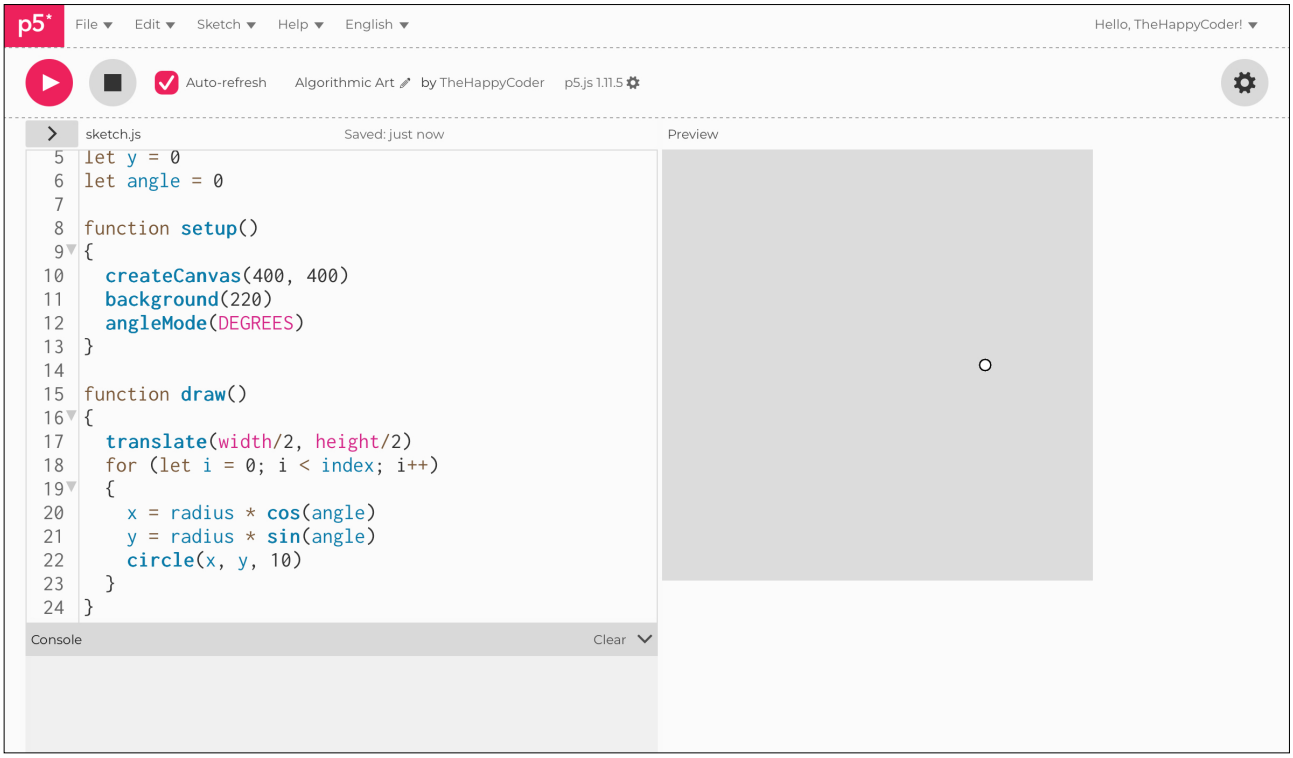
Let's put some values on this so we can see something. Also, we need to work in degrees, so we will add `angleMode()` and so the `index` needs to be `360`. The radius will be `100`.

```
let constant = 0
let radius = 100
let index = 360
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
}

function draw()
{
  translate(width/2, height/2)
  for (let i = 0; i < index; i++)
  {
    x = radius * cos(angle)
    y = radius * sin(angle)
    circle(x, y, 10)
  }
}
```

Notes

At least something has happened, just not a lot!

Figure B10.14



Sketch B10.15 using the golden angle

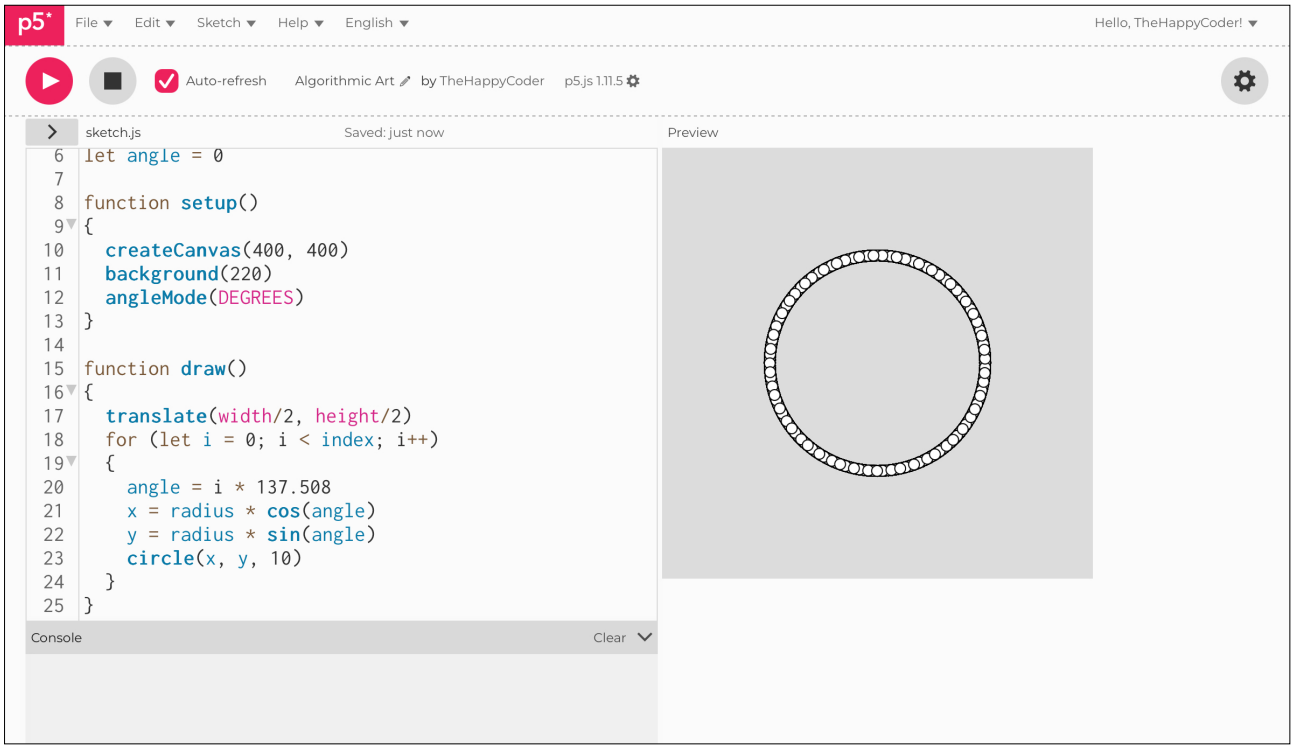
We now need to introduce an angle that changes and add the equation where the angle is the index times 137.508.

```
let constant = 0
let radius = 100
let index = 360
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
}

function draw()
{
  translate(width/2, height/2)
  for (let i = 0; i < index; i++)
  {
    angle = i * 137.508
    x = radius * cos(angle)
    y = radius * sin(angle)
    circle(x, y, 10)
  }
}
```

Notes

Because the `angle` increments are `1°` times `137.508`, it goes round several times, which is what we want; however, the `radius` is constant, which we don't want.

Figure B10.15



Sketch B10.16 the final piece of the puzzle

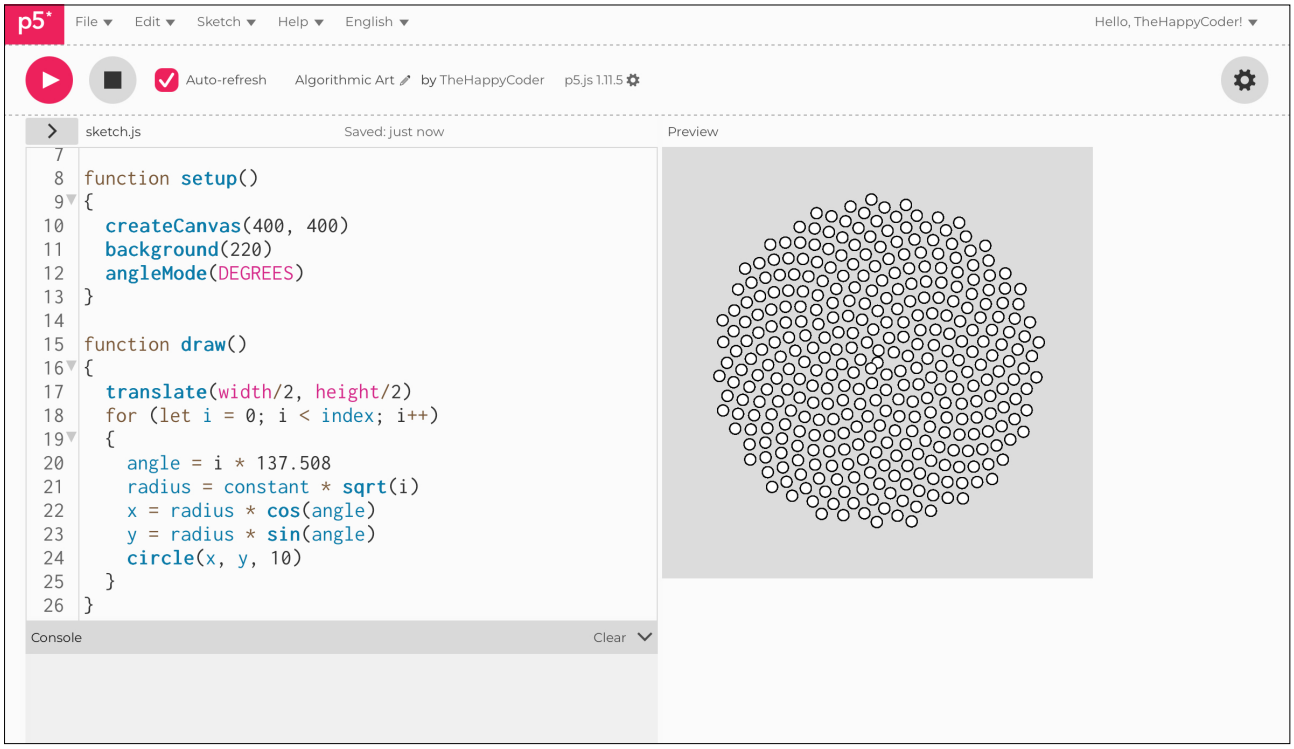
The radius is the constant times the square root of the index; in this case, it is i. We can use a function called sqrt() for this. But we need to give the constant a value; I will pick 8 because I have already played with some values; you can try others.

```
let constant = 8
let radius = 100
let index = 360
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
}

function draw()
{
  translate(width/2, height/2)
  for (let i = 0; i < index; i++)
  {
    angle = i * 137.508
    radius = constant * sqrt(i)
    x = radius * cos(angle)
    y = radius * sin(angle)
    circle(x, y, 10)
  }
}
```

Notes

Wow, we get a rather nice pattern.

Figure B10.16



Sketch B10.17 greyness

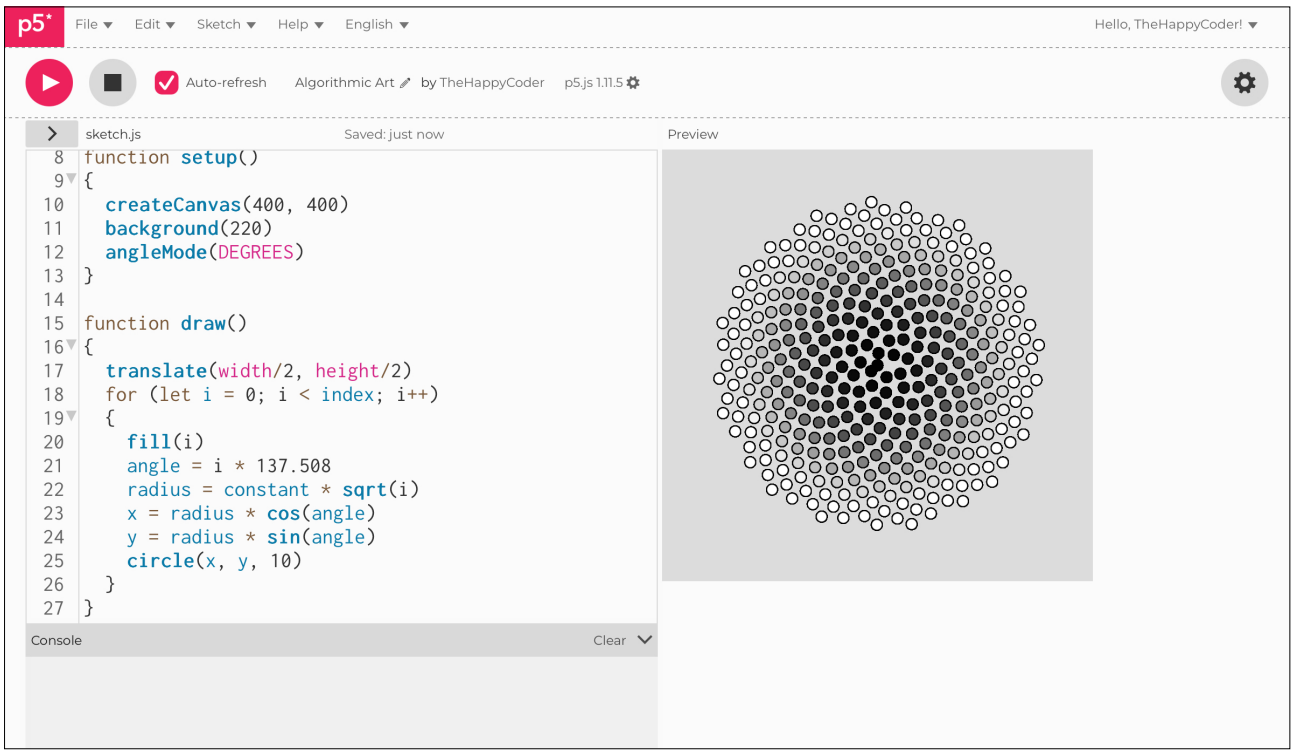
So we can now give it some colour of sorts. Let us start really simply with grey before we move on to other more dynamic colours.

```
let constant = 8
let radius = 100
let index = 360
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
}

function draw()
{
  translate(width/2, height/2)
  for (let i = 0; i < index; i++)
  {
    fill(i)
    angle = i * 137.508
    radius = constant * sqrt(i)
    x = radius * cos(angle)
    y = radius * sin(angle)
    circle(x, y, 10)
  }
}
```

Notes

We have something, but we can do even better.

Figure B10.17



Sketch B10.18 mapping the colour

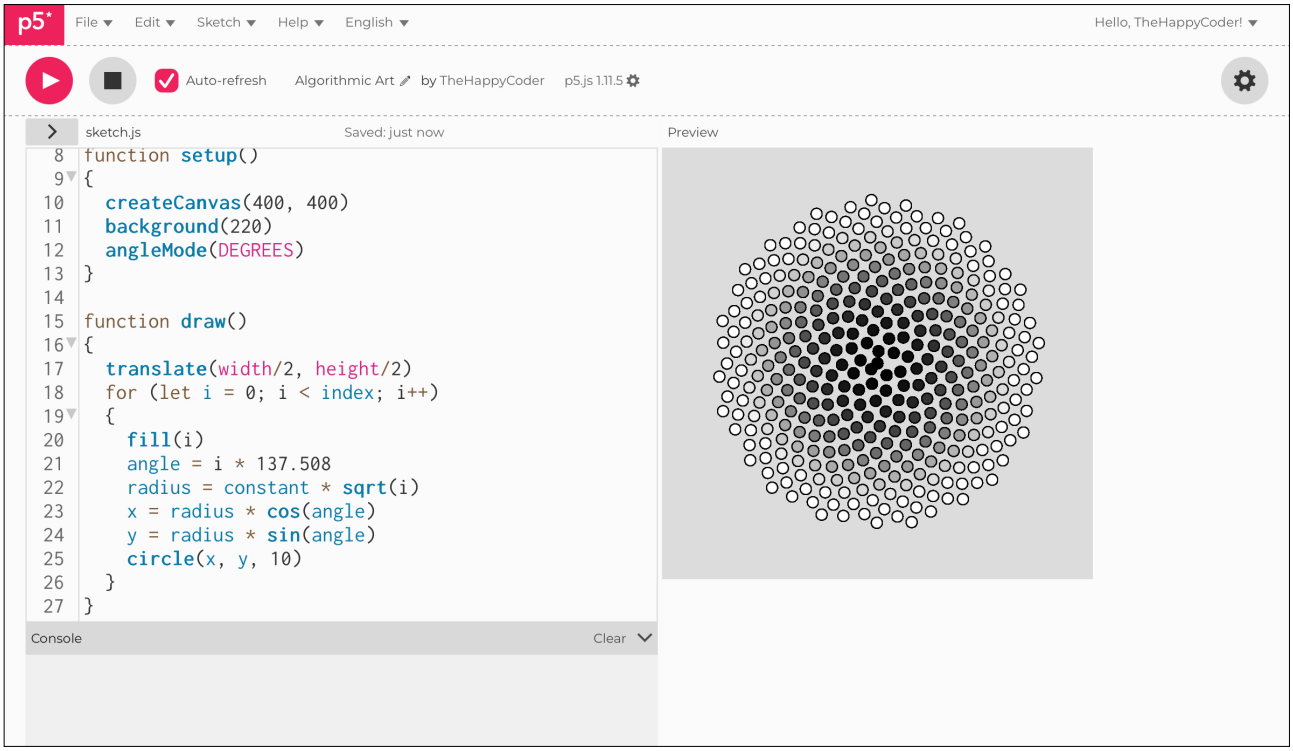
Mapping the colour to the index value.

```
let constant = 8
let radius = 100
let index = 360
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
}

function draw()
{
  translate(width/2, height/2)
  for (let i = 0; i < index; i++)
  {
    let colour = map(i, 0, index, 0, 255)
    fill(colour)
    angle = i * 137.508
    radius = constant * sqrt(i)
    x = radius * cos(angle)
    y = radius * sin(angle)
    circle(x, y, 10)
  }
}
```

Notes

Showing promise.

Figure B10.18



Sketch B10.19 colour HSB

Let us make it colour with HSB rather than RGB. RGB is the default mode, but HSB gives you different options.

```
let constant = 8
let radius = 100
let index = 360
let x = 0
let y = 0
let angle = 0

function setup()
{
  createCanvas(400, 400)
  background(220)
  angleMode(DEGREES)
  colorMode(HSB)
}

function draw()
{
  translate(width/2, height/2)
  for (let i = 0; i < index; i++)
  {
    let colour = map(i, 0, index, 0, 255)
    fill(colour, 100, 100)
    angle = i * 137.508
    radius = constant * sqrt(i)
    x = radius * cos(angle)
    y = radius * sin(angle)
    circle(x, y, 10)
  }
}
```

Notes

Now that is so much nicer.

Challenges

1. See what you can produce.
2. Play with the values.

Figure B10.19