

*Algorithmic
Art
Workbook #6
Classes and
Objects*



Table of Contents

MIT Licence	4
Workbook #6 Quick Content Summary	5
Module D Unit #1: functions, objects and classes	7
Sketch D1.1 starting sketch	8
Sketch D1.2 a single car	9
Sketch D1.3 moving the car	10
Sketch D1.4 it reappears	12
The car as a Function	14
Sketch D1.5 function single car	15
Sketch D1.6 function single car	17
The Car as an Object using Functions	19
Sketch D1.7 car as an object	20
Sketch D1.8 alternative car object	22
Introduction to Classes	24
Sketch D1.9 the constructor function	27
Sketch D1.10 the show() function	29
Sketch D1.11 the move() function	31
Sketch D1.12 creating a car	33
Sketch D1.13 to see it and move it	35
The Power of Classes	38
Sketch D1.14 car attributes	39
Sketch D1.15 a second car	42
Sketch D1.16 lots and lots of cars	44
Module D Unit #2: another class	47
Sketch D2.1 starting again	48
Sketch D2.2 a single ball	50
Sketch D2.3 adding a bit of gravity	52
Sketch D2.4 creating a ball class	54
Sketch D2.5 the constructor() function	56
Sketch D2.6 the show() function	58
Sketch D2.7 the move() function	60
Sketch D2.8 can we have our ball back, please	62
Sketch D2.9 many balls	64
Sketch D2.10 pulling the balls out	66
Sketch D2.11 random velocity	68
Sketch D2.12 refactoring	70
Module D Unit #3: array of objects	73
Sketch D3.1 bubbles	74
Sketch D3.2 classy bubbles	75

Sketch D3.3 time for an argument	76
Sketch D3.4 drawing the bubbles	78
Sketch D3.5 random motion	80
Sketch D3.6 many bubbles	82
Sketch D3.7 more bubbles	84
Sketch D3.8 spacing them out	86
Sketch D3.9 random	88
Sketch D3.10 the shimmering fog	90
Sketch D3.11 simpler times	92
Sketch D3.12 mouse click	94
Sketch D3.13 when push comes to shove	96
Sketch D3.14 a bit of a drag	98



MIT Licence

Copyright ©2026 Warren George

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Brief summary:

Permissions (what you can do)

Commercial use
Modification
Distribution
Private use

Limitations (what is not covered)

Liability
Warranty



Workbook #6 Quick Content Summary

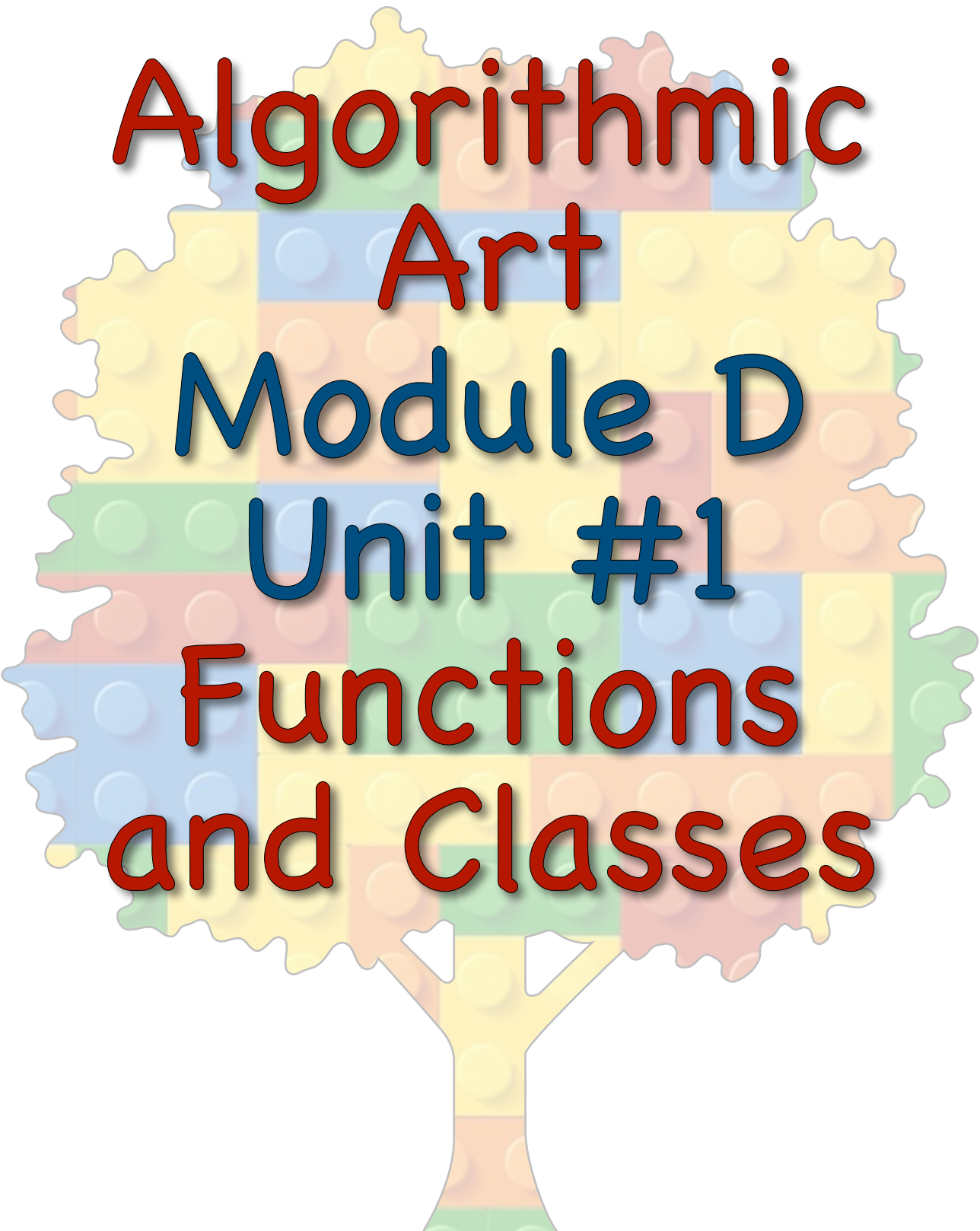
This unit looks a bit academic as well as the last one, covering classes as well as more in-depth work on arrays. I hope it will become clear to you as you work through this unit how powerful some of these concepts are and what you could do with them.

In coding, there are ever-increasing layers of complexity. They are worth learning as they provide more ways to create and develop ideas that you may have.

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 the Chrome browser.

Use version **2.2.3** of **p5.js** for the moment until the latest versions **2.3.x** become less buggy (as of September 2026).



Algorithmic Art Module D Unit #1 Functions and Classes



Module D Unit #1: functions, objects and classes

This next section looks at coding with functions, objects, and classes. It demonstrates the different ways you can code the same effect using different approaches. The context we will use is of a vehicle or vehicles moving either across the canvas.

We can draw the vehicle with a `show()` function and its movement with a `move()` function. These are simple examples to highlight the differences to give you a flavour of what that might look like.



Sketch D1.1 starting sketch

We start a new sketch, as usual.

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

function draw()
{
  background(220)
}
```

Notes

Our ever friendly starting sketch



Sketch D1.2 a single car

We create our car as a simple rectangle, starting at the left-hand edge of the canvas.

```
let x = 0

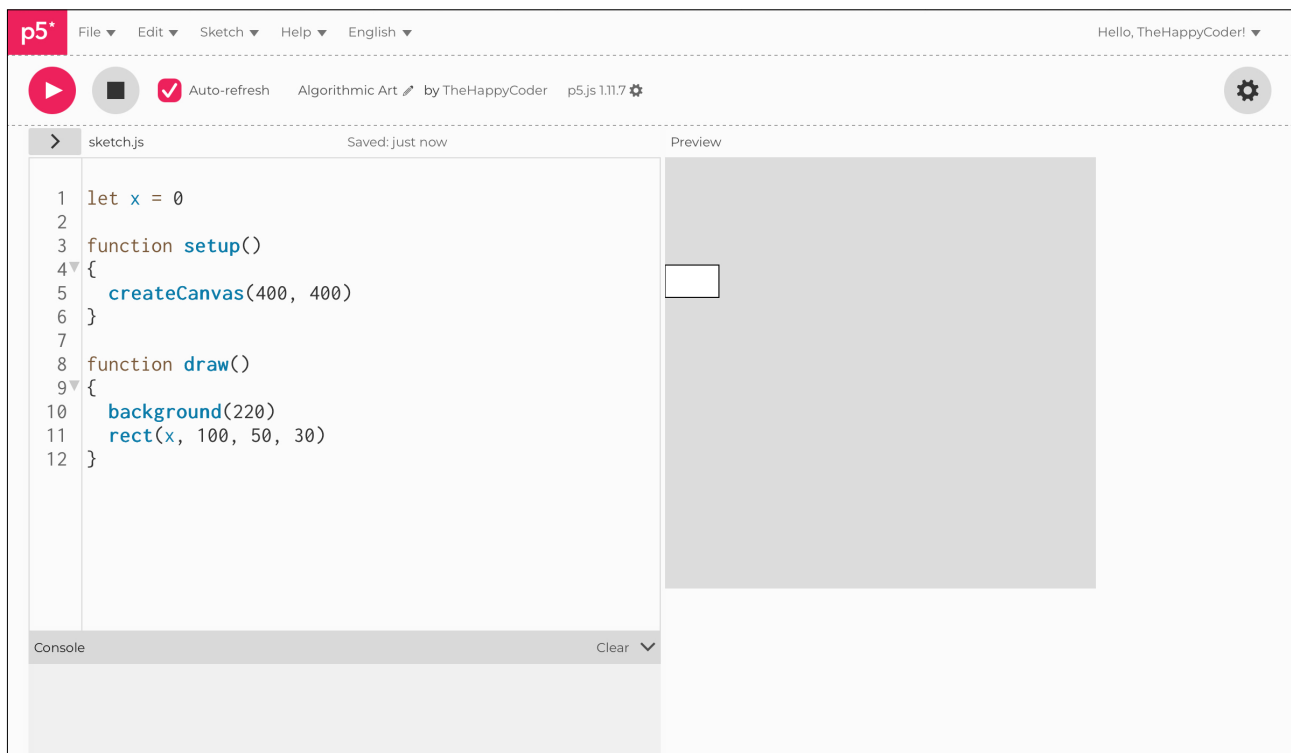
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  rect(x, 100, 50, 30)
}
```

Notes

A single stationary car (rectangle).

Figure D1.2





Sketch D1.3 moving the car

Now we start the car moving across the canvas.

```
let x = 0

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

function draw()
{
  background(220)
  rect(x, 100, 50, 30)
  x += 1
}
```

Notes

It moves slowly across the canvas and disappears from the right-hand edge of the canvas, never to be seen again.

Challenge

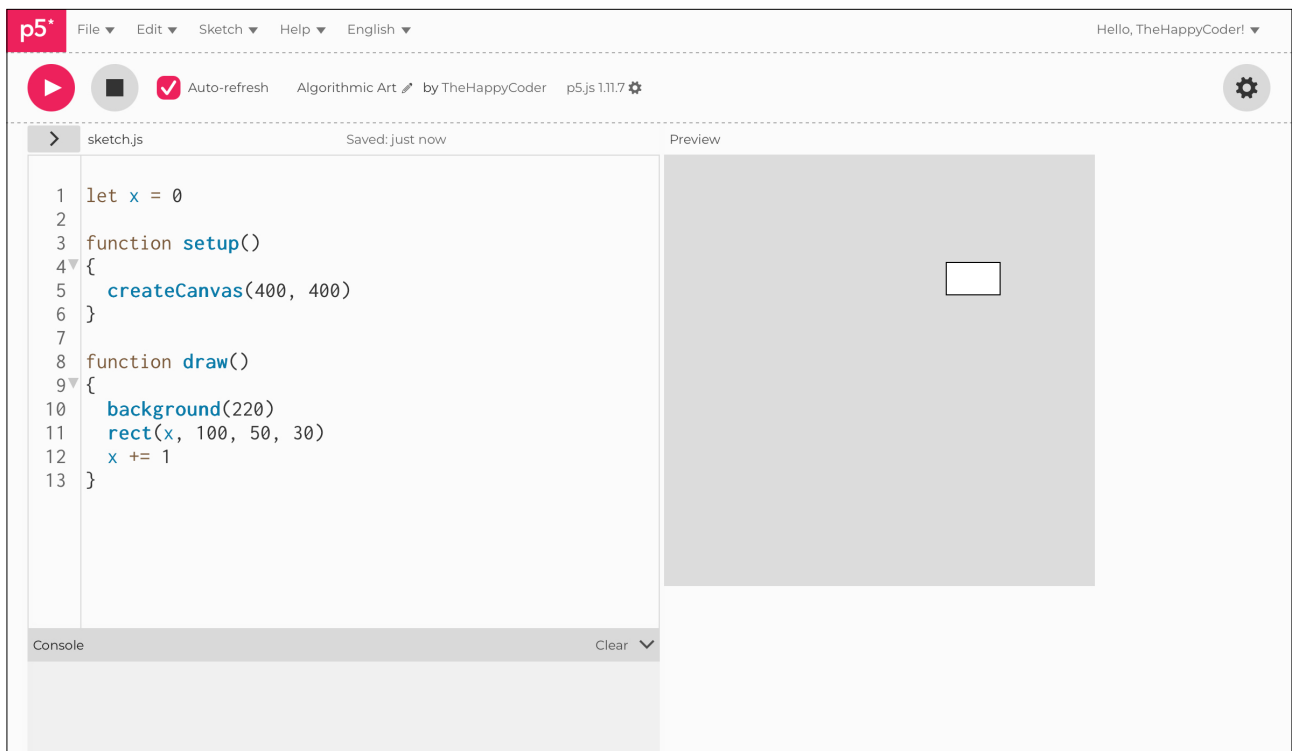
Make it move faster.

Code Explanation

```
x += 1
```

This adds 1 to x on each iteration.

Figure D1.3





Sketch D1.4 it reappears

The car now reappears on the left-hand edge and off it goes again.

```
let x = 0

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

function draw()
{
  background(220)
  rect(x, 100, 50, 30)
  x += 1
  if (x > width)
  {
    x = -50
  }
}
```

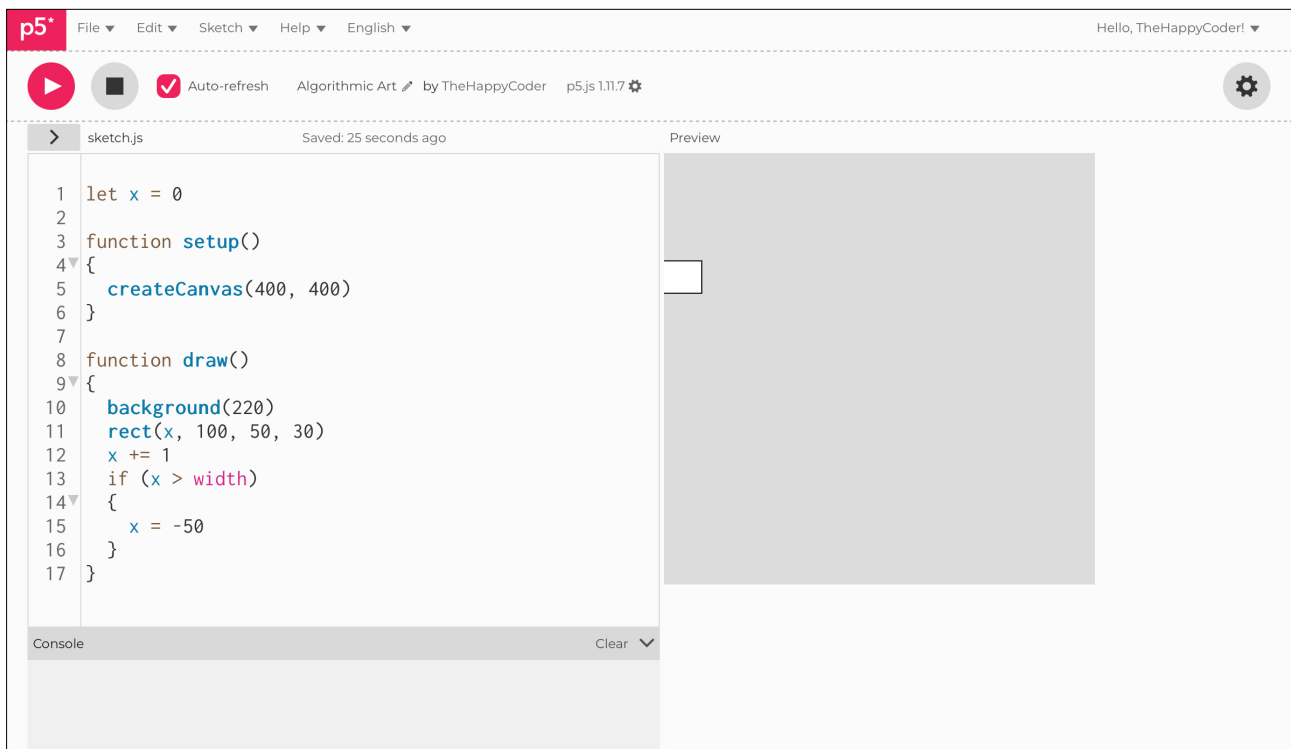
Notes

We have **x** as **-50**, so it looks like it is seamlessly continuous.

Code Explanation

<code>if (x > width)</code>	Checks to see if it has reached the edge of the canvas.
<code>x = -50</code>	Returns the x value to -50 if the car has gone off the edge of the canvas.

Figure D1.4





The car as a Function

We can express the same thing as before, but this time we use functions, two of them to describe the car and to describe the motion. This means we have the `setup()` function as before, we keep the `draw()` function (empty for now), and add the other two functions called `show()` and `move()`, putting a lot of the stuff in those new functions.

I am using a very simple example here, but bear with me as we will build on this concept when we introduce classes later.



Sketch D1.5 function single car

! You will not even get a canvas, so don't panic!

We have moved the code that was in the `draw()` function and split it between the two new functions: `show()` and `move()`. We have used the same code as in the previous sketch, just rearranged it.

```
let x = 0

function setup()
{
  createCanvas(440, 400)
}

function draw()
{
  // empty line of code
}

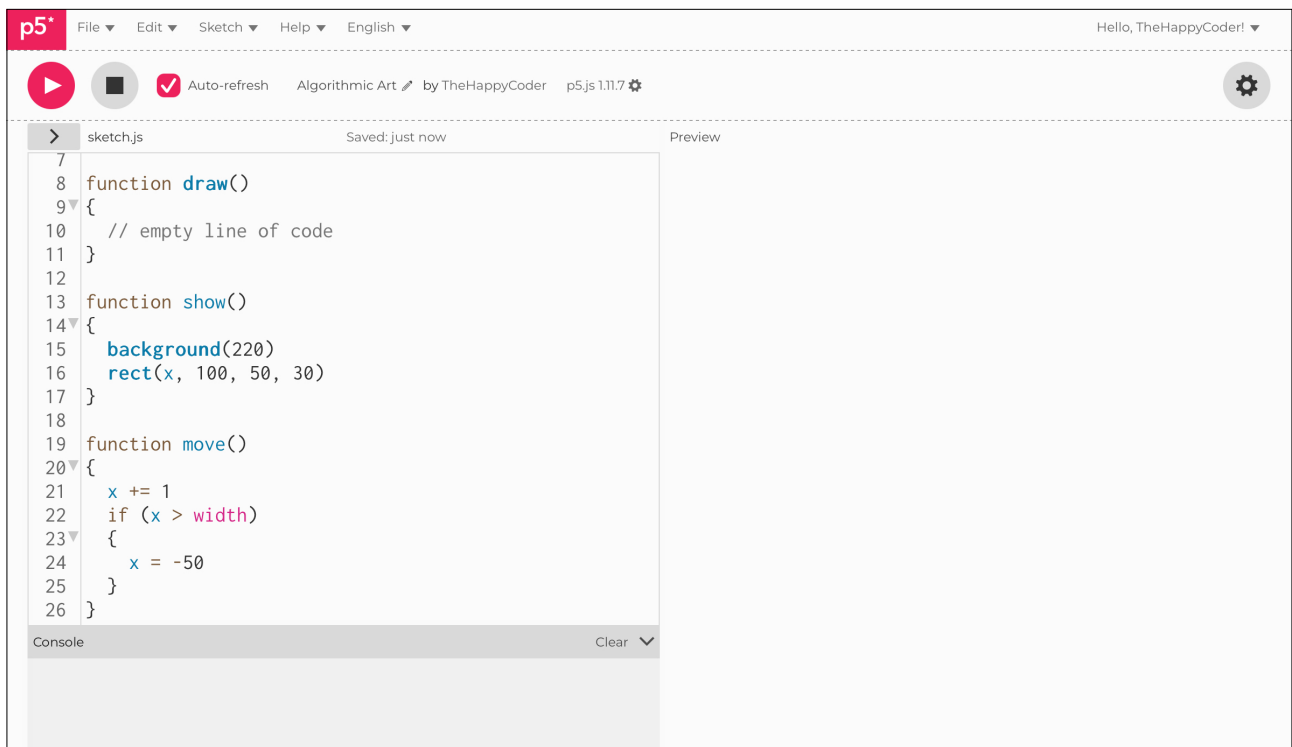
function show()
{
  background(220)
  rect(x, 100, 50, 30)
}

function move()
{
  x += 1
  if (x > width)
  {
    x = -50
  }
}
```

Notes

Nothing to see. You can just cut and paste to save time; however, you will notice that you get nothing, not even a canvas.

Figure D1.5





Sketch D1.6 function single car

To get the two new functions to do anything, we need to call them from inside the `draw()` function, and we do it as shown below.

```
let x = 0

function setup()
{
  createCanvas(440, 400)
}

function draw()
{
  show()
  move()
}

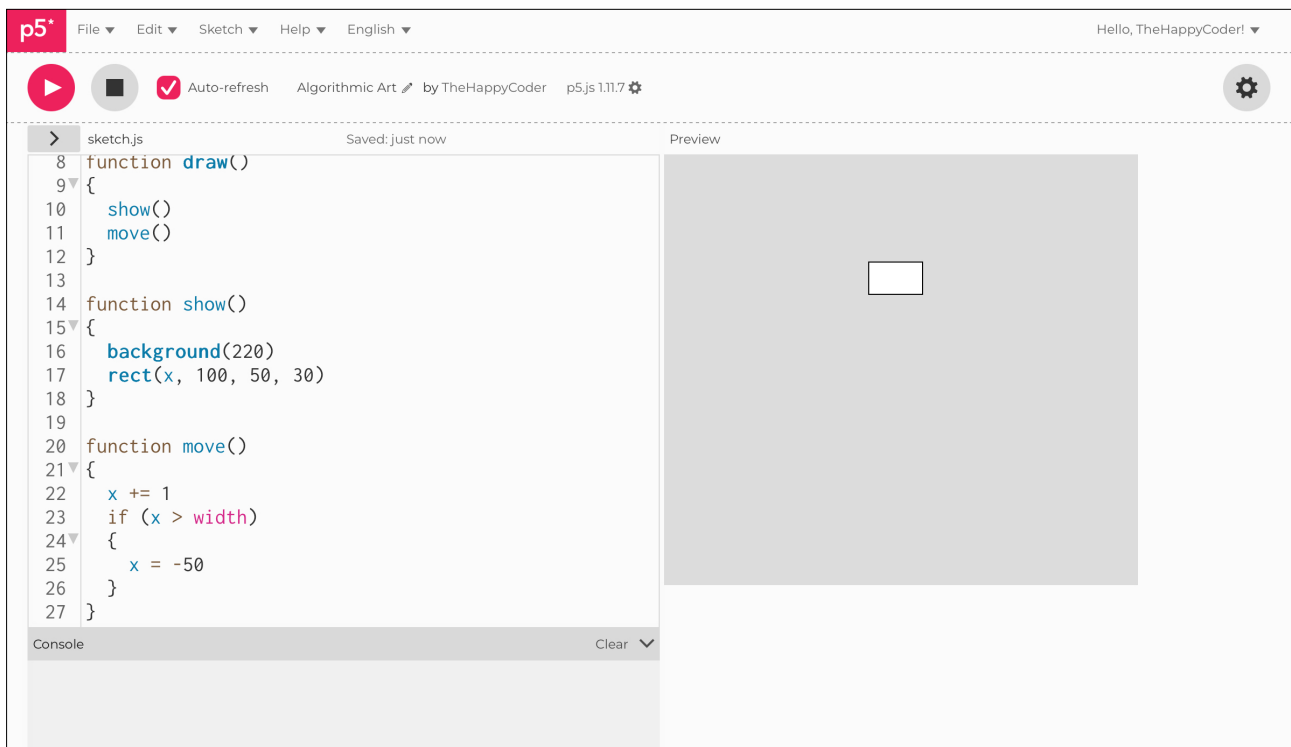
function show()
{
  background(220)
  rect(x, 100, 50, 30)
}

function move()
{
  x += 1
  if (x > width)
  {
    x = -50
  }
}
```

Notes

Now we are back where we started, but let's not stop there; there is yet another way we can do this even before we introduce classes.

Figure D1.6





The Car as an Object using Functions

This exercise is another way of doing the same thing. I include it because it shows the concept of objects in relation to functions. We could easily create two cars, but it would mean doubling all the code for each car. This is another reason where classes come into their own, but we are getting ahead of ourselves here.



Sketch D1.7 car as an object

! Start a new sketch, a basic sketch with extra bits.

We have added the car as an object; notice the similarity to our earlier sketch, but now we have to give it a name. In this case, we call it `car`. The `.x` is the bit that is noticeable.

```
let car = {x: 0}

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

function draw()
{
  background(220)
  rect(car.x, 100, 50, 30)
  car.x += 1
  if (car.x > width)
  {
    car.x = -50
  }
}
```

Notes

Everything behaves just as before. Look at the code carefully and see how you code the car as an object rather than just as a rectangle. Below is a more detailed explanation of the code; it isn't as scary as it might look.

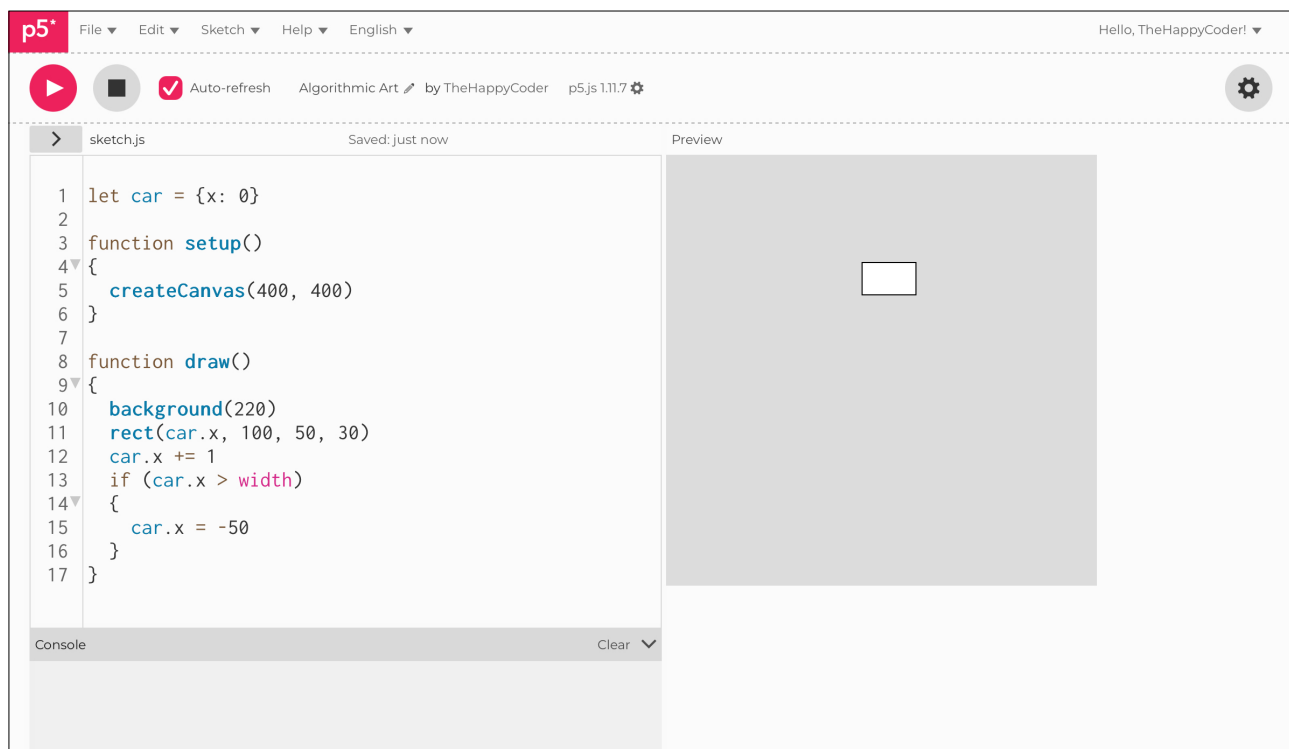
Challenges

1. Give it a `y` component
2. Introduce the `show()` and `move()` functions like we did previously (if you are struggling then see next sketch)

Code Explanation

<code>let car = {x: 0}</code>	We initialise the x component of the car object to 0.
<code>rect(car.x, 100, 50, 30)</code>	The x component of the car object.
<code>car.x += 1</code>	Incrementing the x component by 1 on each iteration.
<code>if (car.x > width)</code>	Check when the car has gone off the edge of the canvas.
<code>car.x = -50</code>	The x component is re-initialised to -50.

Figure D1.7





Sketch D1.8 alternative car object

Now we can use the functions `show()` and `move()` as well as introducing a `y` component. The background can go into `draw()` or `show()`.

```
let car = {x: 0, y: 100}

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

function draw()
{
  background(220)
  show()
  move()
}

function show()
{
  rect(car.x, car.y, 50, 30)
}

function move()
{
  car.x += 1
  if (car.x > width)
  {
    car.x = -50
  }
}
```

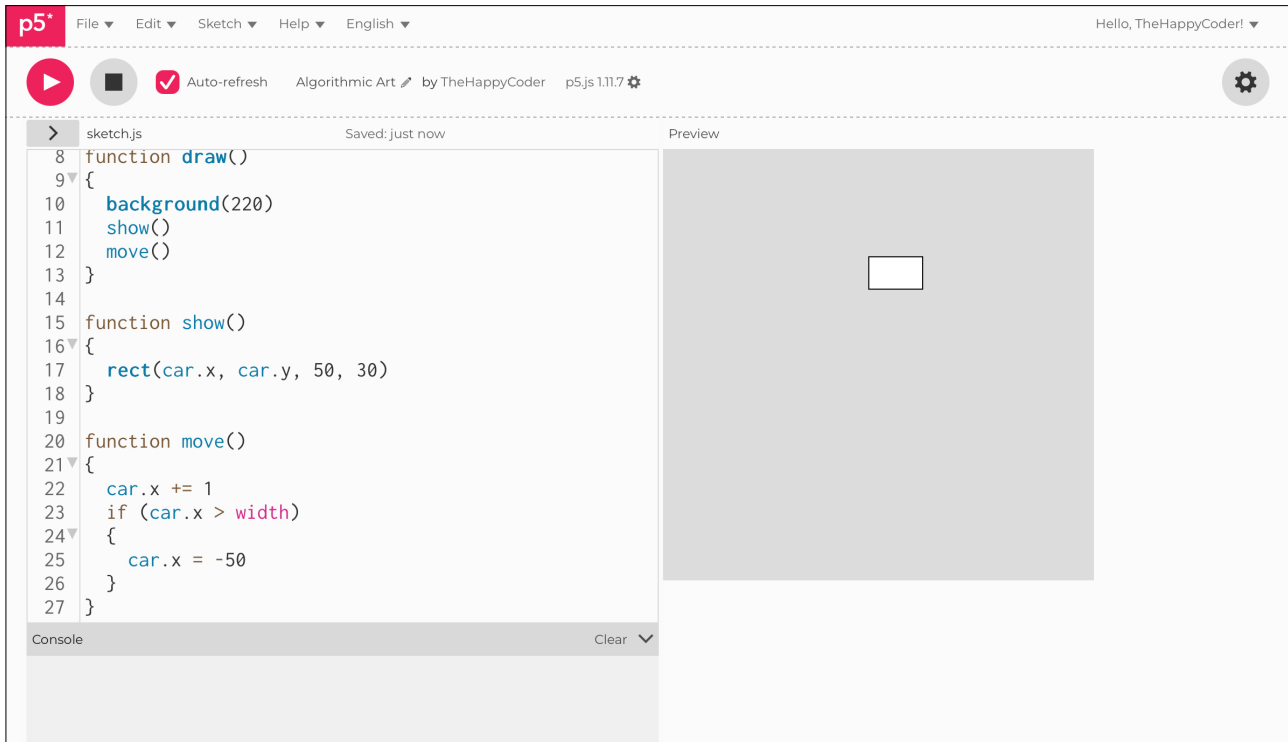
Notes

Looks quite elegant in my opinion; it should be exactly the same as before.

Code Explanation

<code>let car = {x: 0, y: 100}</code>	Adding the y component to the car object.
<code>rect(car.x, car.y, 50, 30)</code>	Adding the y component to the rectangle drawn.

Figure D1.8





Introduction to Classes

Using classes is a common way for coders to organise their code. It is not essential, as you could do the same thing without using classes, but it is a very powerful and useful approach and one worth investing the time in understanding.

It does take a bit of getting used to. I will try to illustrate this with a simple example. Imagine you have a template (or blueprint) to build a car. You, as a consumer, want some choice. The colour, the number of doors, engine size, interior style, and so on. A class is like the basic template. When you order a new car, they don't ask if you want doors, seats, a steering wheel, windows, etc. They come as standard.

A class will have the basics and the options. So that when they make 10,000 cars, they can all be slightly different depending on what the customer wants. This is a very limited comparison, but you will see that you can create lots of cars that all behave slightly differently. In our first example, we will do just that with a sort of car.

In the diagram below ([fig.1](#)), you will see that the class is given a name. It is usual to start the class name with a capital letter. Also, there are three functions in the example below. You can have as many functions as you like and can call them anything you like. You can see the `show()` and `move()` functions we had before, but you can have any number of functions.

The first function I use is called the `constructor()` function. This is just the usual name given to it. This is where we hold the information about any car we are going to build. Because it is a sort of template (or blueprint) where we can make as many cars as we want, we prefix any variable with the word `this`; for instance, the colour would be `this.colour`, or the starting position will be `this.x` and `this.y`, and so on.

The basic structure of the main sketch is demonstrated in [fig.2](#). Where you create the car or cars from the class and call the functions from within the class.

Figure 1: class structure

```
class Car
{
    constructor()
    {
        // this.something
    }

    show()
    {
        // what it looks like
    }

    move()
    {
        // how it will move/change
    }
}
```

Figure 2: main elements in sketch

```
let car

function setup()
{
  createCanvas(400, 400)
  car = new Car()
}

function draw()
{
  background(220)
  car.show()
  car.move()
}
```



Sketch D1.9 the constructor function

! New sketch, a basic plus the addition of a class function.

We start with our basic sketch and add something called a **class** called **Car**. In that class, we have a **constructor()** function. This function has four elements that give us details about the car: its **colour**, its **x** position, its **y** position, and its **velocity**. This first example will not reveal the full power of using classes but introduces you to the key concepts.

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

function draw()
{
  background(220)
}

class Car
{
  constructor()
  {
    this.colour = 255
    this.x = 0
    this.y = 100
    this.velocity = 1
  }
}
```

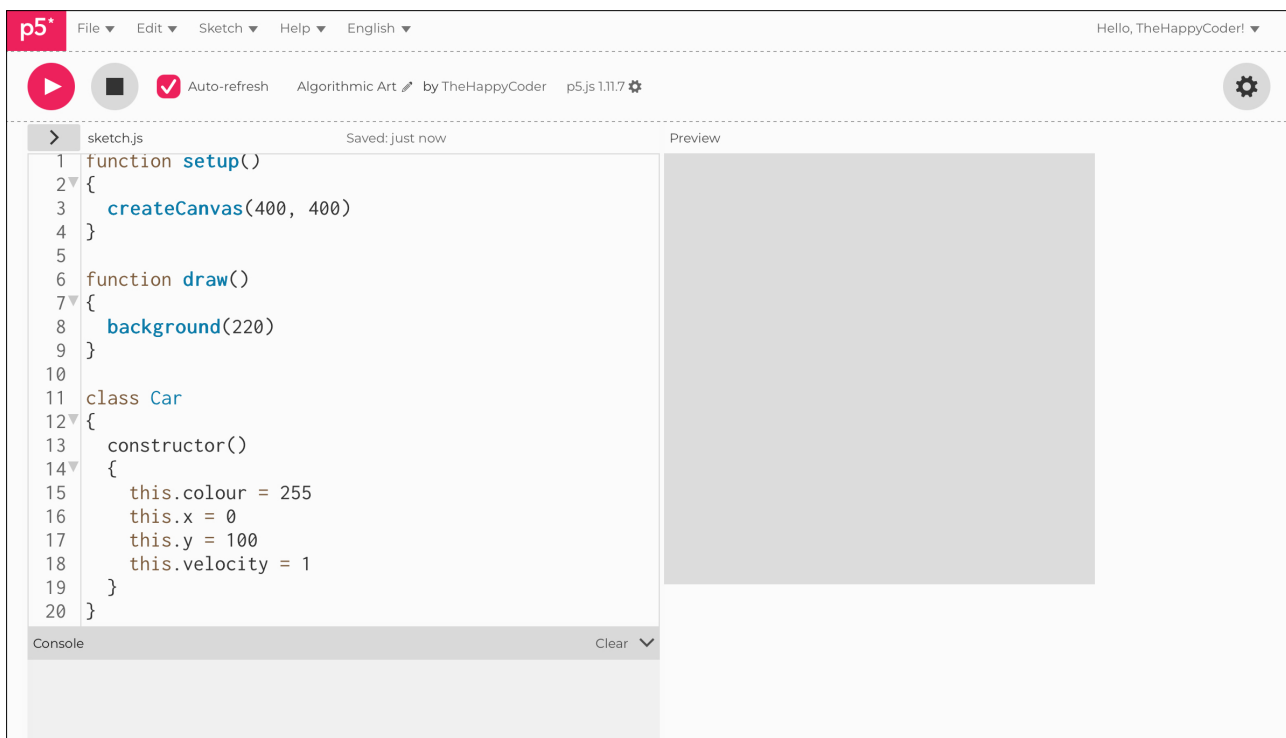
Notes

When we give attributes to an object in a class, we always use **this.** before the attribute. There is nothing to see at this point.

Code Explanation

<code>this.colour = 255</code>	For any car, we define its colour.
<code>this.x = 0</code>	For any car, we define its x position.
<code>this.y = 100</code>	For any car, we define its y position.
<code>this.velocity = 1</code>	For any car, we define its velocity.

Figure D1.9





Sketch D1.10 the show() function

In the `show()` function, we will describe what the car will look like.

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

function draw()
{
  background(220)
}

class Car
{
  constructor()
  {
    this.colour = 255
    this.x = 0
    this.y = 100
    this.velocity = 1
  }

  show()
  {
    fill(this.colour)
    rect(this.x, this.y, 50, 30)
  }
}
```

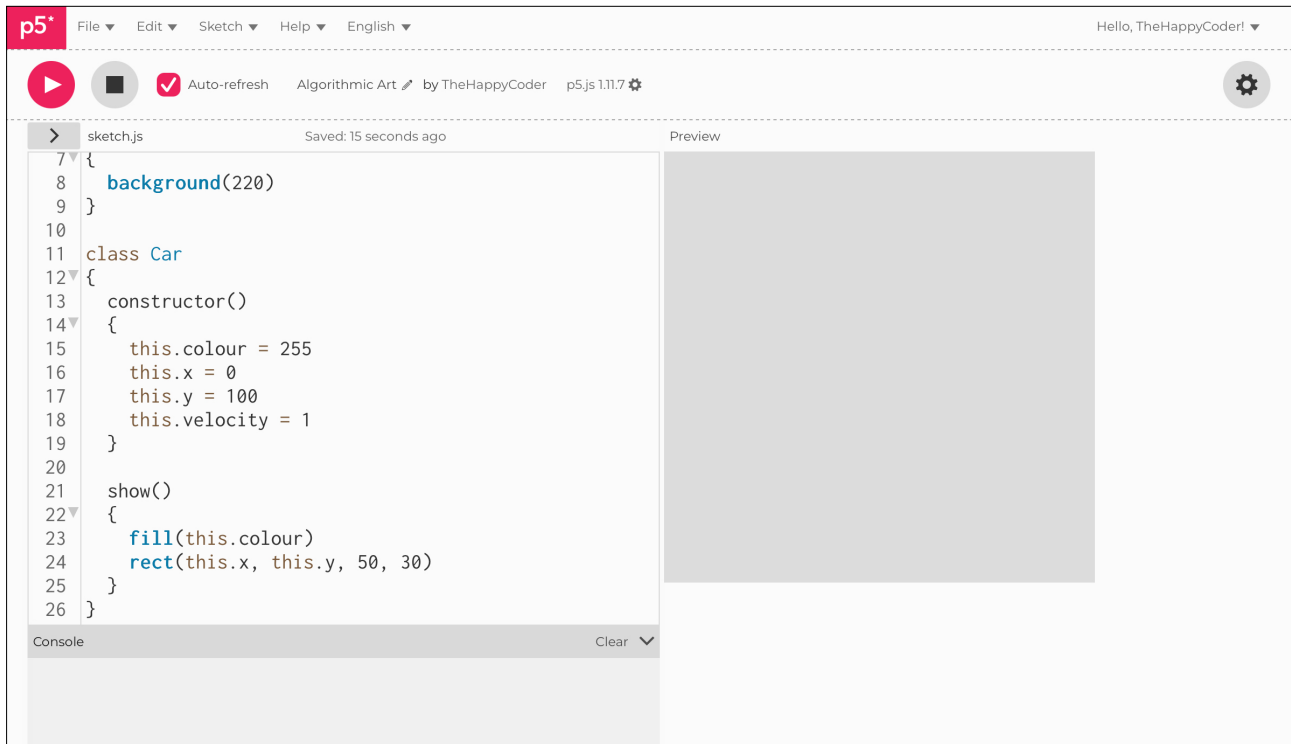
Notes

It pulls the information from the `constructor()` function. Still nothing to see yet.

Code Explanation

<code>fill(this.colour)</code>	This will fill it with white (255).
<code>rect(this.x, this.y, 50, 30)</code>	Creates a rectangle <code>rect(0, 100, 50, 30)</code> .

Figure D1.10





Sketch D1.11 the move() function

Next, we describe how the car is going to move with the `move()` function inside the `Car` class.

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

function draw()
{
  background(220)
}

class Car
{
  constructor()
  {
    this.colour = 255
    this.x = 0
    this.y = 100
    this.velocity = 1
  }

  show()
  {
    fill(this.colour)
    rect(this.x, this.y, 50, 30)
  }

  move()
  {
    this.x += this.velocity
    if (this.x > width)
    {
      this.x = -50
    }
  }
}
```

```
}
```

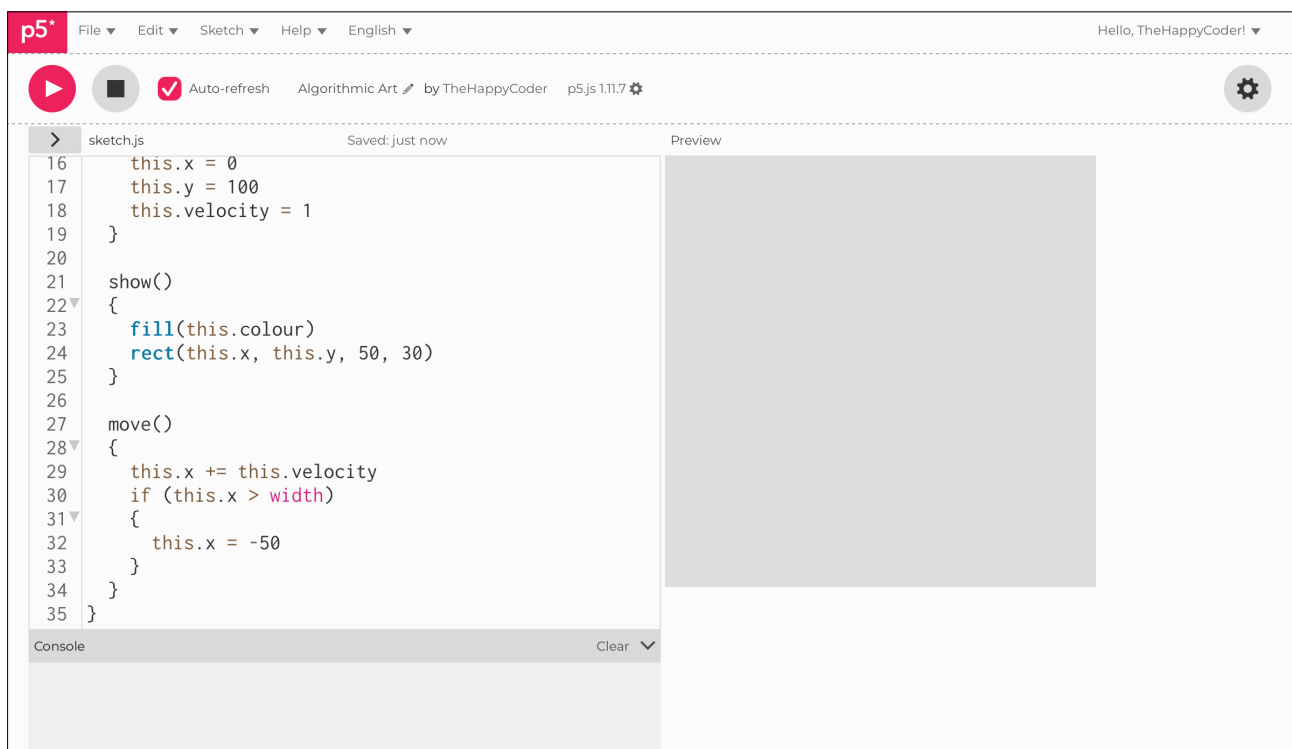
Notes

This is exactly the same as with the previous examples of a moving car. However, we created a variable for the **velocity** rather than just having a value (1). This allows us to alter it later. As before, still nothing to see.

Code Explanation

<code>this.x += this.velocity</code>	For each car, we add the velocity.
<code>if (this.x > width)</code>	If a car reaches the edge of the canvas...
<code>this.x = -50</code>	Return that car back to the left-hand edge.

Figure D1.11





Sketch D1.12 creating a car

To create a **car**, we first give this car a name. Then, in **setup()**, we create a **new Car** from the class as a template. We currently have fixed values such as **colour**, **x**, **y**, and **velocity**.

```
let car

function setup()
{
  createCanvas(400, 400)
  car = new Car()
}

function draw()
{
  background(220)
}

class Car
{
  constructor()
  {
    this.colour = 255
    this.x = 0
    this.y = 100
    this.velocity = 1
  }

  show()
  {
    fill(this.colour)
    rect(this.x, this.y, 50, 30)
  }

  move()
  {
    this.x += this.velocity
    if (this.x > width)
```

```

    {
      this.x = -50
    }
  }
}

```

Notes

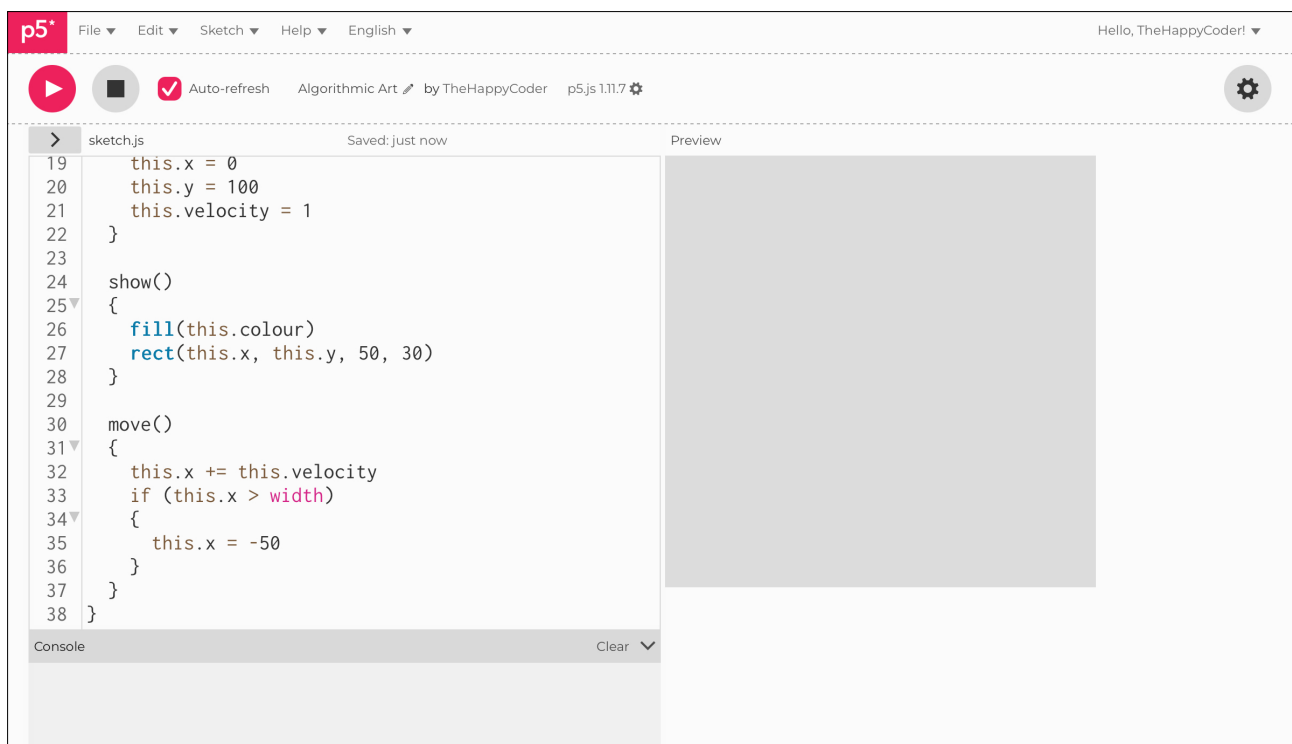
Be aware that the variable name for the car is a lowercase **c**, and the name of the class is an uppercase **C**. They both have the same name, which I admit is a little confusing, but they are totally separate entities. One is a variable name, the other is a class name. Still nothing to see here.

Code Explanation

```
car = new Car()
```

Creates a new car

Figure D1.12





Sketch D1.13 to see it and move it

In order to see the car, we have to call the `show()` function, and to move the car, we have to call the `move()` function, both in the `draw()` function. We ascribe these two functions to the new car we have created, called `car`.

```
let car

function setup()
{
  createCanvas(400, 400)
  car = new Car()
}

function draw()
{
  background(220)
  car.show()
  car.move()
}

class Car
{
  constructor()
  {
    this.colour = 255
    this.x = 0
    this.y = 100
    this.velocity = 1
  }

  show()
  {
    fill(this.colour)
    rect(this.x, this.y, 50, 30)
  }

  move()
  {
```

```
    this.x += this.velocity
    if (this.x > width)
    {
        this.x = -50
    }
}
}
```

Notes

Finally, we get to see the car and watch it move.

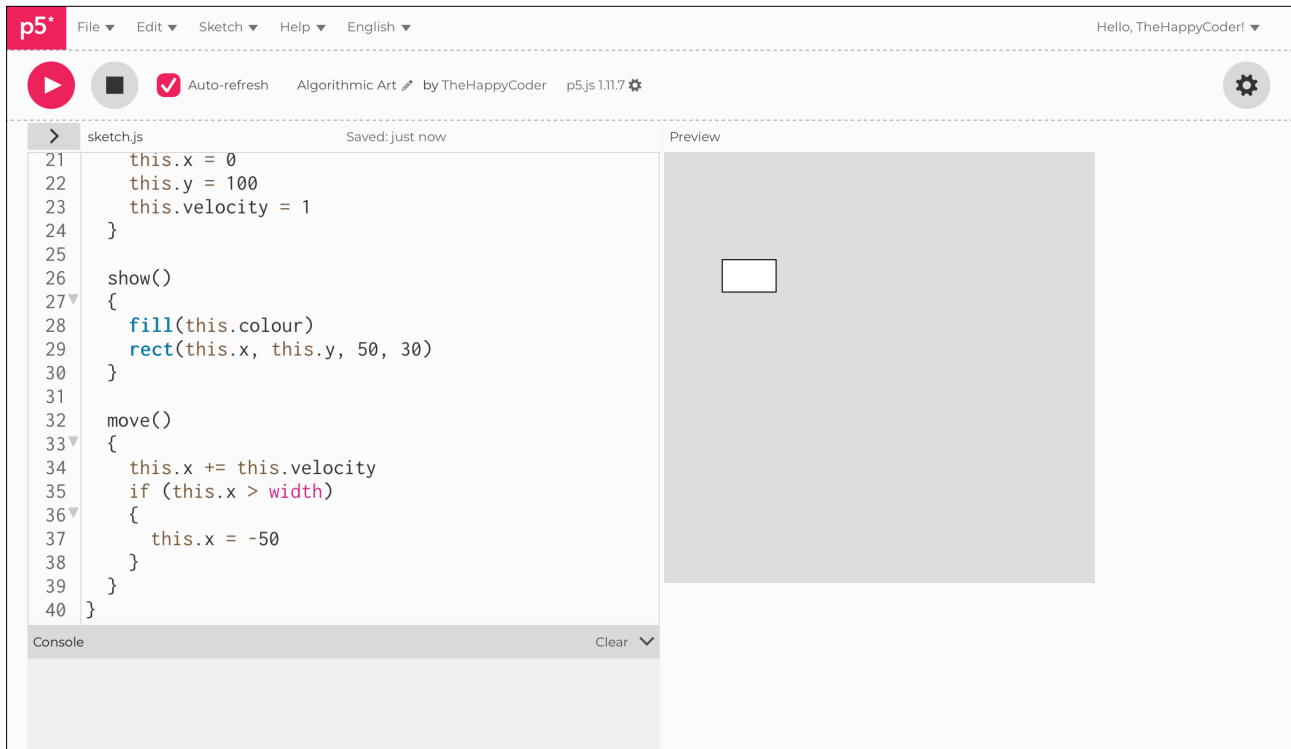
Challenges

1. Change the **colour**
2. Change the **x** value
3. Change the **y** value
4. Change the **velocity**
5. Change the name of the variable to **myCar**
6. Change the name of the **class**
7. Change the name of the **constructor()**, **show()** and **move()** functions

Code Explanation

<code>car.show()</code>	For this car, we show it according to the <code>show()</code> function.
<code>car.move()</code>	For this car, we move it according to the <code>move()</code> function.

Figure D1.13





The Power of Classes

In the following sections, we will consider what we can do with classes which makes all the trouble of creating them worthwhile. This is evident when we want hundreds of them, where each one can be created separately, independently.



Sketch D1.14 car attributes

When we create the car, we can specify its attributes rather than hard-code them in the `constructor()` function. We have given the car the same values as before. They become the arguments in the `constructor()` function: `colour`, `x`, `y`, and `velocity`. This is more like a template where you can now specify what you want.

```
let car

function setup()
{
  createCanvas(400, 400)
  car = new Car(255, 0, 100, 1)
}

function draw()
{
  background(220)
  car.show()
  car.move()
}

class Car
{
  constructor(colour, x, y, velocity)
  {
    this.colour = colour
    this.x = x
    this.y = y
    this.velocity = velocity
  }

  show()
  {
    fill(this.colour)
    rect(this.x, this.y, 50, 30)
  }

  move()
```

```

{
  this.x += this.velocity
  if (this.x > width)
  {
    this.x = -50
  }
}
}

```

Notes

The result is exactly the same as before because we have specified the same features of our car. The beauty of this is that we can create a second (or more) car with different features.

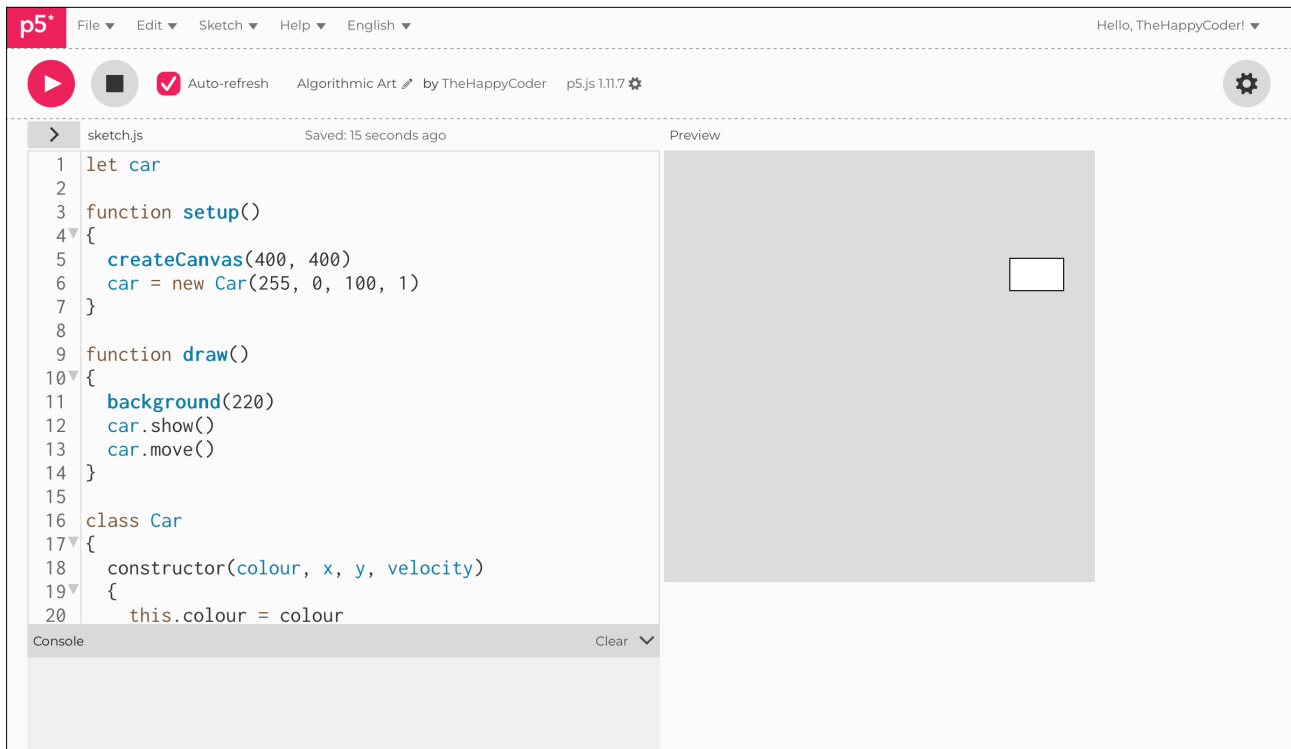
Challenge

Change the values/features of the car.

Code Explanation

<code>car = new Car(255, 0, 100, 1)</code>	We give this car some attributes.
<code>constructor(colour, x, y, velocity)</code>	The attributes are received as arguments in the <code>constructor()</code> function.
<code>this.colour = colour</code>	This car has the colour argument.
<code>this.x = x</code>	This car has the x position argument.
<code>this.y = y</code>	This car has the y position argument.
<code>this.velocity = velocity</code>	This car has the velocity argument.

Figure D1.14





Sketch D1.15 a second car

We add a second car and give it different features.

```
let car
let car2

function setup()
{
  createCanvas(400, 400)
  car = new Car(255, 0, 100, 1)
  car2 = new Car(55, 0, 300, 2)
}

function draw()
{
  background(220)
  car.show()
  car.move()
  car2.show()
  car2.move()
}

class Car
{
  constructor(colour, x, y, velocity)
  {
    this.colour = colour
    this.x = x
    this.y = y
    this.velocity = velocity
  }

  show()
  {
    fill(this.colour)
    rect(this.x, this.y, 50, 30)
  }
}
```

```

move()
{
  this.x += this.velocity
  if (this.x > width)
  {
    this.x = -50
  }
}
}
}

```

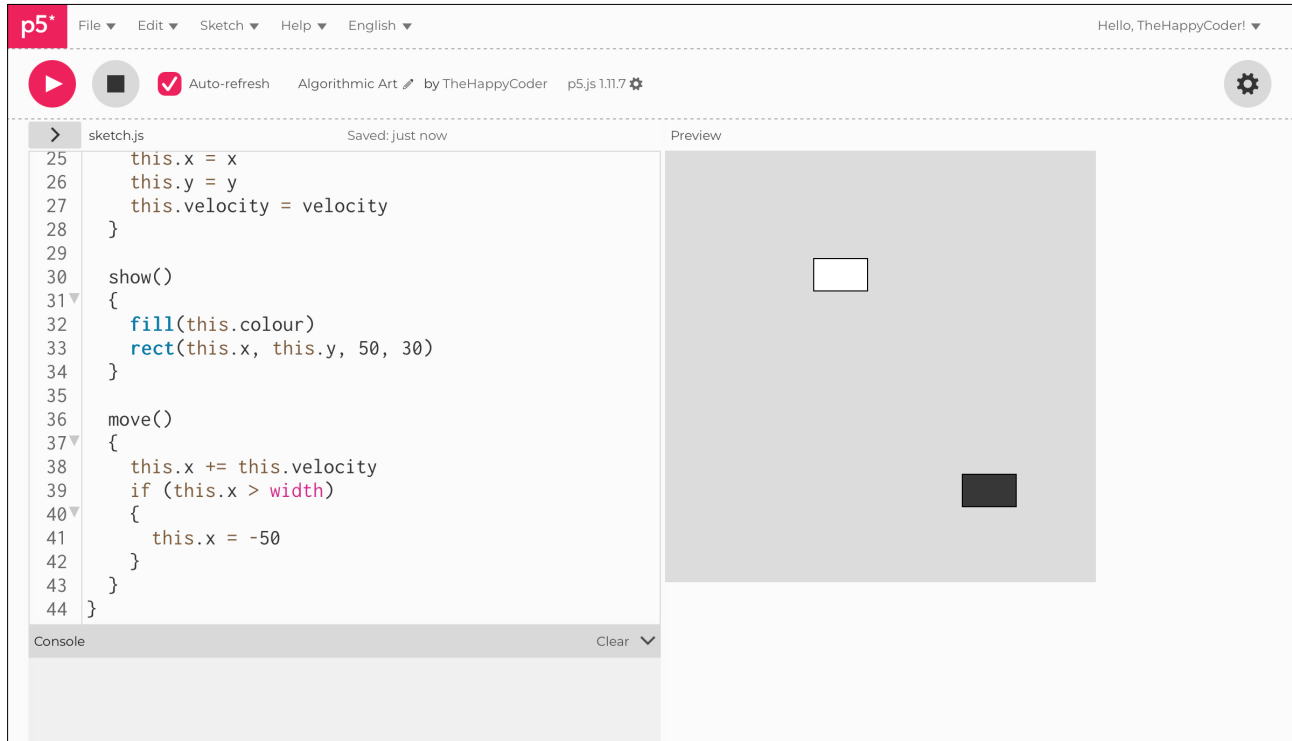
Notes

With just a few lines of code, we have created a second car. You can see the simple logic.

Challenge

Add a third car.

Figure D1.15





Sketch D1.16 lots and lots of cars

Here is a quick peek at what we could do with loops to draw lots of cars. We have covered **arrays** and **for()** loops before. We create an array of cars and cycle through them with random values for all the features (except **x**). We then cycle through the array of cars, show and move them. All this is done in the **setup()** and **draw()** functions; we don't touch the **Car** class!

```
let car = []

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < 10; i++)
  {
    car[i] = new Car(random(255), 0, random(400), random(1, 5))
  }
}

function draw()
{
  background(220)
  for (let i = 0; i < car.length; i++)
  {
    car[i].show()
    car[i].move()
  }
}

class Car
{
  constructor(colour, x, y, velocity)
  {
    this.colour = colour
    this.x = x
    this.y = y
    this.velocity = velocity
  }

  show()
}
```

```

{
  fill(this.colour)
  rect(this.x, this.y, 50, 30)
}

move()
{
  this.x += this.velocity
  if (this.x > width)
  {
    this.x = -50
  }
}
}

```

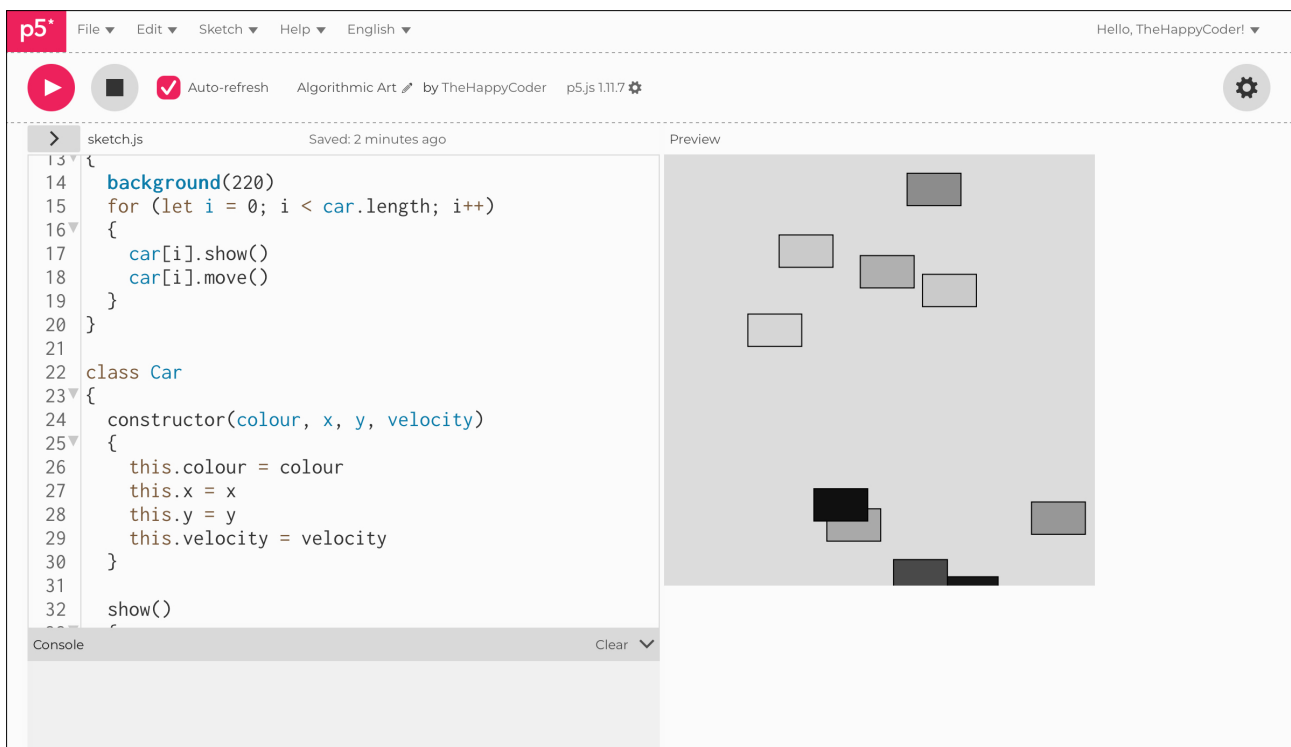
Notes

I think that is pretty elegant!

Challenge

Just have a play.

Figure D1.16



Algorithmic Art

Module D

Unit #2

In Another Class



Module D Unit #2: another class

This is another simple example of using classes to create something. The previous unit introduced classes as well as comparing them with functions and objects. Here we add a few more features to create an explosion of balls cascading or raining down. We will draw on this when we create a fireworks display.



Sketch D2.1 starting again

! Starting a new sketch

A circle (ball) at (x, y) coordinates.

```
let x = 0
let y = 200

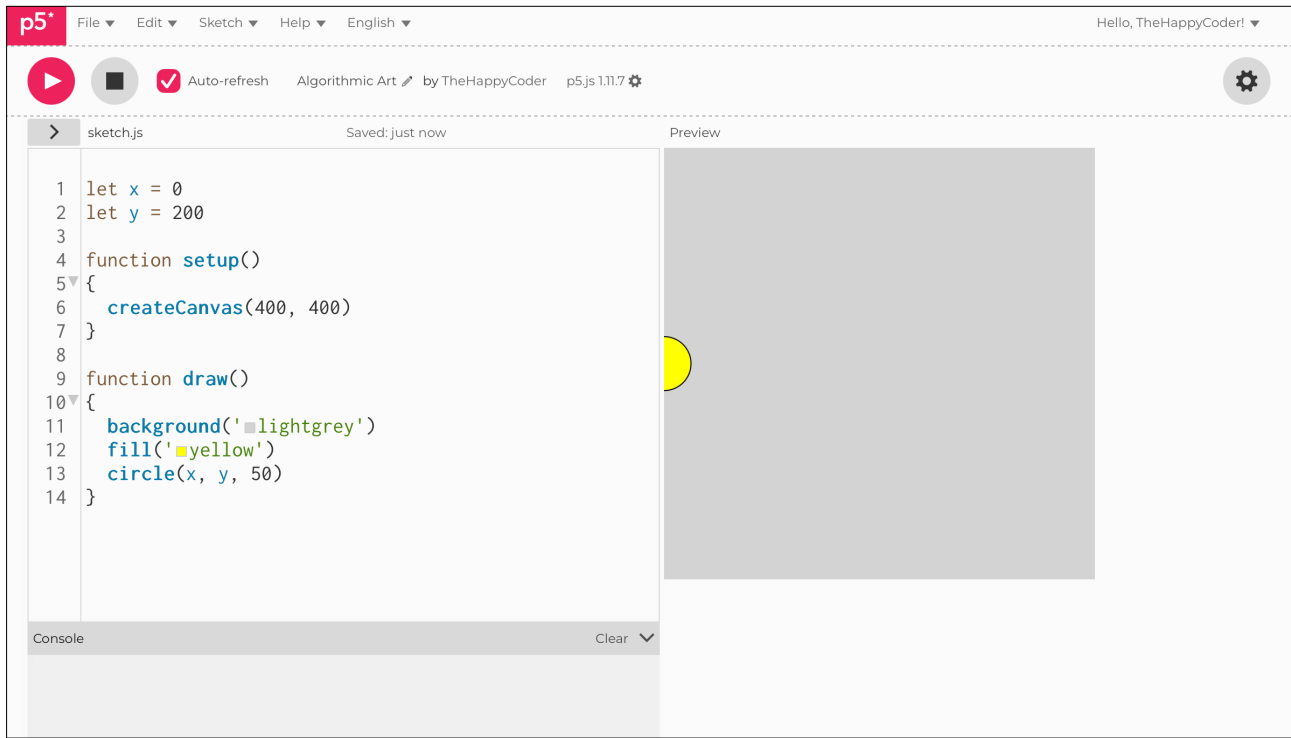
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background('lightgrey')
  fill('yellow')
  circle(x, y, 50)
}
```

Notes

A static circle.

Figure D2.1





Sketch D2.2 a single ball

Now a ball moving across the canvas with a **x** velocity of **3** and a **y** velocity of **0**.

```
let x = 0
let y = 200
let x_velocity = 3
let y_velocity = 0

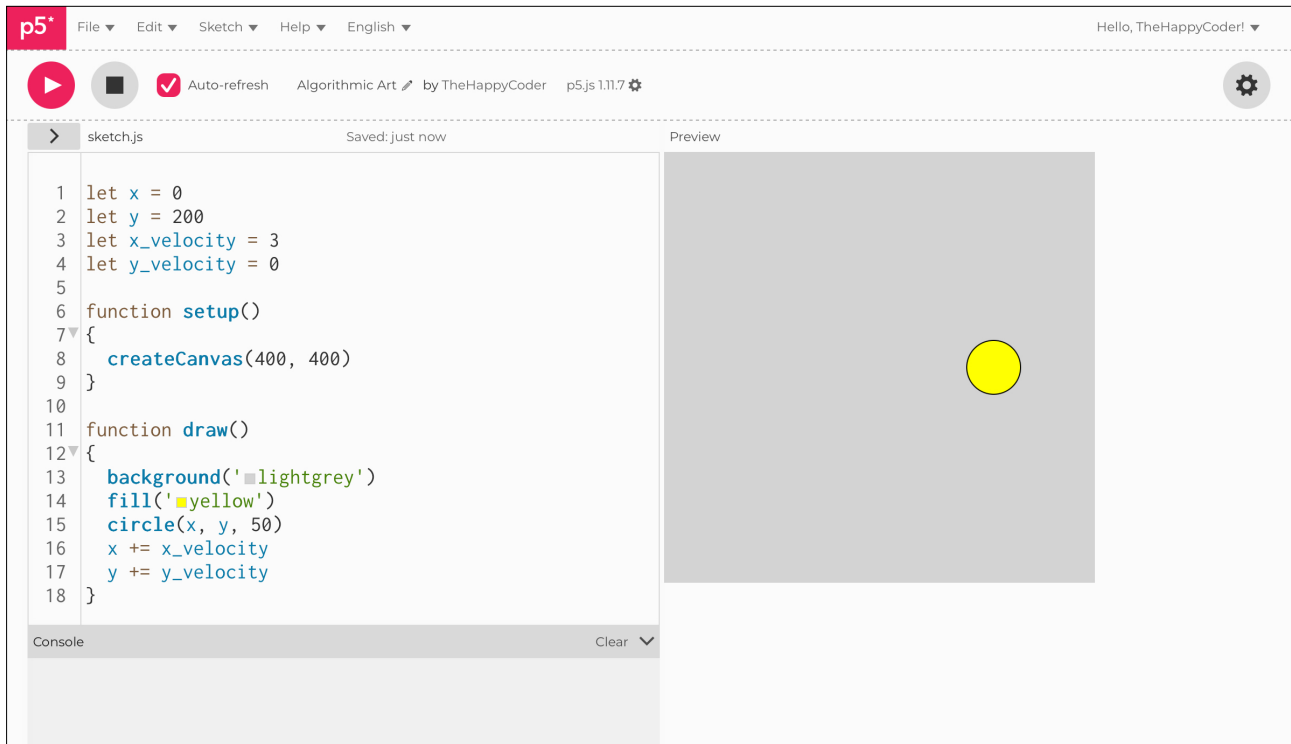
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background('lightgrey')
  fill('yellow')
  circle(x, y, 50)
  x += x_velocity
  y += y_velocity
}
```

Notes

Another naming convention is to use an underscore if splitting word and letters in a variable name.

Figure D2.2





Sketch D2.3 adding a bit of gravity

We give the **y_velocity** a bit of movement upwards (hence the negative value), but by adding a bit of **gravity**, it causes the ball to fall downwards to the ground after a while.

```
let x = 0
let y = 200
let x_velocity = 3
let y_velocity = -3
let gravity = 0.07

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

function draw()
{
  background('lightgrey')
  fill('yellow')
  circle(x, y, 50)
  x += x_velocity
  y += y_velocity
  y_velocity += gravity
}
```

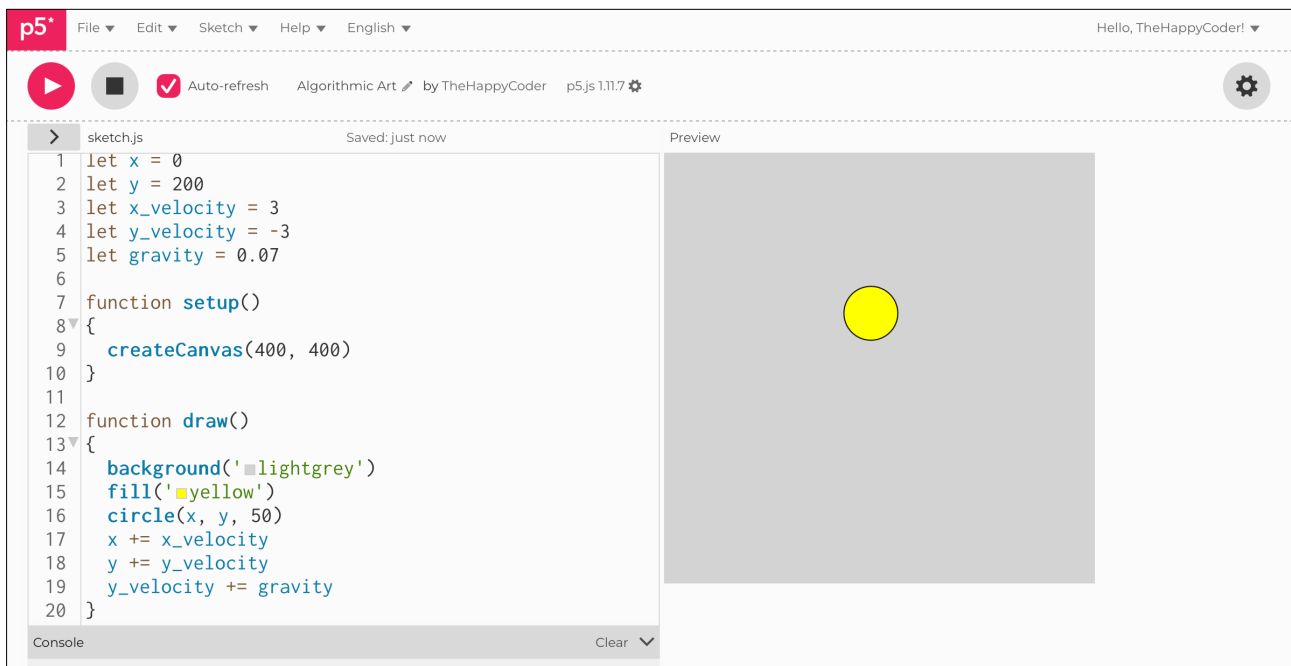
Notes

The ball goes up briefly and then starts to fall downwards once the velocity becomes positive.

Challenges

1. Try different **y** velocities and **gravity** values.
2. Wind could be a variable in the **x** direction.

Figure D2.3





Sketch D2.4 creating a ball class

We have just built the basic structure for the class. We have our `constructor()` function; we will also need a `show()` function and a `move()` function. I want to emphasise now that the names of these functions are made up; you can call each one of them whatever names you want. It is convention to have a constructor function, but the others can have other names and often do.

```
let x = 0
let y = 200
let x_velocity = 3
let y_velocity = -3
let gravity = 0.07

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

function draw()
{
  background('lightgrey')
  fill('yellow')
  circle(x, y, 50)
  x += x_velocity
  y += y_velocity
  y_velocity += gravity
}
```

```
class Ball
{
  constructor()
  {

  }

  show()
  {

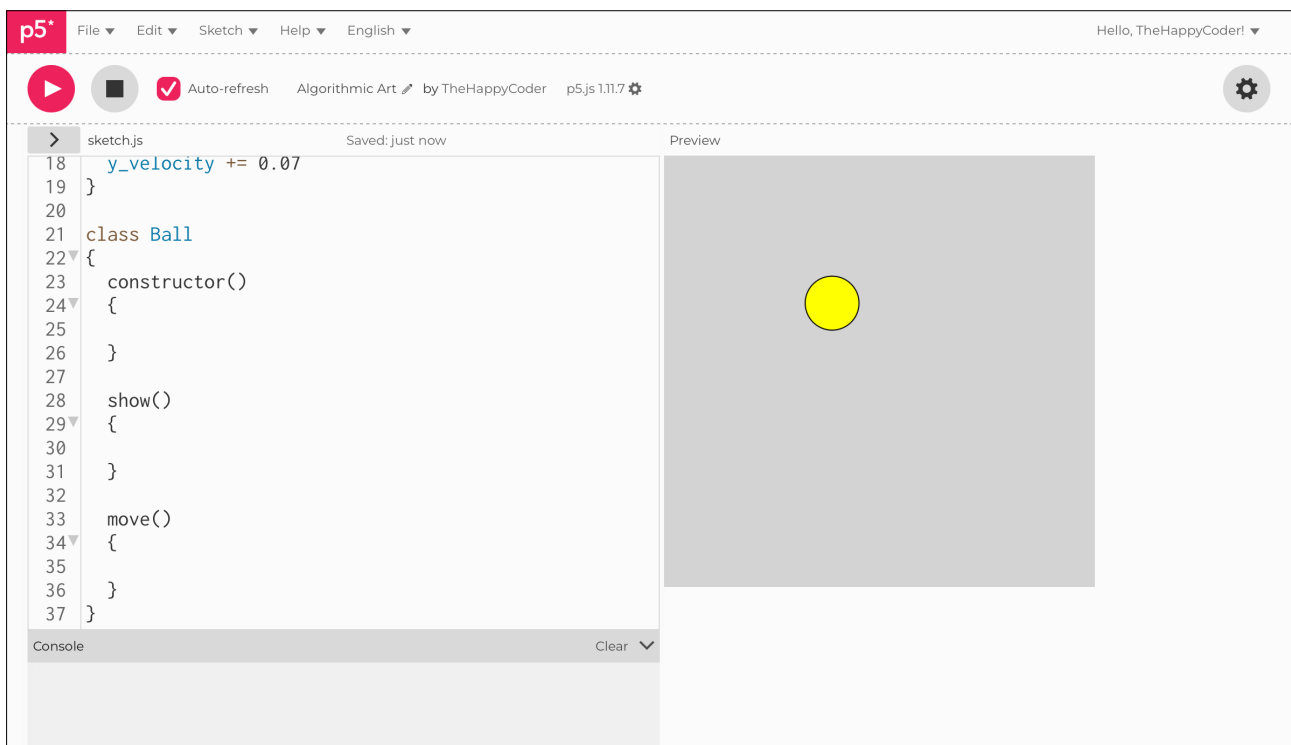
  }
}
```

```
move()  
{  
  
}  
}
```

Notes

Nothing new will happen.

Figure D2.4





Sketch D2.5 the constructor() function

! Remove the commented lines of code. You will get an error message!

First, we will fill in the `constructor()` function using the variables highlighted. At the moment, you will get an error message if you try to run it now.

```
// let x = 0
// let y = 200
// let x_velocity = 3
// let y_velocity = -3
// let gravity = 0.07

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

function draw()
{
  background('lightgrey')
  fill('yellow')
  circle(x, y, 50)
  x += x_velocity
  y += y_velocity
  y_velocity += gravity
}

class Ball
{
  constructor()
  {
    this.x = 0
    this.y = 200
    this.x_velocity = 3
    this.y_velocity = -3
    this.gravity = 0.07
  }
}
```

```
show()  
{  
  
}  
  
move()  
{  
  
}  
}
```

Notes

The **this._____** means it is referring to a specific ball, not necessarily to all of them. You should get an error message re: the **x** variable.



Sketch D2.6 the show() function

! Remove commented out lines of code. The error message should persist!

We can now fill in the `show()` function.

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

function draw()
{
  background('lightgrey')
  // fill('yellow')
  // circle(x, y, 50)
  x += x_velocity
  y += y_velocity
  y_velocity += gravity
}

class Ball
{
  constructor()
  {
    this.x = 0
    this.y = 200
    this.x_velocity = 3
    this.y_velocity = -3
    this.gravity = 0.07
  }

  show()
  {
    fill('yellow')
    circle(this.x, this.y, 50)
  }

  move()
```

```
{  
  
}  
}
```

Notes

The **x** and **y** coordinates become **this.x** and **this.y** for each ball we create. Still an error message, but don't worry.



Sketch D2.7 the move() function

! Remove the commented lines of code. No error message this time!

And finally, the `move()` function.

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

function draw()
{
  background('lightgrey')
  // x += x_velocity
  // y += y_velocity
  // y_velocity += gravity
}

class Ball
{
  constructor()
  {
    this.x = 0
    this.y = 200
    this.x_velocity = 3
    this.y_velocity = -3
    this.gravity = 0.07
  }

  show()
  {
    fill('yellow')
    circle(this.x, this.y, 50)
  }

  move()
  {
    this.x += this.x_velocity
```

```
    this.y += this.y_velocity  
    this.y_velocity += this.gravity  
}  
}
```

Notes

You shouldn't get an error message now, but neither will you see anything. We need to create the ball next and call those functions.



Sketch D2.8 can we have our ball back, please

We have finally got our ball back, phew!

```
let ball

function setup()
{
  createCanvas(400, 400)
  ball = new Ball()
}

function draw()
{
  background('lightgrey')
  ball.show()
  ball.move()
}

class Ball
{
  constructor()
  {
    this.x = 0
    this.y = 200
    this.x_velocity = 3
    this.y_velocity = -3
    this.gravity = 0.07
  }

  show()
  {
    fill('yellow')
    circle(this.x, this.y, 50)
  }

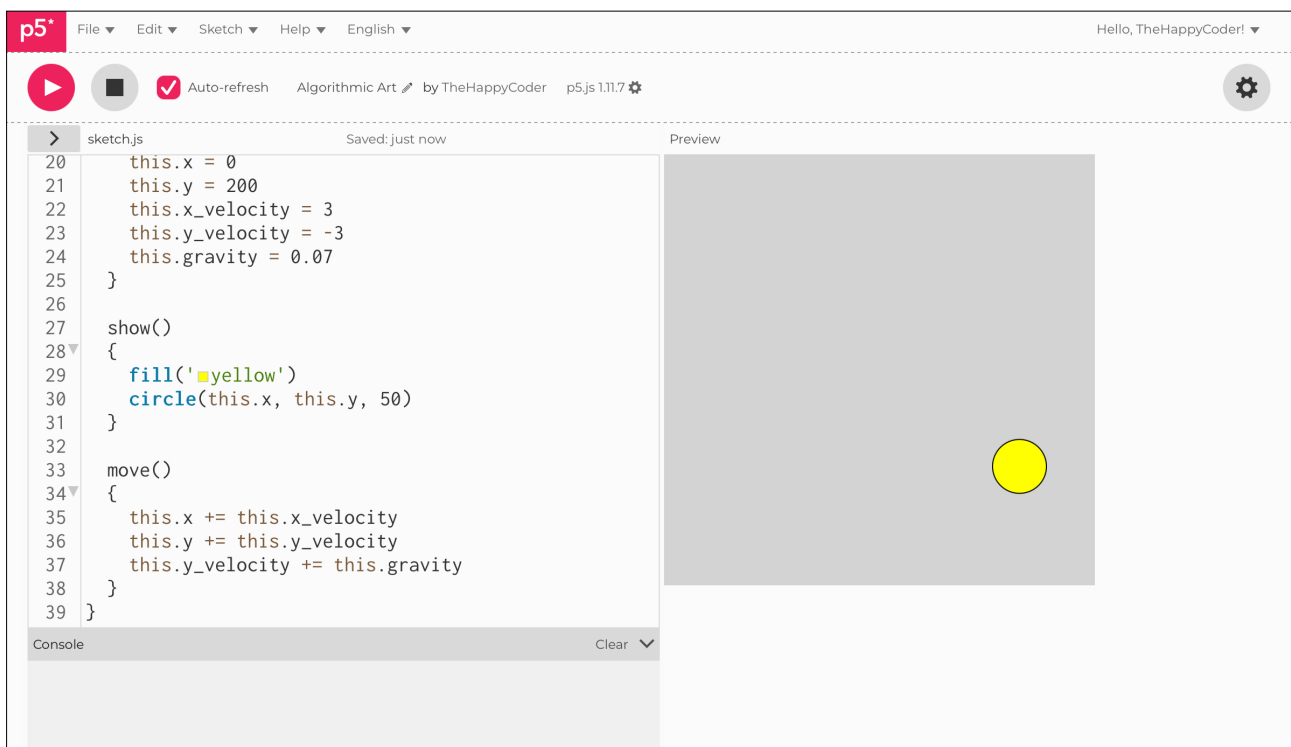
  move()
  {
    this.x += this.x_velocity
```

```
this.y += this.y_velocity  
this.y_velocity += this.gravity  
}  
}
```

Notes

Everything should now be as before if all is well. Back at square one.

Figure D2.8





Sketch D2.9 many balls

! Remove and replace `ball = new Ball()` with the `for()` loop. Expected error message!

We can have more than one ball, so let's have `20` and for that we create an array to store them by pushing each new ball into that array, warning you that you will get an error message.

```
let ball

let balls = []
let number = 20

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < number; i++)
  {
    balls.push(new Ball())
  }
}

function draw()
{
  background('lightgrey')
  ball.show()
  ball.move()
}

class Ball
{
  constructor()
  {
    this.x = 0
    this.y = 200
    this.x_velocity = 3
    this.y_velocity = -3
    this.gravity = 0.07
  }
}
```

```
show()  
{  
  fill('yellow')  
  circle(this.x, this.y, 50)  
}  
  
move()  
{  
  this.x += this.x_velocity  
  this.y += this.y_velocity  
  this.y_velocity += this.gravity  
}  
}
```

Notes

We have broken the sketch, but have no fear, we will fix that. Error message!



Sketch D2.10 pulling the balls out

We need to pull each of the balls from the array and `show()` and `move()` each one. We use the `for...of` loop.

```
let ball
let balls = []
let number = 20

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < number; i++)
  {
    balls.push(new Ball())
  }
}

function draw()
{
  background('lightgrey')
  for (ball of balls)
  {
    ball.show()
    ball.move()
  }
}

class Ball
{
  constructor()
  {
    this.x = 0
    this.y = 200
    this.x_velocity = 3
    this.y_velocity = -3
    this.gravity = 0.07
  }
}
```

```

show()
{
  fill('yellow')
  circle(this.x, this.y, 50)
}

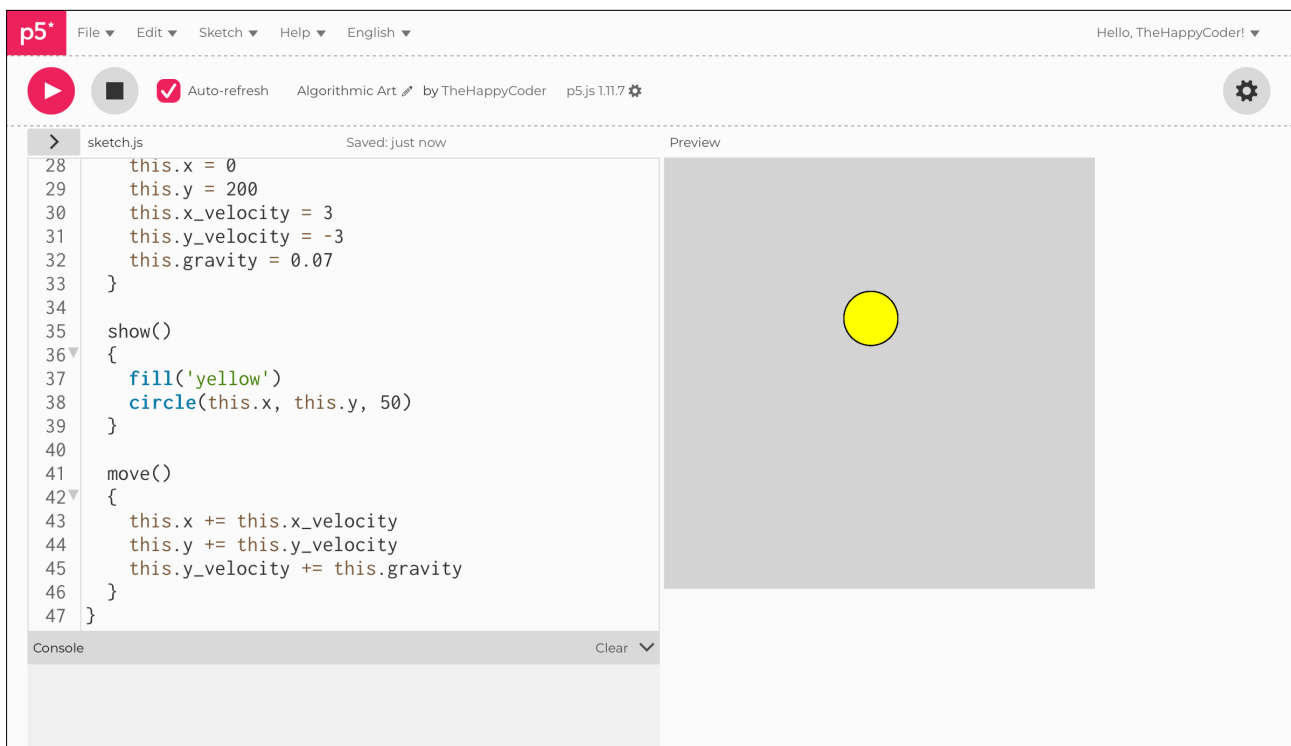
move()
{
  this.x += this.x_velocity
  this.y += this.y_velocity
  this.y_velocity += this.gravity
}
}

```

Notes

We have fixed the sketch, but there were supposed to be **20** balls, not just one! To see them, we need to mix them up a bit; they are occupying the same space.

Figure D2.10





Sketch D2.11 random velocity

To mix it up, we can introduce some randomness to their velocities for both **x** and **y**.

```
let ball
let balls = []
let number = 20

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < number; i++)
  {
    balls.push(new Ball())
  }
}

function draw()
{
  background('lightgrey')
  for (ball of balls)
  {
    ball.show()
    ball.move()
  }
}

class Ball
{
  constructor()
  {
    this.x = 0
    this.y = 200
    this.x_velocity = 3 * random(1, 2)
    this.y_velocity = -3 * random(-1, 1)
    this.gravity = 0.07
  }

  show()
```

```

{
  fill('yellow')
  circle(this.x, this.y, 50)
}

move()
{
  this.x += this.x_velocity
  this.y += this.y_velocity
  this.y_velocity += this.gravity
}
}

```

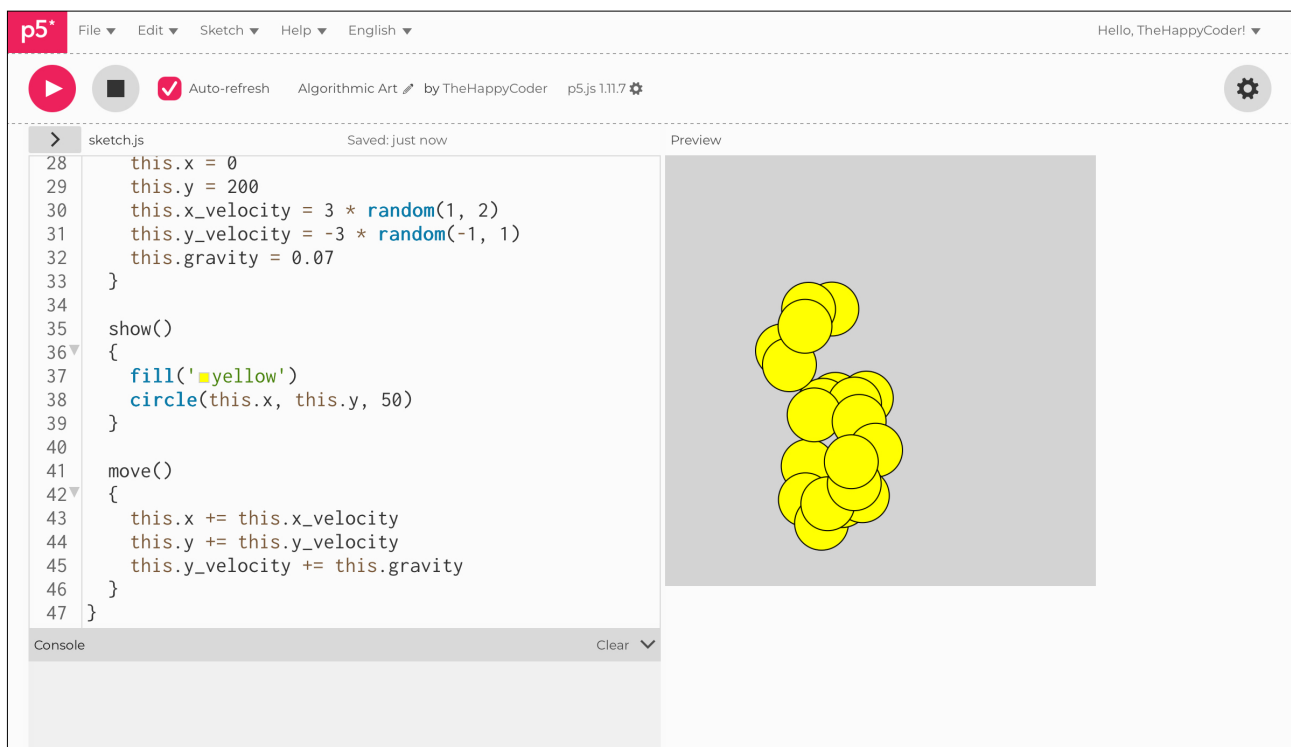
Notes

They can now wander off in different directions.

Challenge

Try other random values.

Figure D2.11





Sketch D2.12 refactoring

A bit of refactoring for a slightly different effect.

```
let ball
let balls = []
let number = 200

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < number; i++)
  {
    balls.push(new Ball())
  }
}

function draw()
{
  background('lightgrey')
  for (ball of balls)
  {
    ball.show()
    ball.move()
  }
}

class Ball
{
  constructor()
  {
    this.x = 200
    this.y = 200
    this.x_velocity = 3 * random(-1, 1)
    this.y_velocity = -3 * random(-1, 2)
    this.gravity = 0.07
  }

  show()
}
```

```

{
  fill('yellow')
  circle(this.x, this.y, 20)
}

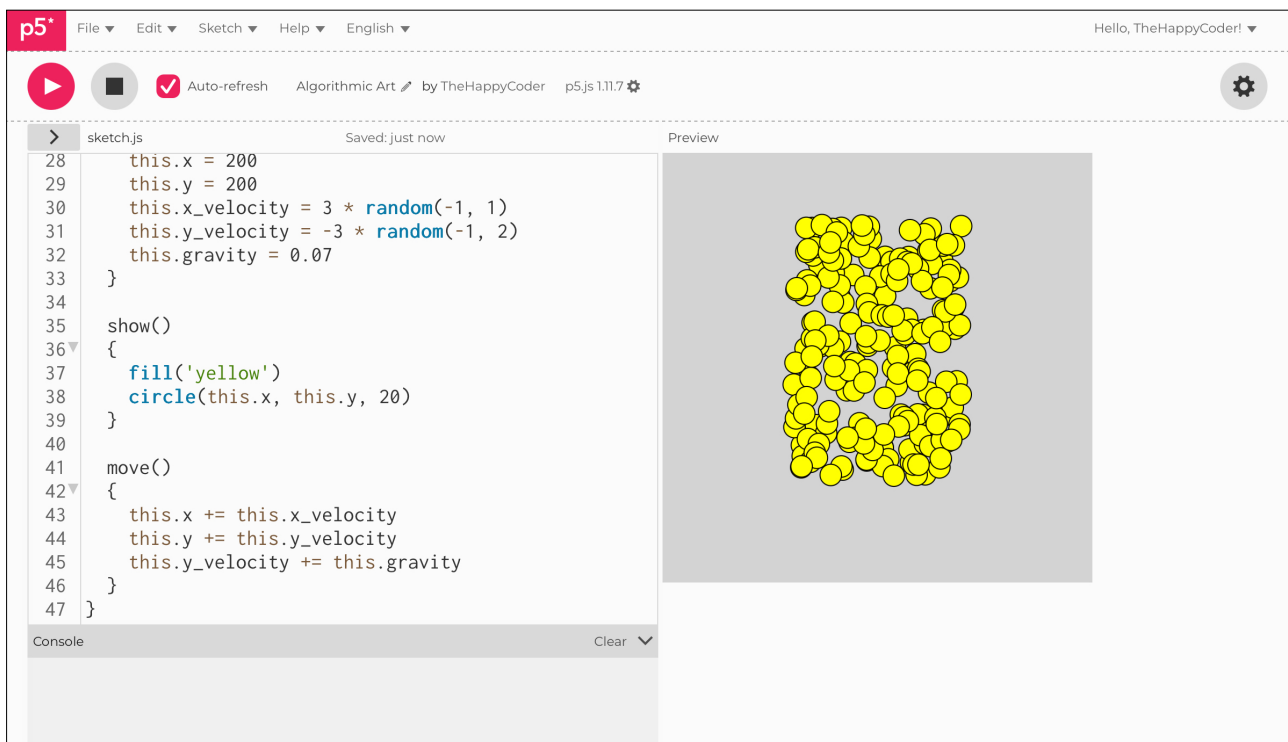
move()
{
  this.x += this.x_velocity
  this.y += this.y_velocity
  this.y_velocity += this.gravity
}
}

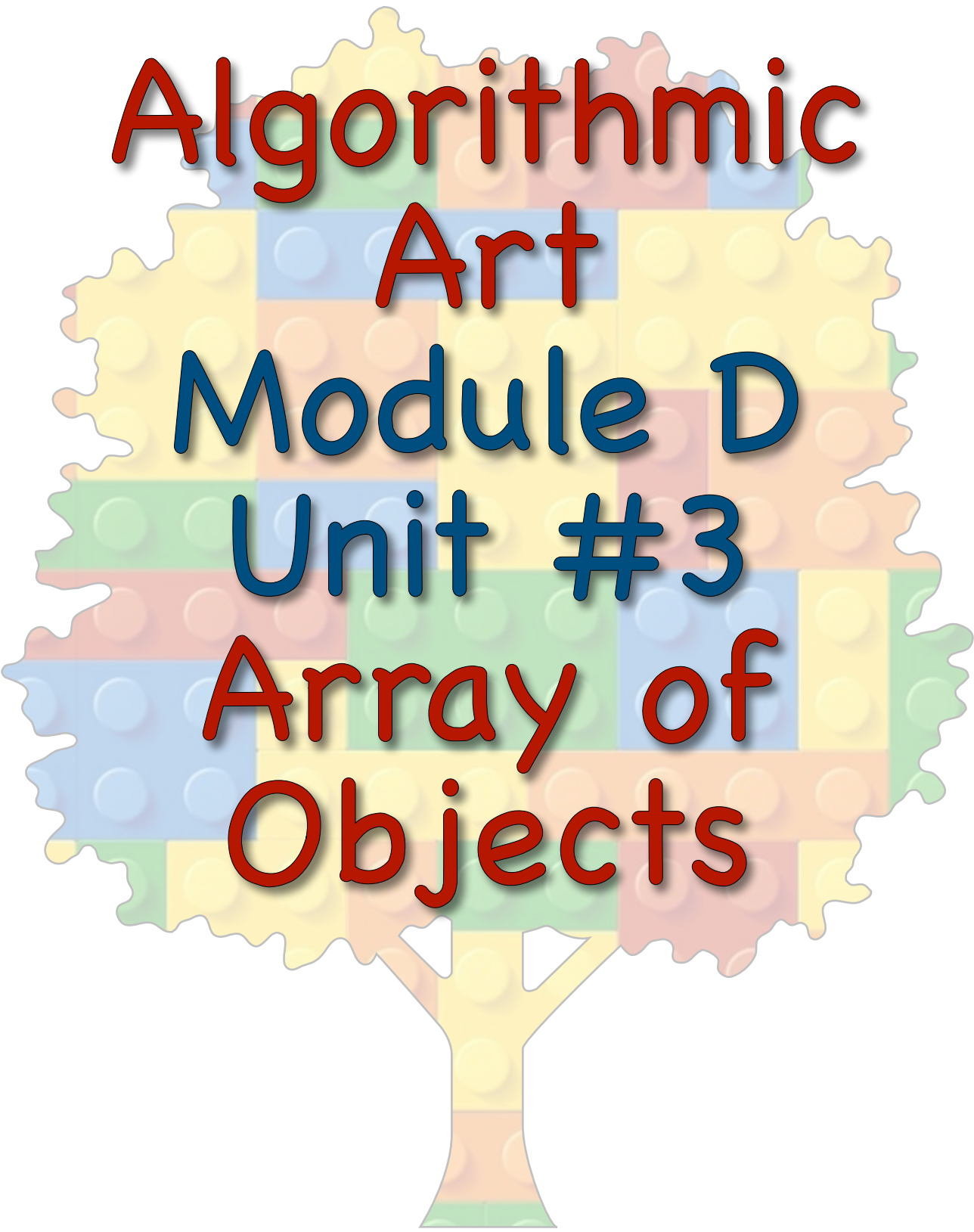
```

Notes

We will have **200** smaller balls emitting from the centre; you could have something like a firework-type display.

Figure D2.12





Algorithmic Art

Module D

Unit #3

Array of Objects



Module D Unit #3: array of objects

In this unit, we will mash together arrays and classes with a dash of mouse functions.

Recap: Arrays are extremely useful and powerful in so many ways. An array is a list of elements inside square brackets separated by a comma. The numbering of each element starts at zero; they are called the index number. Let's create an array called num and put some numbers in it.

```
let num = [42, 56, 63, 79, 88]
```

There are five numbers (elements) in the array. They are, in order: the first one is 42, the second one is 56, the third one is 63, the fourth one is 79, and the fifth one is 88.

However, to reference the first element (42), we say it is at index 0. The second element (56) is at index 1, the third (63) is index 2, the fourth (79) is index 3, and the fifth (88) is at index 4. This may seem a bit silly, but it is the way it is, so you will need to get used to it.



Sketch D3.1 bubbles

! Create a new sketch.

We are going to make an array of objects. We start with an empty array called **bubbles**.

```
let bubbles = []

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

function draw()
{
  background(220)
}
```

Notes

Nothing to see



Sketch D3.2 classy bubbles

We are going to put **bubble** objects in that array, but first we need to create a **Bubble** class to create some **bubbles**.

```
let bubbles = []

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

function draw()
{
  background(220)
}
```

```
class Bubble
{
  constructor()
  {

  }

  move()
  {

  }

  show()
  {

  }
}
```

Notes

Nothing to see here.



Sketch D3.3 time for an argument

We need three arguments for the bubble: the **x**, **y** (coordinates), and **r** (radius). The radius will be randomly generated. The **x** and **y** positions will be created with a click or drag of the mouse.

```
let bubbles = []

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

function draw()
{
  background(220)
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
  {

  }

  show()
  {
    circle(this.x, this.y, this.r)
  }
}
```

Notes

We are not moving anything because we haven't drawn a bubble. So there is nothing to see just yet; all will be revealed soon.



Sketch D3.4 drawing the bubbles

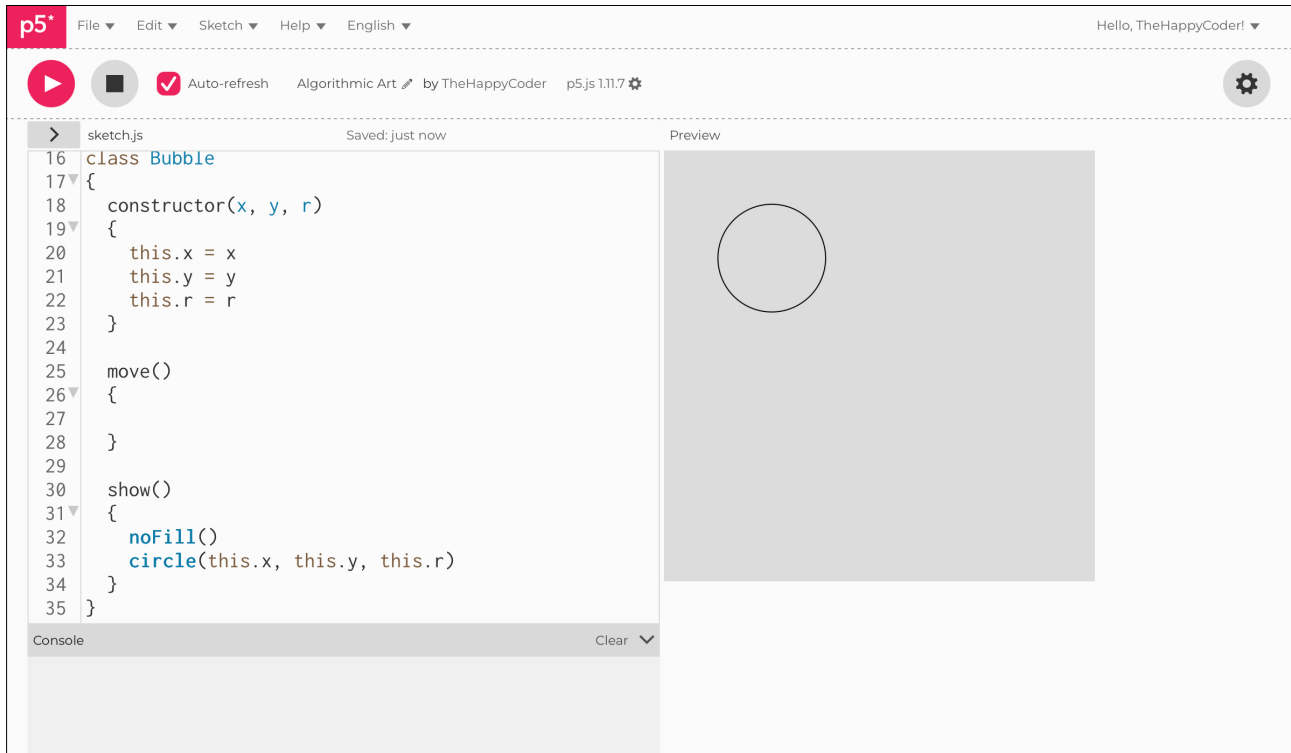
Let us draw one bubble to start with.

```
let bubbles = []  
let bubble  
  
function setup()  
{  
  createCanvas(400, 400)  
  bubble = new Bubble(100, 100, 100)  
}  
  
function draw()  
{  
  background(220)  
  bubble.show()  
}  
  
class Bubble  
{  
  constructor(x, y, r)  
  {  
    this.x = x  
    this.y = y  
    this.r = r  
  }  
  
  move()  
  {  
  
  }  
  
  show()  
  {  
    noFill()  
    circle(this.x, this.y, this.r)  
  }  
}
```

Notes

We finally have something to see, our first bubble.

Figure D3.4





Sketch D3.5 random motion

Let's move it around in a random manner.

```
let bubbles = []
let bubble

function setup()
{
  createCanvas(400, 400)
  bubble = new Bubble(100, 100, 100)
}

function draw()
{
  background(220)
  bubble.show()
  bubble.move()
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
  {
    this.x = this.x + random(-5, 5)
    this.y = this.y + random(-5, 5)
  }

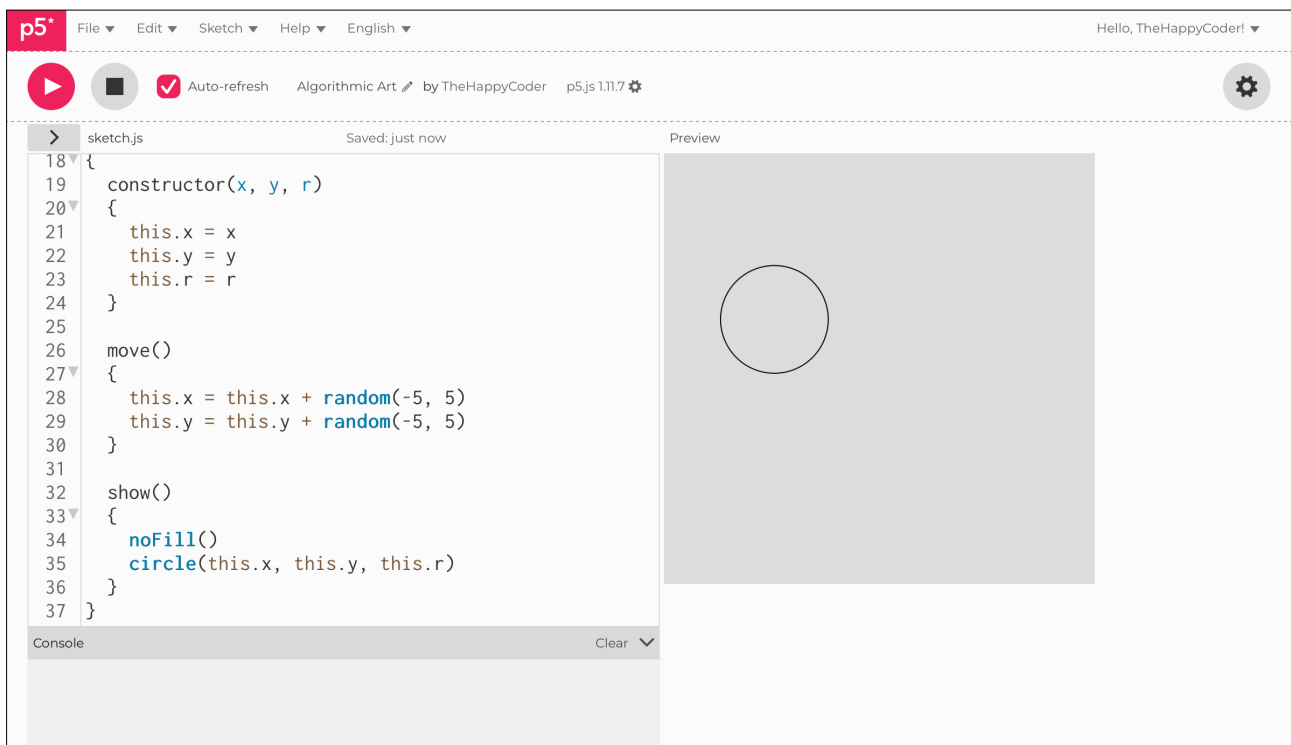
  show()
  {
    noFill()
    circle(this.x, this.y, this.r)
  }
}
```

```
}  
}
```

Notes

What you should see is a circle wobbling around the canvas (or wandering off it eventually).

Figure D3.5





Sketch D3.6 many bubbles

We can create many of them with a nice `for()` loop and put them into the `bubbles` array. Then we have another `for()` loop to pull them out. Here we are just creating and drawing one bubble for the moment.

```
let bubbles = []
let bubble

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < 1; i++)
  {
    bubbles[i] = new Bubble(100, 100, 100)
  }
}

function draw()
{
  background(220)
  for (let i = 0; i < 1; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
  {

```

```

    this.x = this.x + random(-5, 5)
    this.y = this.y + random(-5, 5)
  }

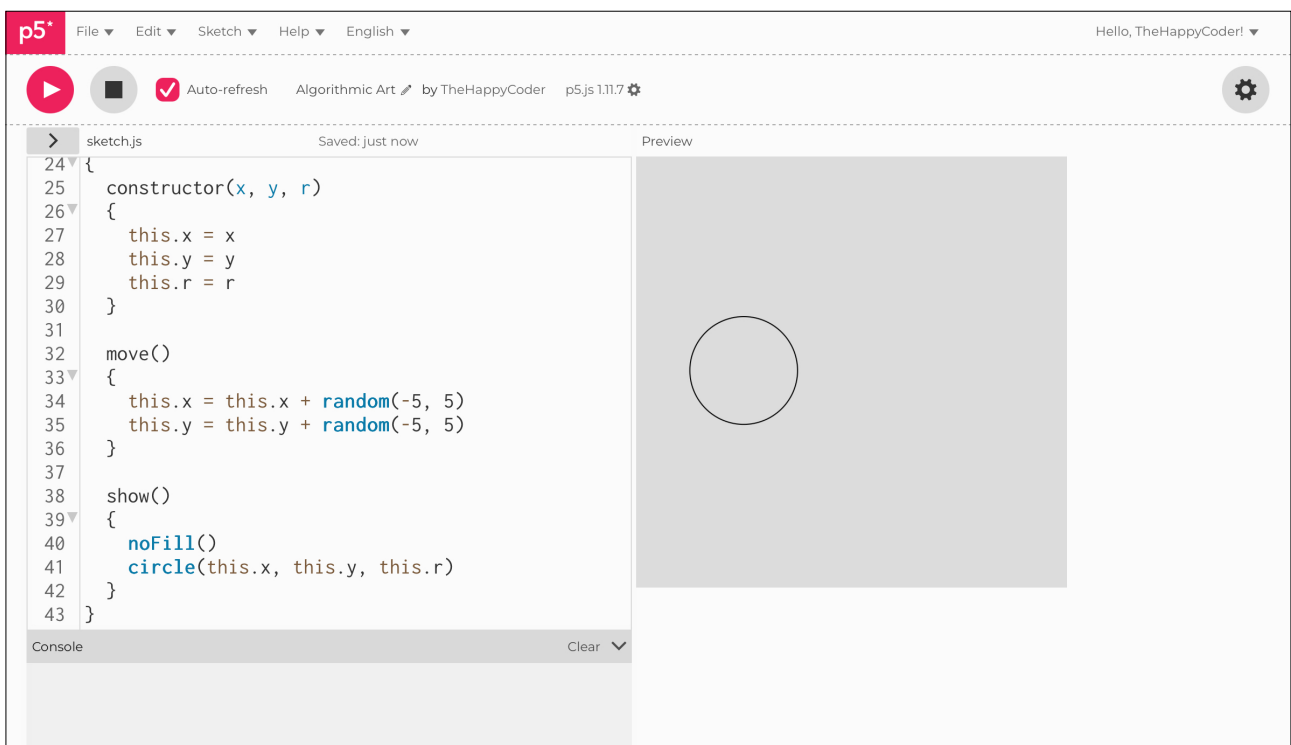
  show()
  {
    noFill()
    circle(this.x, this.y, this.r)
  }
}

```

Notes

We have not really changed anything; you should have the same effect.

Figure D3.6





Sketch D3.7 more bubbles

We only drew one **bubble** above. There is only one **bubble** in the array of **bubbles**. We can change that by increasing it from **1** to **10**. In the second **for()** loop, we can call the length of the array rather than hardcode it just in case we change the length of the array.

```
let bubbles = []
let bubble

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < 10; i++)
  {
    bubbles[i] = new Bubble(100, 100, 100)
  }
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
```

```

{
  this.x = this.x + random(-5, 5)
  this.y = this.y + random(-5, 5)
}

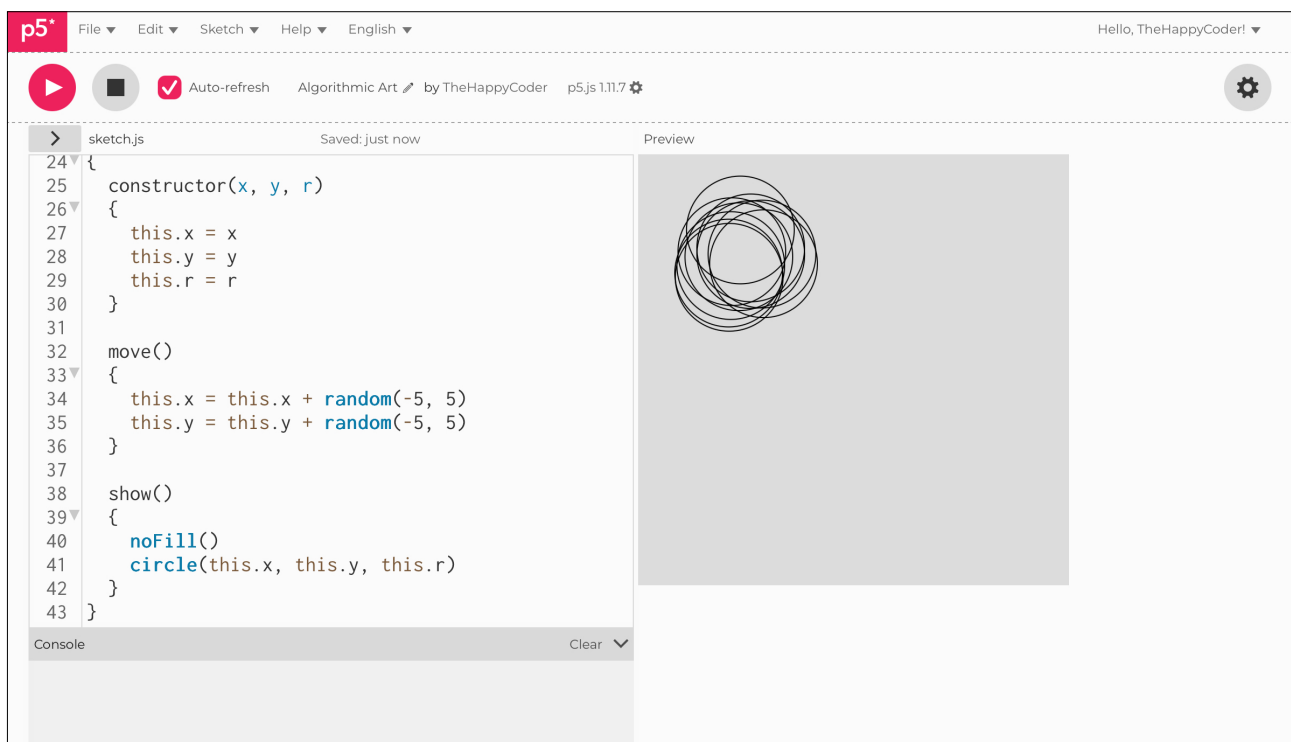
show()
{
  noFill()
  circle(this.x, this.y, this.r)
}
}

```

Notes

A series of wobbling circles

Figure D3.7





Sketch D3.8 spacing them out

We started each one in the same spot. Let's space them along a line using the loop. We start at **10** and multiply the index **i** by **40**.

```
let bubbles = []
let bubble

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < 10; i++)
  {
    let x = 10 + 40 * i
    bubbles[i] = new Bubble(x, 100, 100)
  }
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
  {

```

```

    this.x = this.x + random(-5, 5)
    this.y = this.y + random(-5, 5)
  }

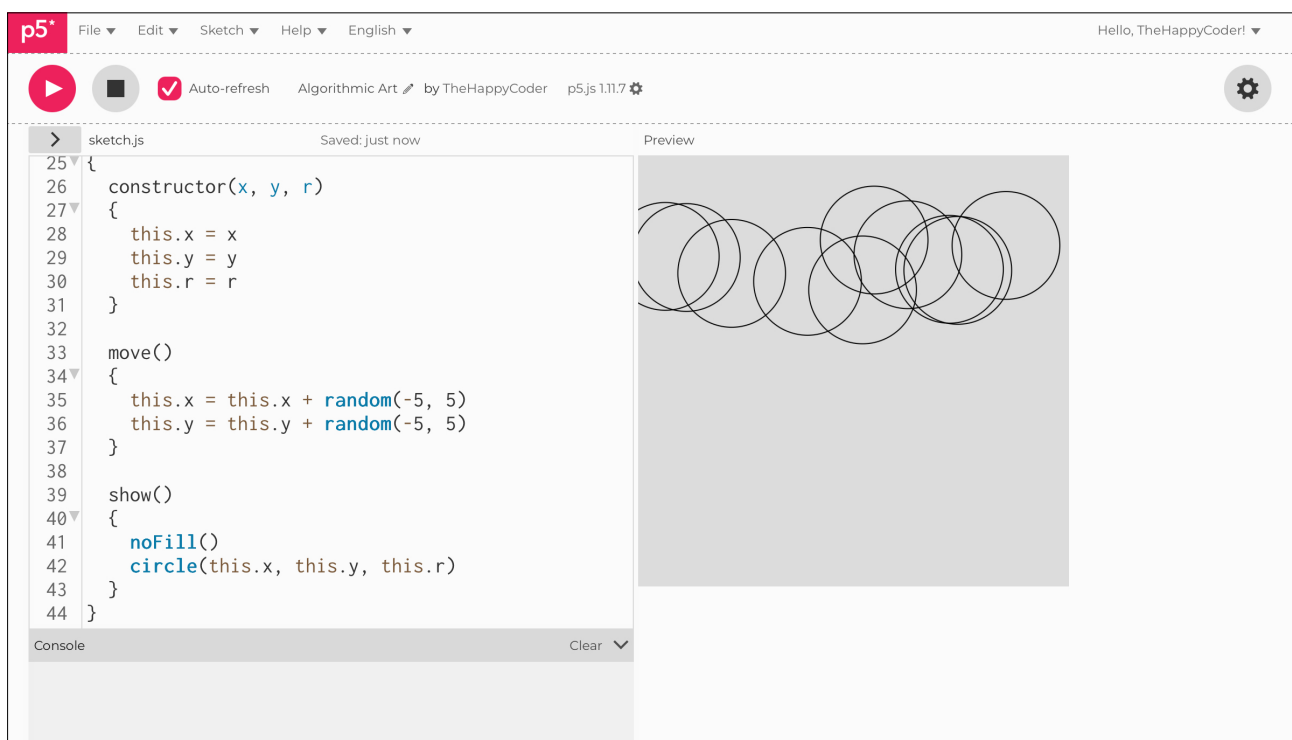
  show()
  {
    noFill()
    circle(this.x, this.y, this.r)
  }
}

```

Notes

Notice we haven't touched anything in the class itself.

Figure D3.8





Sketch D3.9 random

Let us create a set of bubbles of random positions and sizes.

```
let bubbles = []
let bubble

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < 10; i++)
  {
    let x = random(width)
    let y = random(height)
    let r = random(20, 100)
    bubbles[i] = new Bubble(x, y, r)
  }
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
```

```

{
  this.x = this.x + random(-5, 5)
  this.y = this.y + random(-5, 5)
}

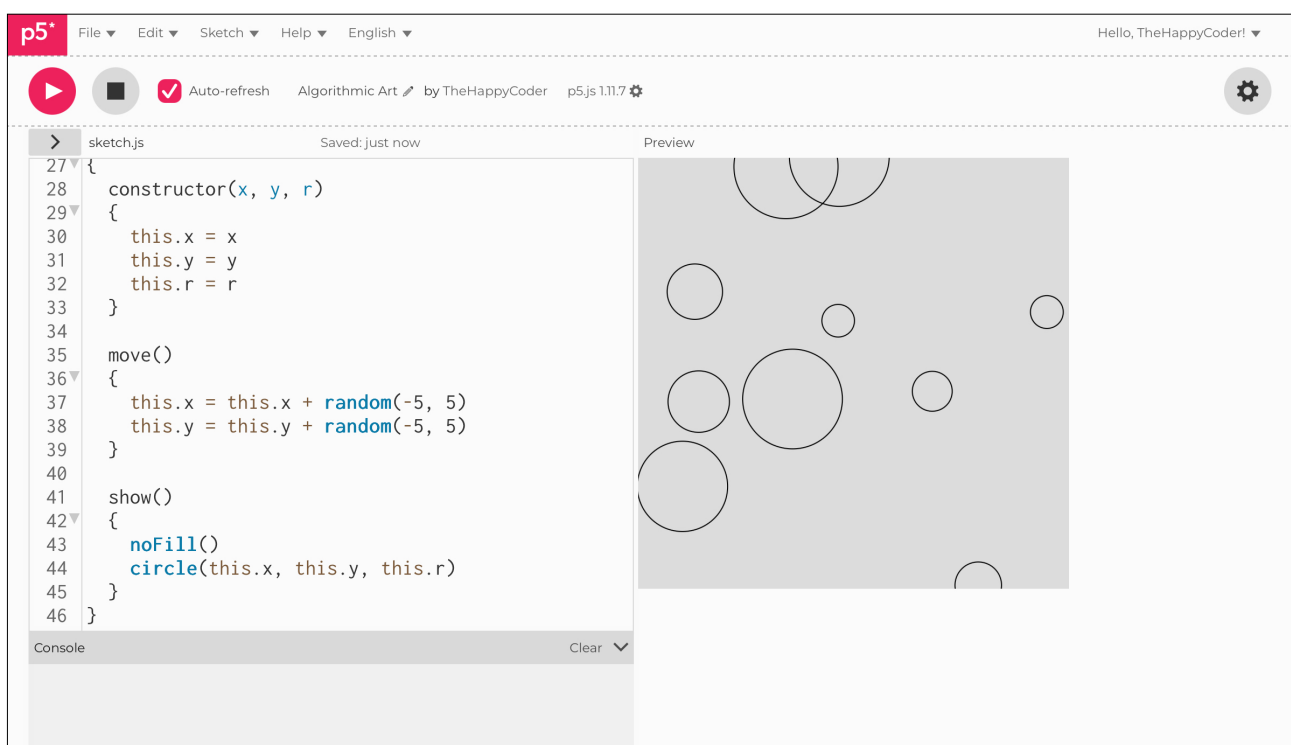
show()
{
  noFill()
  circle(this.x, this.y, this.r)
}
}

```

Notes

You get ten circles of various sizes and positions.

Figure D3.9





Sketch D3.10 the shimmering fog

That is all well and good, but what can we do to make it more interesting? With a few tweaks, we can do something quite slightly unexpected.

```
let bubbles = []
let bubble

function setup()
{
  createCanvas(400, 400)
  for (let i = 0; i < 1000; i++)
  {
    let x = random(width)
    let y = random(height)
    let r = random(20, 100)
    bubbles[i] = new Bubble(x, y, r)
  }
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }
}
```

```

move()
{
  this.x = this.x + random(-3, 3)
  this.y = this.y + random(-3, 3)
}

show()
{
  noStroke()
  fill(51, 10)
  circle(this.x, this.y, this.r)
}
}

```

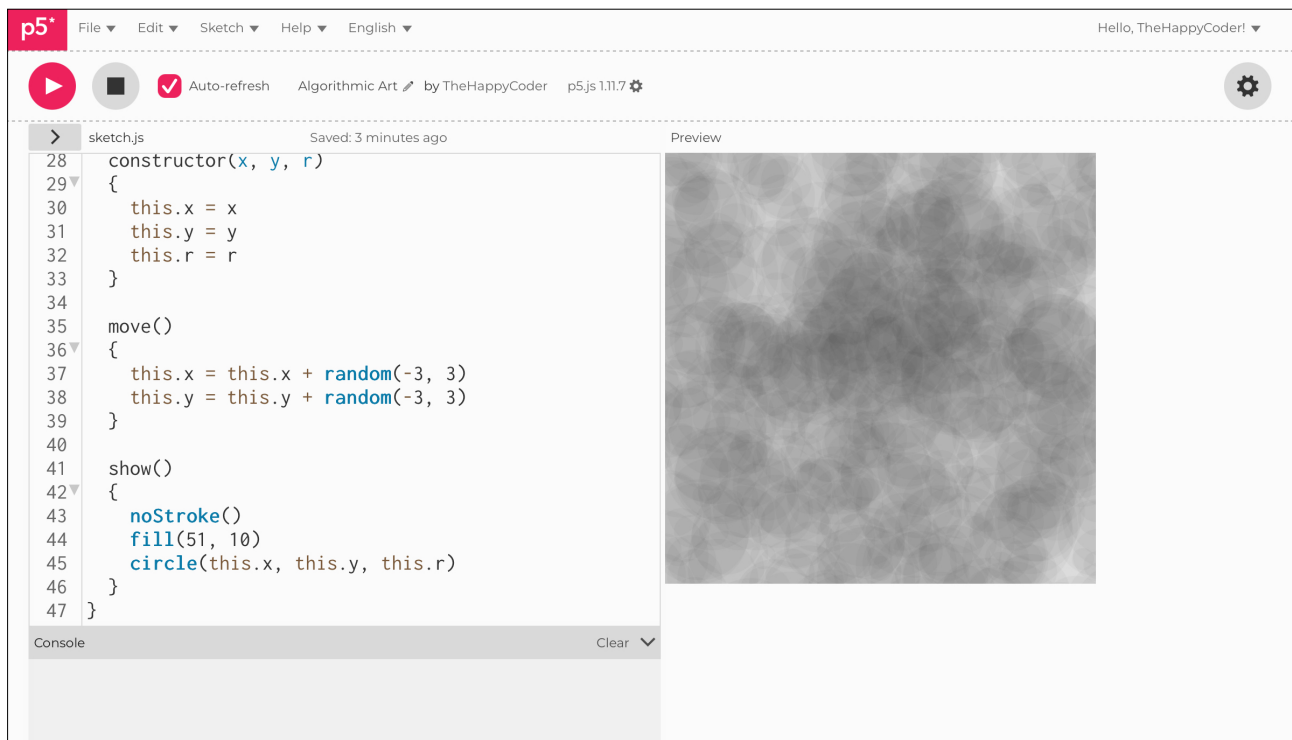
Notes

Sometimes it doesn't take much.

Challenge

Play with it.

Figure D3.10





Sketch D3.11 simpler times

Let's go back to a simpler sketch. A bit of deleting and a couple of lines of code will bring you back to a familiar starting point.

! Remove the lines of code highlighted in blue and commented out (`//`), and add the `noFill()` in the `show()` function.

```
let bubbles = []
let bubble

function setup()
{
  createCanvas(400, 400)
  // for (let i = 0; i < 1000; i++)
  // {
  //   let x = random(width)
  //   let y = random(height)
  //   let r = random(20, 100)
  //   bubbles = new Bubble(x, y, r)
  // }
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }
}
```

```
}

move()
{
  this.x = this.x + random(-3, 3)
  this.y = this.y + random(-3, 3)
}

show()
{
  // noStroke()
  // fill(51, 10)
  noFill()
  circle(this.x, this.y, this.r)
}
}
```

Notes

A blank grey canvas.



Sketch D3.12 mouse click

Creating a new bubble at the click of a mouse, we create a function called `mousePressed()` to do this.

```
let bubbles = []
let bubble

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

function mousePressed()
{
  bubble = new Bubble(mouseX, mouseY, 20)
  bubbles[0] = bubble
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
```

```

{
  this.x = this.x + random(-3, 3)
  this.y = this.y + random(-3, 3)
}

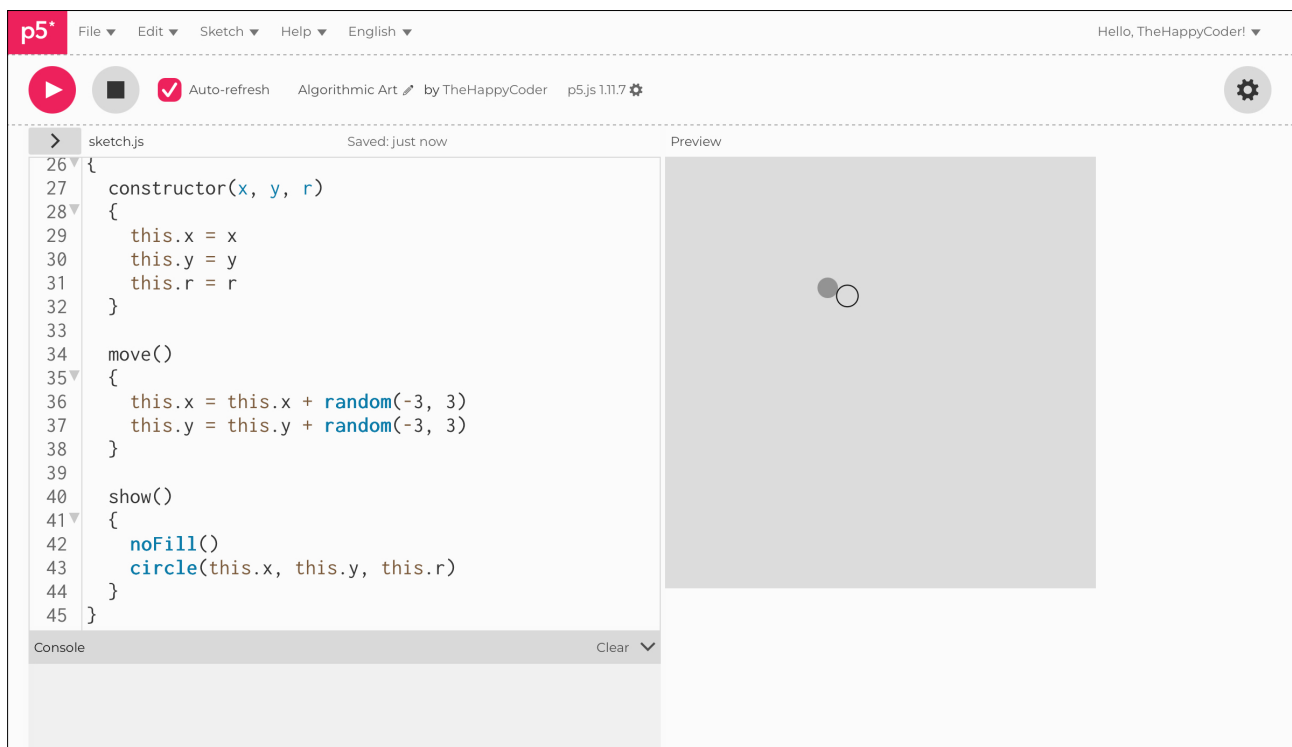
show()
{
  noFill()
  circle(this.x, this.y, this.r)
}
}

```

Notes

We create a new **bubble** every time we click on the canvas, but only one at a time. We want to keep each one, not discard the last one.

Figure D3.12





Sketch D3.13 when push comes to shove

We use a `push()` function to push each `bubble` into the array. Now we will keep all the `bubbles` we create with the mouse.

```
let bubbles = []
let bubble

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

function mousePressed()
{
  bubble = new Bubble(mouseX, mouseY, 20)
  bubbles.push(bubble)
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
```

```

{
  this.x = this.x + random(-3, 3)
  this.y = this.y + random(-3, 3)
}

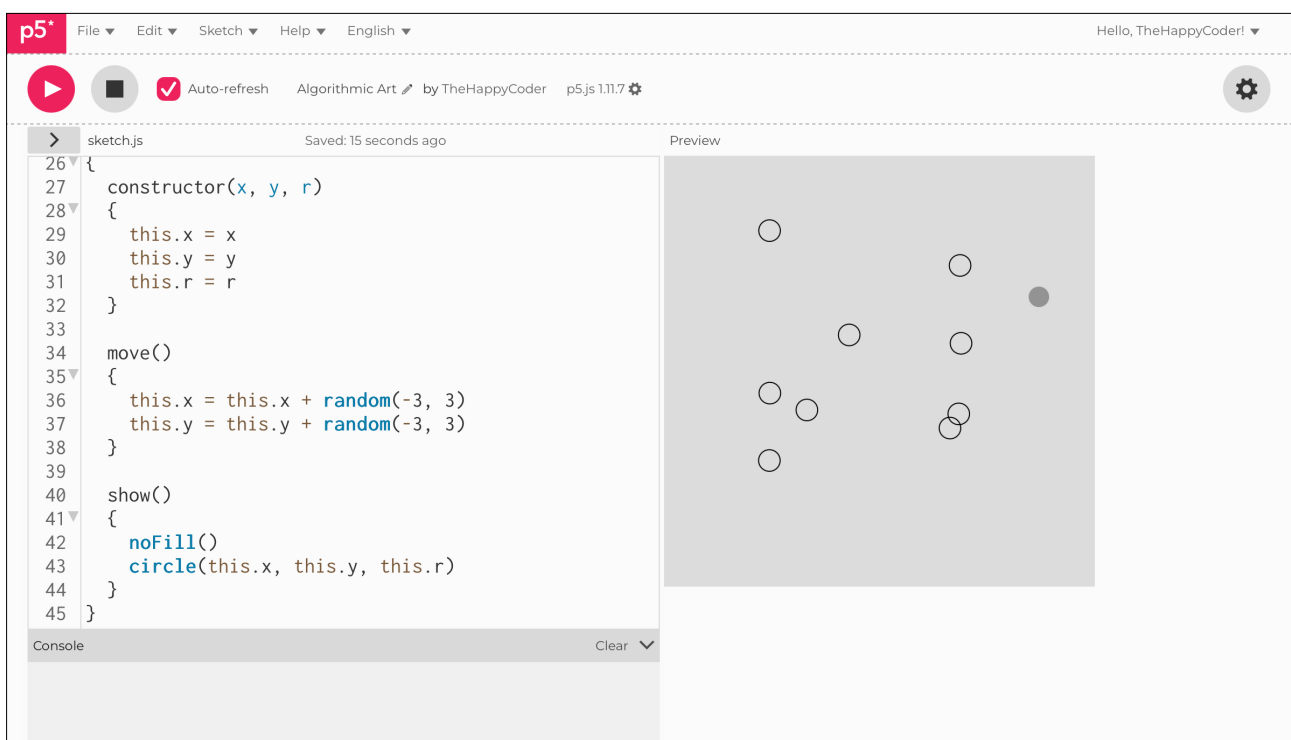
show()
{
  noFill()
  circle(this.x, this.y, this.r)
}
}

```

Notes

Now when you click on the canvas, each bubble remains on the canvas because they have been added to the array of bubbles.

Figure D3.13





Sketch D3.14 a bit of a drag

Finally, using `mouseDragged()` rather than just `mousePressed()`, we can make lots of **bubbles** as we click and drag the mouse.

```
let bubbles = []
let bubble

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

function mouseDragged()
{
  bubble = new Bubble(mouseX, mouseY, 20)
  bubbles.push(bubble)
}

function draw()
{
  background(220)
  for (let i = 0; i < bubbles.length; i++)
  {
    bubbles[i].show()
    bubbles[i].move()
  }
}

class Bubble
{
  constructor(x, y, r)
  {
    this.x = x
    this.y = y
    this.r = r
  }

  move()
```

```

{
  this.x = this.x + random(-3, 3)
  this.y = this.y + random(-3, 3)
}

show()
{
  noFill()
  circle(this.x, this.y, this.r)
}
}

```

Challenges

1. What could you create with a few tweaks?
2. How would you create random shapes?

Figure D3.14

