

Algorithmic Intelligence The Joy of Coding Workbook #4 Neural Evolution





Table of Contents

Module D Unit #1: coding snippets #4	6
Introduction to perlin noise	7
Sketch D1.1 starting with a standard sketch	8
Sketch D1.2 randomly moving circle	9
Sketch D1.3 smooth random movement	11
It is all about functions, objects and classes	13
Sketch D1.4 starting sketch	14
Sketch D1.5 a single car	15
Sketch D1.6 moving the car	16
Sketch D1.7 a single car	18
The car as a function	20
Sketch D1.8 function single car	21
Sketch D1.9 function single car	23
The car as an object using functions	25
Sketch D1.10 car as an object	26
Sketch D1.11 alternative car object	28
Introduction to classes	30
Sketch D1.12 the constructor function	32
Sketch D1.13 the show function	34
Sketch D1.14 the move function	35
Sketch D1.15 creating a car	37
Sketch D1.16 to see it and move it	39
The power of classes	41
Sketch D1.17 car attributes	42
Sketch D1.18 a second car	45
Sketch D1.19 lots and lots of cars	48
Seeking a target using classes	50
Adding files	51
Sketch D1.20 index.html	54
Sketch D1.21 new sketch with a target	55
Sketch D1.22 the Vehicle class	57
Sketch D1.23 a vehicle design	59
Sketch D1.24 a starting position	61
Movement with vectors	63
Sketch D1.25 velocity and acceleration	64
Sketch D1.26 adding acc, vel and pos	65
Sketch D1.27 applying a force	66
Sketch D1.28 seeking the target	68
Sketch D1.29 seek, move and show	70

Sketch D1.30 reset acceleration	71
Sketch D1.31 steering	73
Sketch D1.32 rotate to the heading	75
Sketch D1.33 getting it working	77
Sketch D1.34 the maximum force	79
Sketch D1.35 the force is now steering	82
Module D Unit #2: neural evolution?	85
Smart cars and flappy birds	86
What is a genetic algorithm?	87
Example 1: Smart cars	89
Example 2: Flappy Bird	90
Module D Unit # smart cars neural evolution	93
The index.html file	94
Adding target.js and vehicle.js files	95
Adding them to the index.html file	96
Sketch D3.1 starting sketch	97
Sketch D3.2 target circle	98
Sketch D3.3 target class	100
Sketch D3.4 perlin noise	102
Sketch D3.5 moving target	104
Sketch D3.6 a vehicle	106
Sketch D3.7 random position	107
Sketch D3.8 moving towards the target	109
Sketch D3.9 the seek function	111
Sketch D3.10 seek and ye shall find	113
Sketch D3.11 more vehicles	115
A neural evolution brain	117
Sketch D3.12 giving the vehicle a brain	118
Sketch D3.13 steering the vehicles	120
Sketch D3.14 the outputs	123
Sketch D3.15 they move	125
Sketch D3.16 fitness scores	128
Sketch D3.17 we need a radius	131
Sketch D3.18 overlap	132
Weighted selection	134
Sketch D3.19 the relay race	135
Reproduction	138
Sketch D3.20 the next generation	139
Sketch D3.21 birth of a child	142
Sketch D3.22 lifeCounter	145
What next?	148

Sketch D3.23 adding a slider	149
Module D Unit #4: flappy bird neural evolution	154
The basic game	155
The index.html file (don't forget)	156
Sketch D4.1 starting sketch	157
Sketch D4.2 Bird class	158
Sketch D4.3 gravity	159
Sketch D4.4 pipes	161
Sketch D4.5 drawing the pipes	162
Sketch D4.6 collision	164
Sketch D4.7 drawing the bird!	166
Sketch D4.8 mouse pressed	167
Sketch D4.9 oops	168
Sketch D4.10 culling the array	170
Sketch D4.11 splice the pipe	172
Adding the brain	174
Sketch D4.12 simple bird brain	175
Sketch D4.13 next pipe please	177
Sketch D4.14 input data	179
Sketch D4.15 normalise	182
Sketch D4.16 synchronicity	184
Sketch D4.17 population of birds	187
Sketch D4.18 to flap or not to flap	189
Sketch D4.19 GPU v CPU	191
Sketch D4.20 bird fitness	193
Sketch D4.21 the bird is dead	196
Sketch D4.22 collision	198
Sketch D4.23 is anyone there	201
Sketch D4.24 mating the best ones	204
Sketch D4.25 normalising fitness	207
Sketch D4.26 crossover	210
Sketch D4.27 mutation	213
Sketch D4.28 next generation	216
Sketch D4.29 a new brain	219
Sketch D4.30 reset and off we go again	222
Sketch D4.31 eliminating silly birds	226

Algorithmic Intelligence Module D Unit #1 p5.js code snippets 4



Module D Unit #1: coding snippets #4

This is a more challenging coding snippet than the first three versions. We are going to cover some concepts which aren't neat little snippets but, rather, large chunks of code. The first section is about Perlin noise, which is a form of random number generation. It is less erratic; it is still random, but it is random in relation to the previous value, whereas using the `random()` function just generates a number with no regard to the previous number generated.

The second section looks at a number of things in the context of a moving vehicle. The main concept here is the introduction to classes. Using classes is a way of creating a template that we can use over and over again with variations. I will introduce an example where we will add files and code how a vehicle would seek a moving target.



Introduction to perlin noise

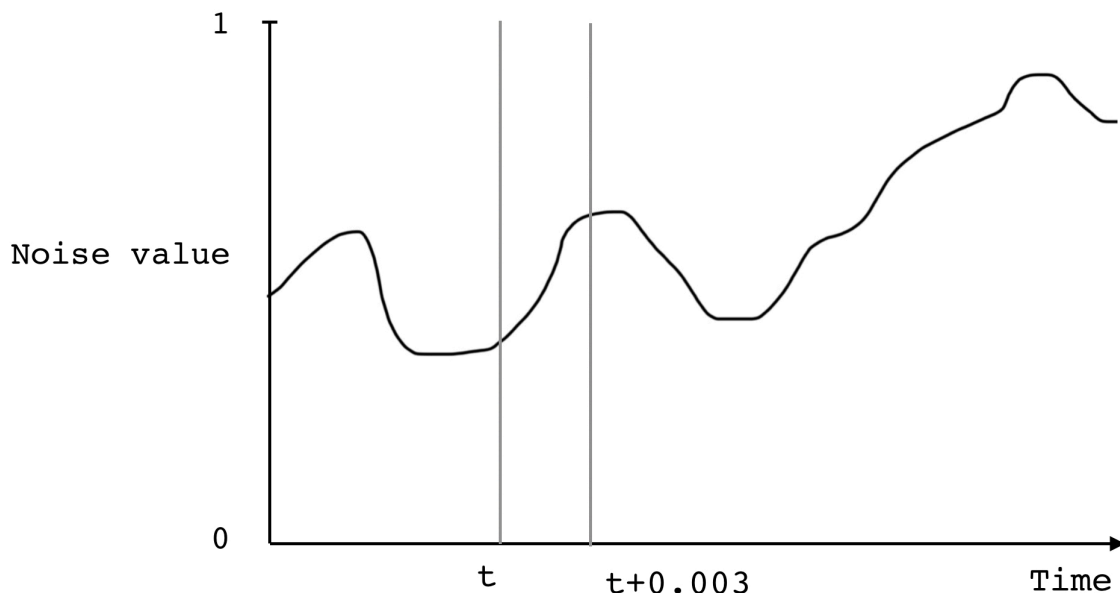
Perlin `noise()` returns a random value between `0.0` and `1.0` at a specific time. You specify the start time. It is then incremented along this smooth random time line in steps; the smaller the steps (for instance, `0.005`) means there are very small changes but smoother, whereas larger steps (`0.03`) obviously create a much greater degree of randomness and possibly less smooth. It seems to work best between `0.005` and `0.03`.

If this seems strange, then don't let it be. If you have two variables that you want to have a different random outcome, you simply grab the `noise()` starting at different times, for example, `3` or `100`, etc.

The beauty of this is that it is random but it bears some relationship with the previous random number at a particular point in time. So if you move it on a small increment, you get a slight adjustment to the random value.

If you really want to understand how it works and who invented it (and why!), then I suggest you read up about it in Dan Shiffman's book/website [Nature of Code](#) or look at [Algorithmic Art module A unit #1 and #2](#) see under the resources tab ([elegantAI.org](#)).

Figure 1: perlin noise graph





Sketch D1.1 starting with a standard sketch

Starting with our normal basic sketch.

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

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



Sketch D1.2 randomly moving circle

We start the circle in the centre of the canvas and randomly move it. We are just going to use the `random()` function to move the circle around the canvas. You will notice that it is not very smooth or fluid; `noise` will give you something more natural.

```
let x = 200
let y = 200

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

function draw()
{
  background(220)
  circle(x, y, 100)
  x = x + random(-2, 2)
  y = y + random(-2, 2)
}
```

Notes

You could also use:

```
x += random(-2, 2)
y += random(-2, 2)
```

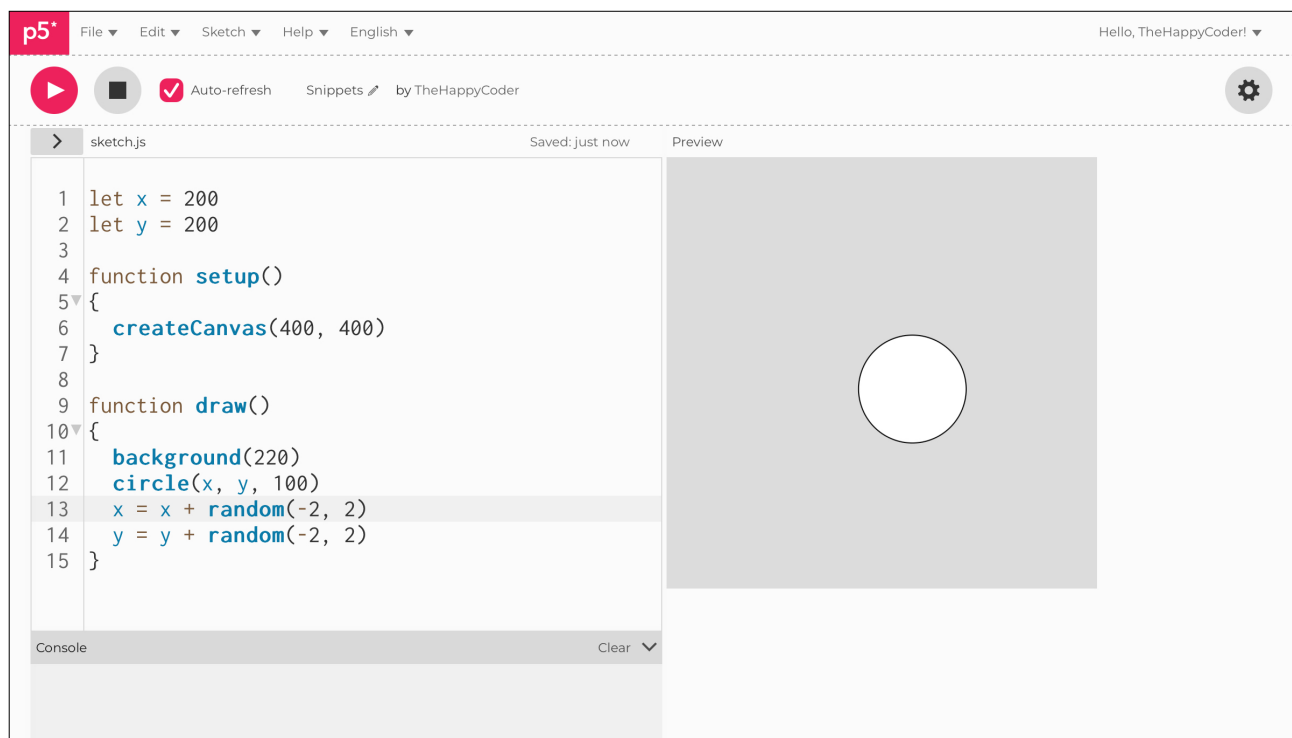
Challenges

1. Try random values of (5, -5).
2. Try different values for x and y.

Code Explanation

<code>x = x + random(-2, 2)</code>	Add a random value between -2 and 2 to the x value on each iteration
<code>y = y + random(-2, 2)</code>	Add a random value between -2 and 2 to the y value on each iteration

Figure D1.2





Sketch D1.3 smooth random movement

We have replaced the `random()` function with the `noise()` function. Notice that the jerkiness has gone, replaced by a much smoother movement, almost as if floating in the air. Also, it looks a lot more complicated. We have two start times (3 and 10). Because noise returns values between 0 and 1, we use `map()` to scale the movement up to the width and height of the canvas. Then, on each iteration, we move along the time line by 0.005.

```
let timeX = 3
let timeY = 10

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

function draw()
{
  background(220)
  let x = map(noise(timeX), 0, 1, 0, width)
  let y = map(noise(timeY), 0, 1, 0, height)
  circle(x, y, 100)
  timeX = timeX + 0.005
  timeY = timeY + 0.005
}
```

Notes

To see how noise works and why it is so much better than just `random()`, this short programme illustrates the smoothness of the movement.

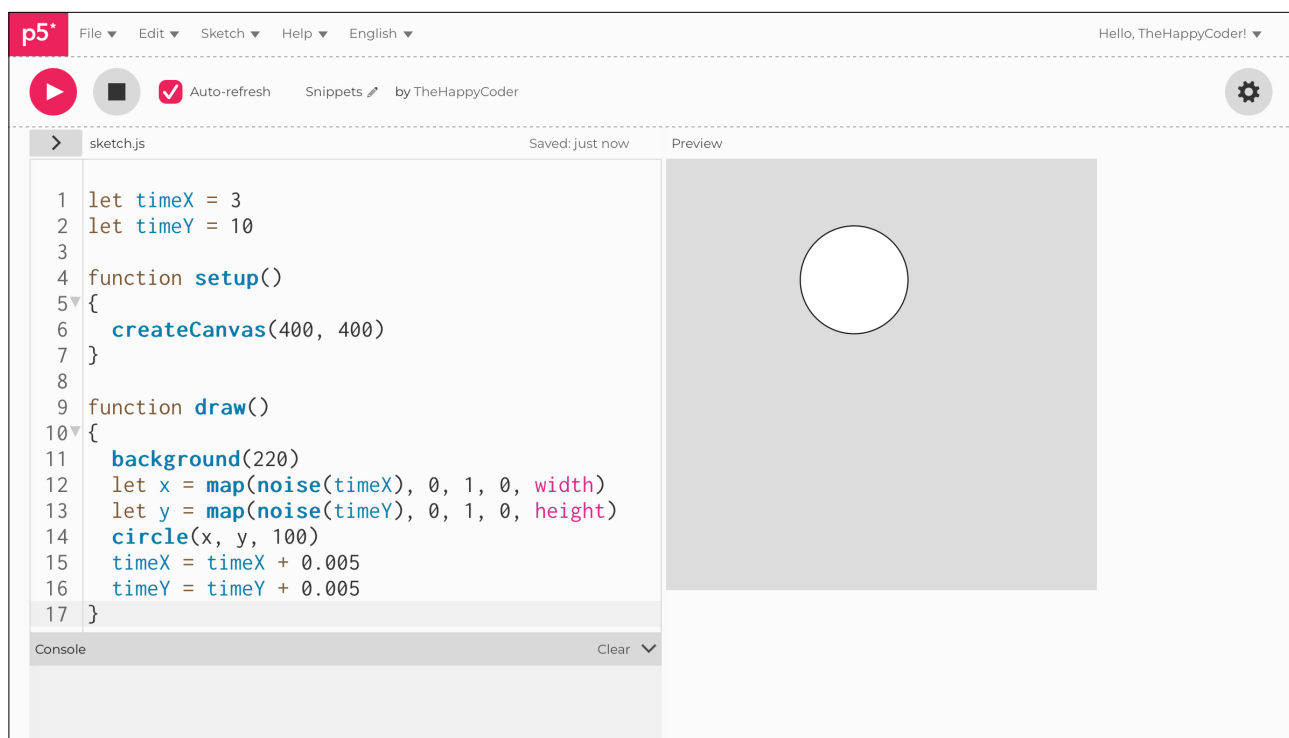
Challenges

1. Try different increments for `timeX` and `timeY`.
2. What happens if you give them both the same start on the timeline?
3. Try: `timeX = timeX + 0.05`.
4. Replace it with `timeX += 0.005`.

Code Explanation

<code>let timeX = 3</code>	The starting value for the x timeline
<code>let timeY = 10</code>	The starting value for the y timeline
<code>let x = map(noise(timeX), 0, 1, 0, width)</code>	Maps the timeX value to the width of the canvas
<code>let y = map(noise(timeY), 0, 1, 0, height)</code>	Maps the timeY value to the height of the canvas
<code>timeX = timeX + 0.005</code>	Adds an increment to the timeX timeline
<code>timeY = timeY + 0.005</code>	Adds an increment to the timeY timeline

Figure D1.3





It is all about 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 or around it.

We can describe 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 D1.4 starting sketch

! We start a new sketch, as usual.

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

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



Sketch D1.5 a single car

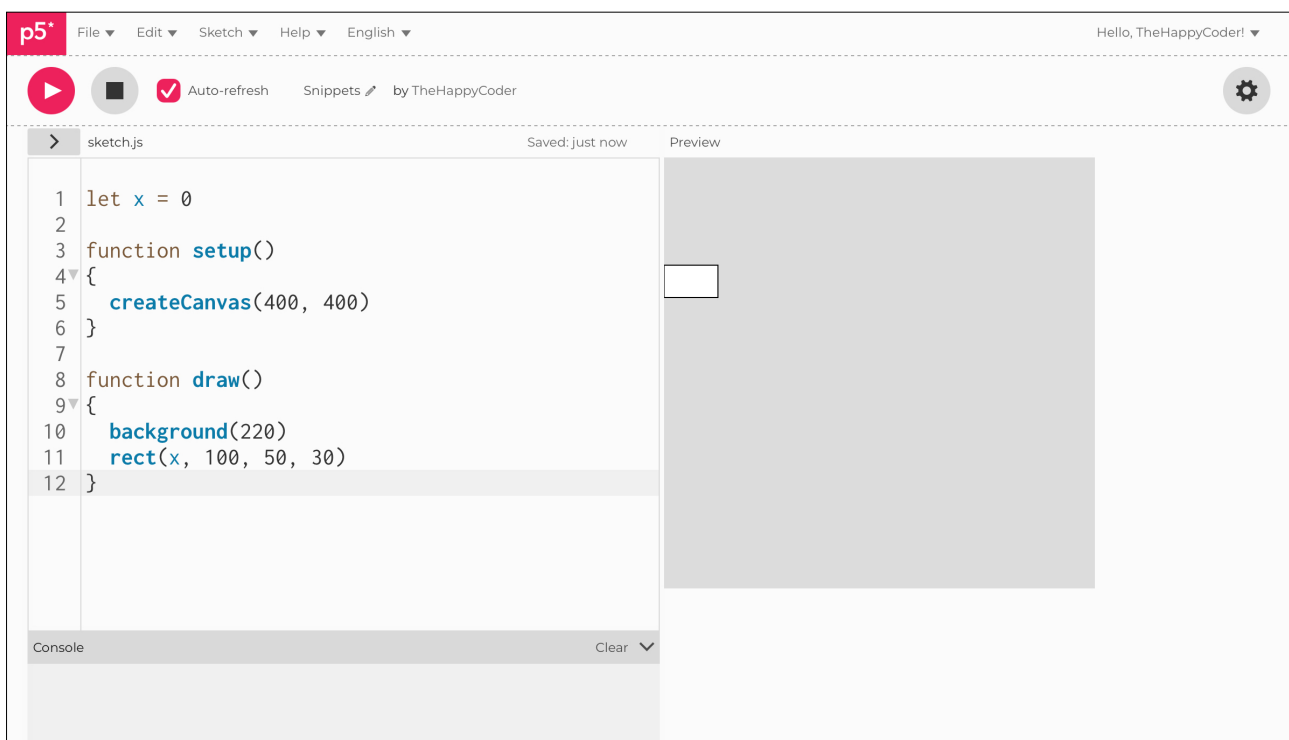
We create our car as a simple rectangle, starting at the left-hand edge of the canvas.

```
let x = 0

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

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

Figure D1.5





Sketch D1.6 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 off the right-hand edge of the canvas, never to be seen again.

Challenge

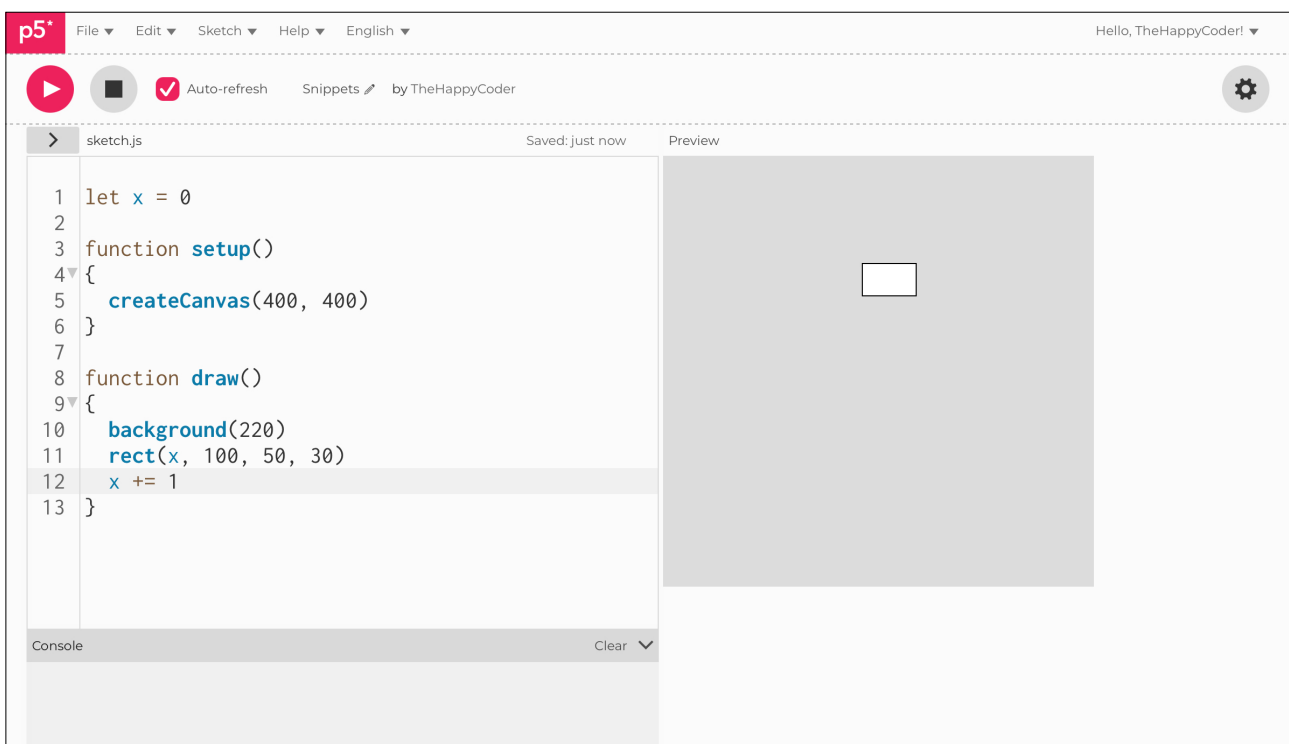
Make it move faster.

Code Explanation

```
x += 1
```

This adds 1 to x on each iteration

Figure D1.6





Sketch D1.7 a single car

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
  }
}
```

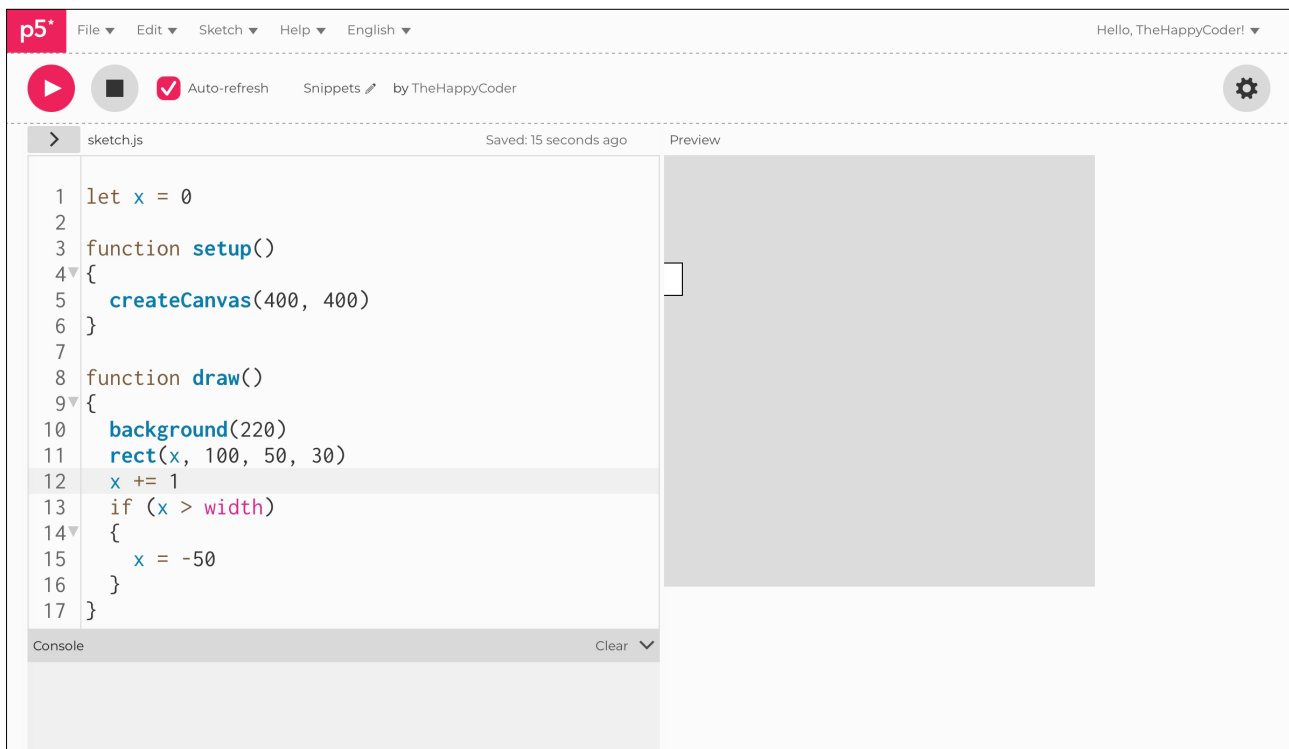
Notes

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 D1.7





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 D1.8 function single car

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 moved it around.

```
let x = 0

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

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

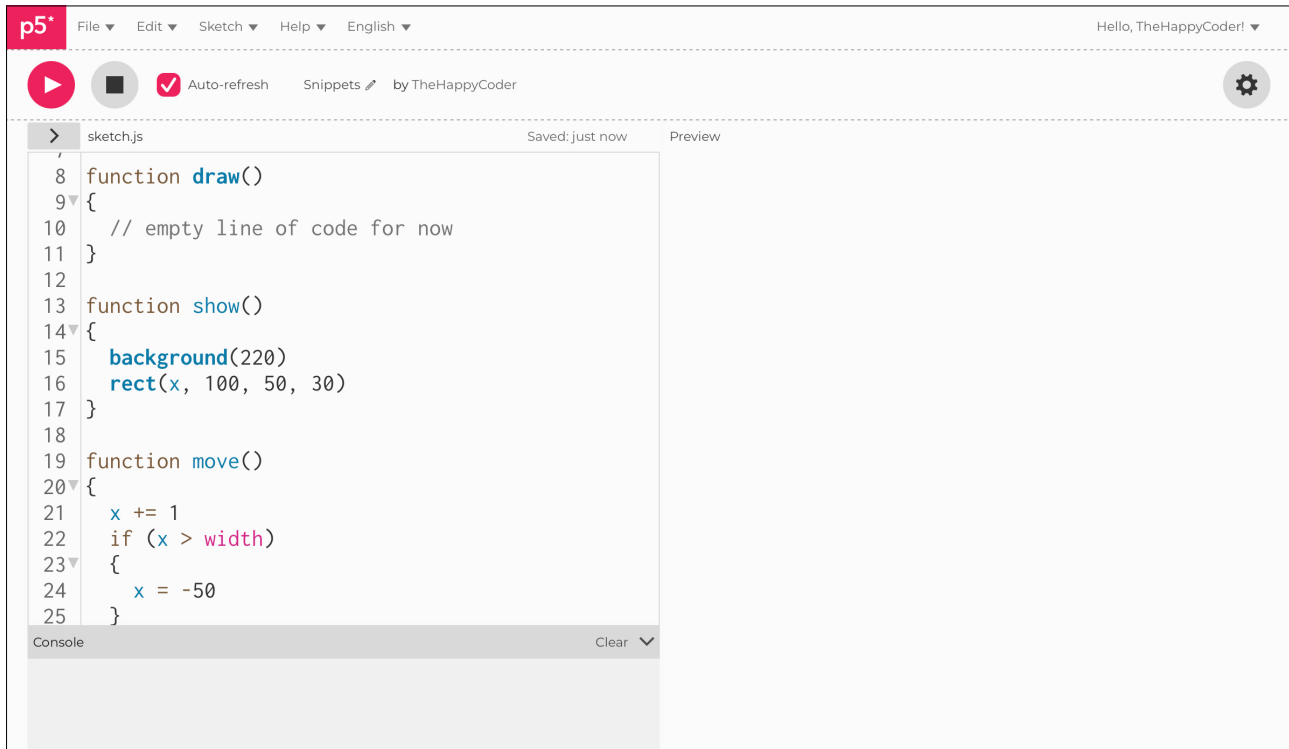
function show()
{
  background(220)
  rect(x, 100, 50, 30)
}

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

Notes

You will notice that nothing happens; there isn't even a canvas. You can just cut and paste to save time.

Figure D1.8





Sketch D1.9 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()
}

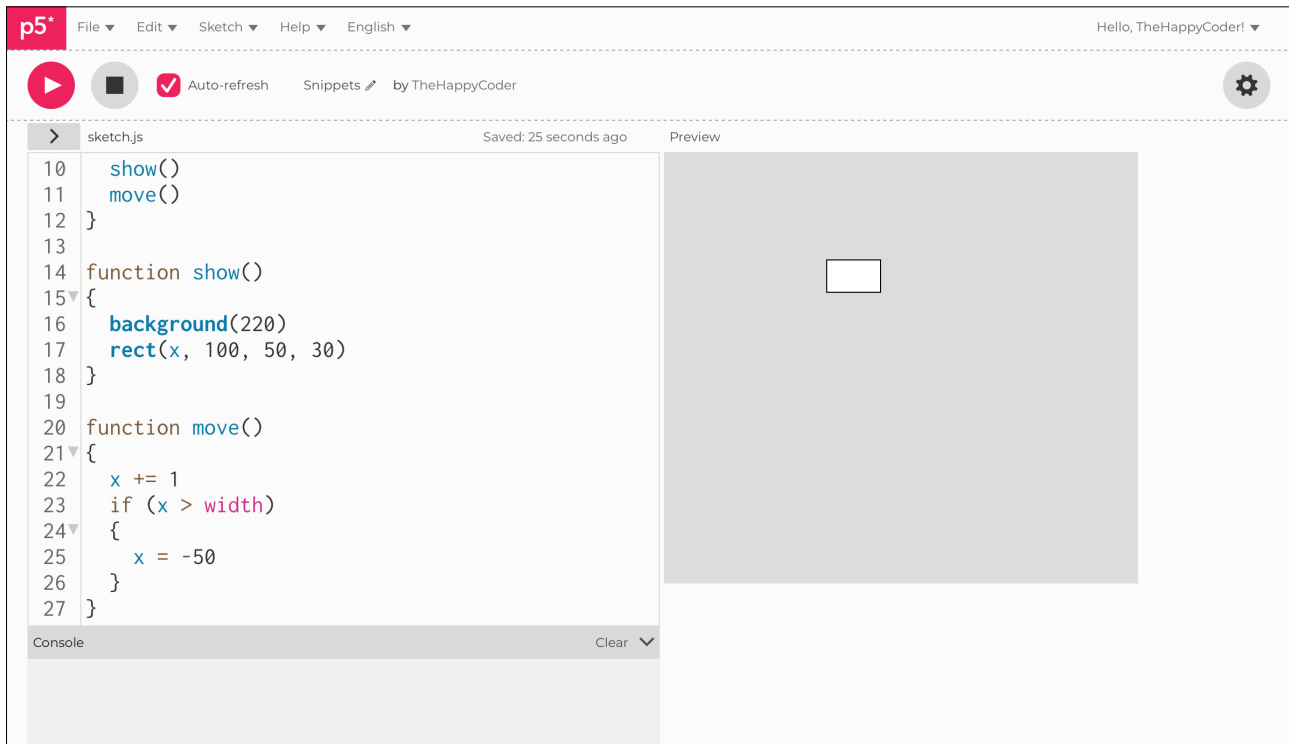
function show()
{
  background(220)
  rect(x, 100, 50, 30)
}

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 D1.9





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 D1.10 car as an object

! Start a new sketch (highlighted differences to basic sketch).

We have added the car as an object; notice the similarity to our earlier sketch, but now we have to give it a name.

```
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.

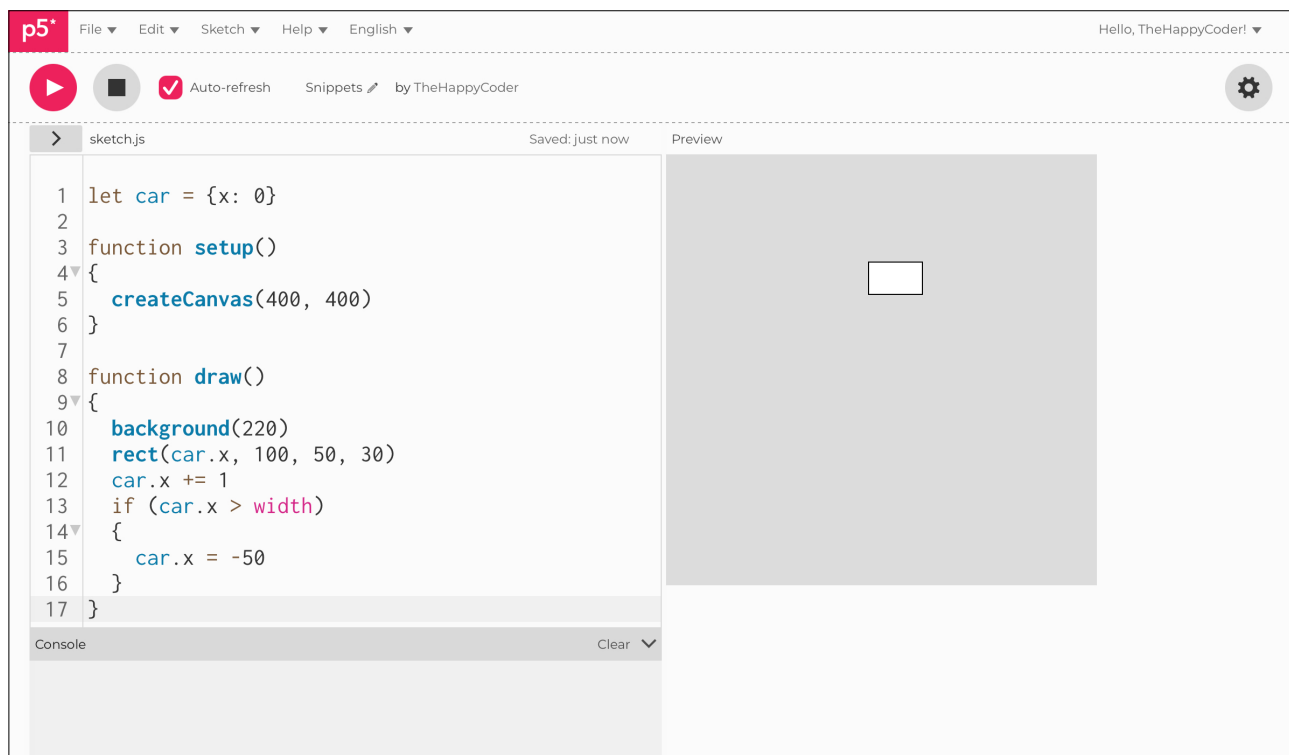
Challenges

1. Give it a **y** component.
2. Use the **show()** and **move()** functions (if struggling, see the 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 D1.10





Sketch D1.11 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.



Introduction to classes

Using classes is a common way to organise your 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 to understand.

It does take a bit of getting used to. I will try to illustrate this with a simple example. Imagine you have a template 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 (**figure 2**), 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.

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 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 **figure 3**. Where you create the car or cars from the class and call the functions within the class.

Figure 2: in the class

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

  show()
  {
    // what the car looks like
  }

  move()
  {
    // how the car will move
  }
}
```

Figure 3: in the main sketch

```
let car

function setup()
{
  car = new Car()
}

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



Sketch D1.12 the constructor function

! New Sketch.

We start with our basic sketch and create 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 power of using classes but a very gentle introduction to creating a 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
  }
}
```

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



Sketch D1.13 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>



Sketch D1.14 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 lefthand edge



Sketch D1.15 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



Sketch D1.16 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.

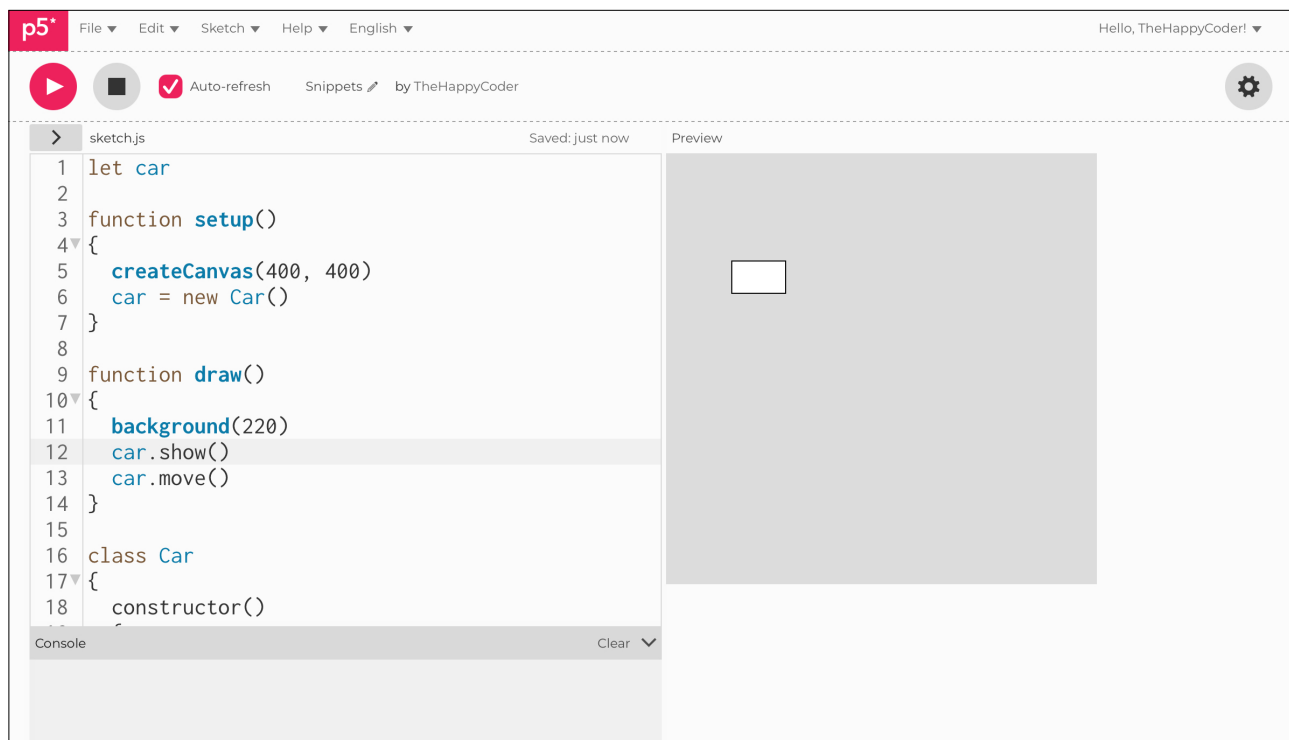
Challenges

1. Change the colour.
2. Change the **x** and **y** value.
3. Change the **velocity**.
4. Change the name of the variable to **myCar**.

Code Explanation

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

Figure D1.16





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 and independently.



Sketch D1.17 car attributes

When we create the car, we can specify its attributes rather than hard-code them in the `constructor()` function. We have to give 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 for our car. The beauty of this is that we can create a second (or more) car with different features.

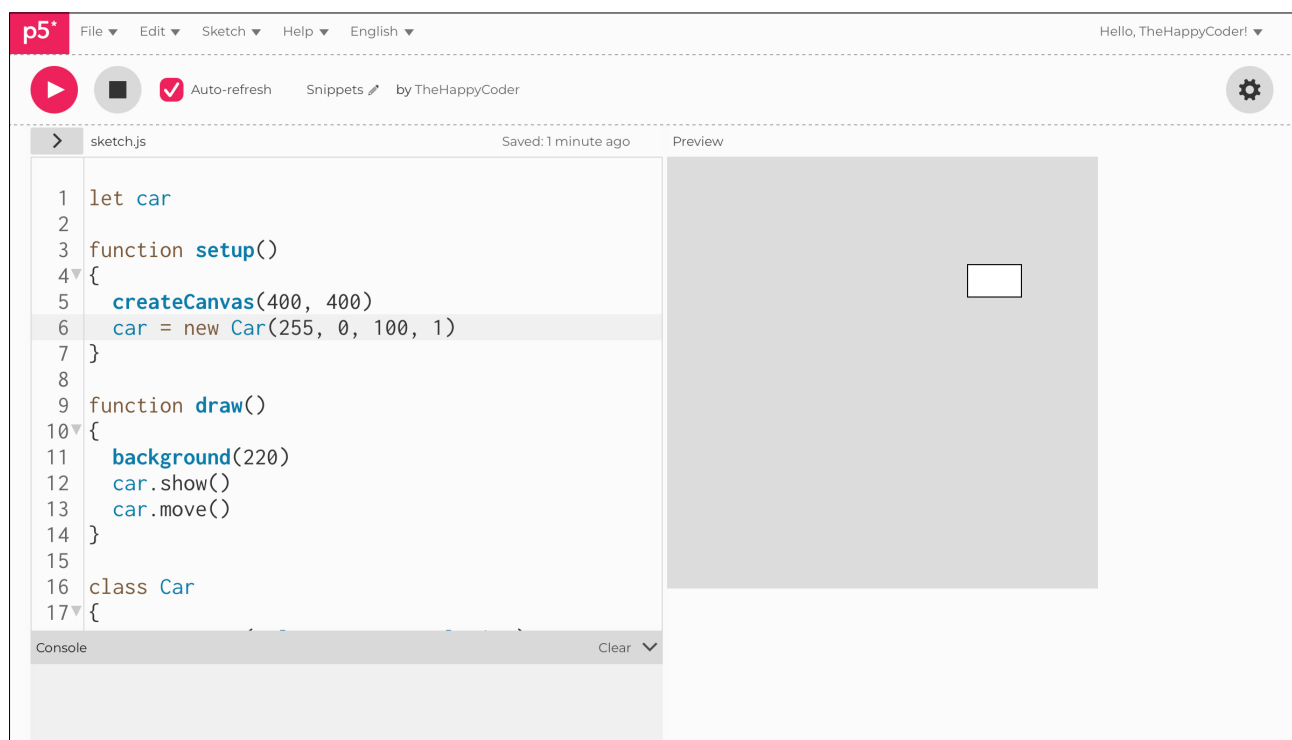
Challenge

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 D1.17





Sketch D1.18 a second car

We added a second car and gave 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
    }
}
```

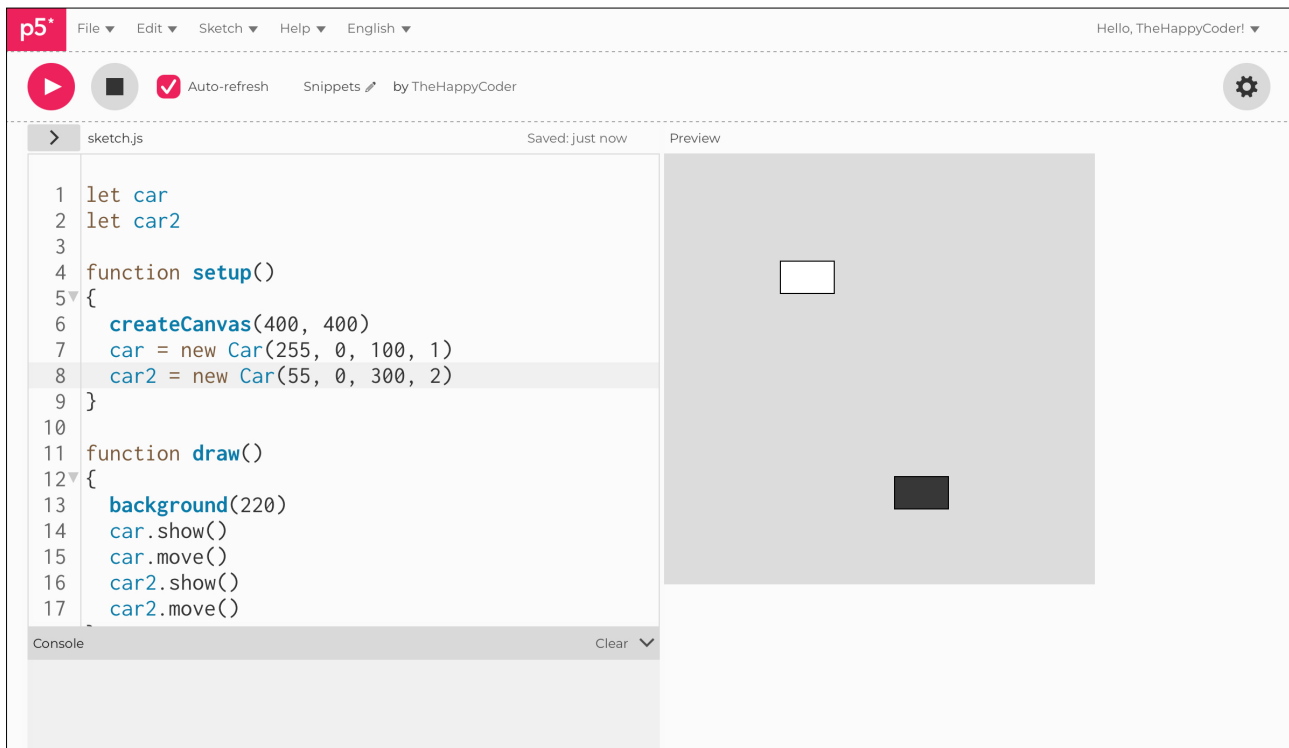
Notes

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 D1.18





Sketch D1.19 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 function; 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
  }
}
}

```

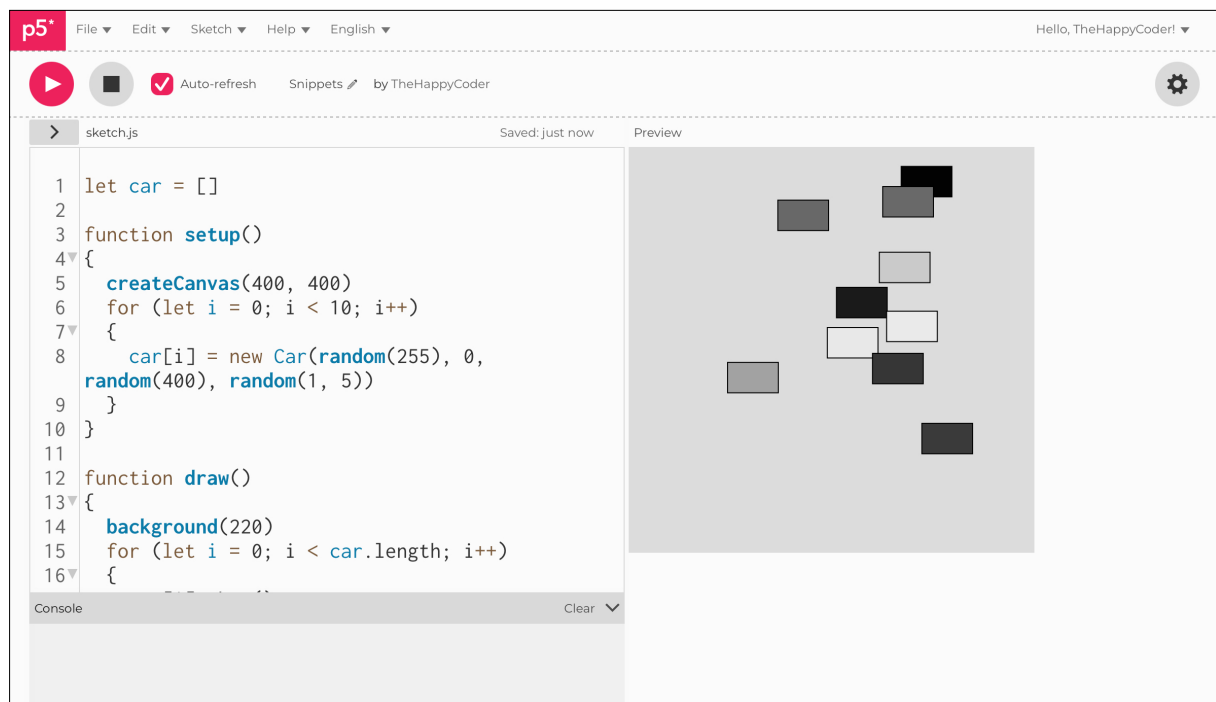
Notes

I think that is pretty elegant!

Challenge

Just have a play.

Figure D1.19





Seeking a target using classes

In this example, we are going to steer a vehicle towards a moving object (your mouse). This is in preparation for the **Neural Evolution** part of the AI tutorial. At the same time, we will have a more detailed look at using classes. For this, we will also add a file for the **Vehicle class**. This seemingly simple idea is quite a challenge.

If a vehicle is travelling in a certain direction with a particular speed, how does it steer towards a target so that it realistically changes direction rather than simply stopping, turning, and moving? The vehicle is going to seek the target and steer towards it.

The simple formula is described: $\text{steering} = (\text{desired velocity}) - \text{velocity}$.



Adding files

Before we go any further, we need to add a file called `vehicle.js`. I have already described how to add a file in the introductory unit, [a quick exploration of p5.js](#).

If you are still unfamiliar, I suggest going there quickly as a reminder and then returning here. In brief, you click on the `+` sign next to `Sketch Files`, which gives you a drop-down menu, and click on `Create file`. In the box, give it the name `vehicle.js`.

When you have done that, go to `index.html` and add in the line of code:

```
<script src="vehicle.js"></script>.
```

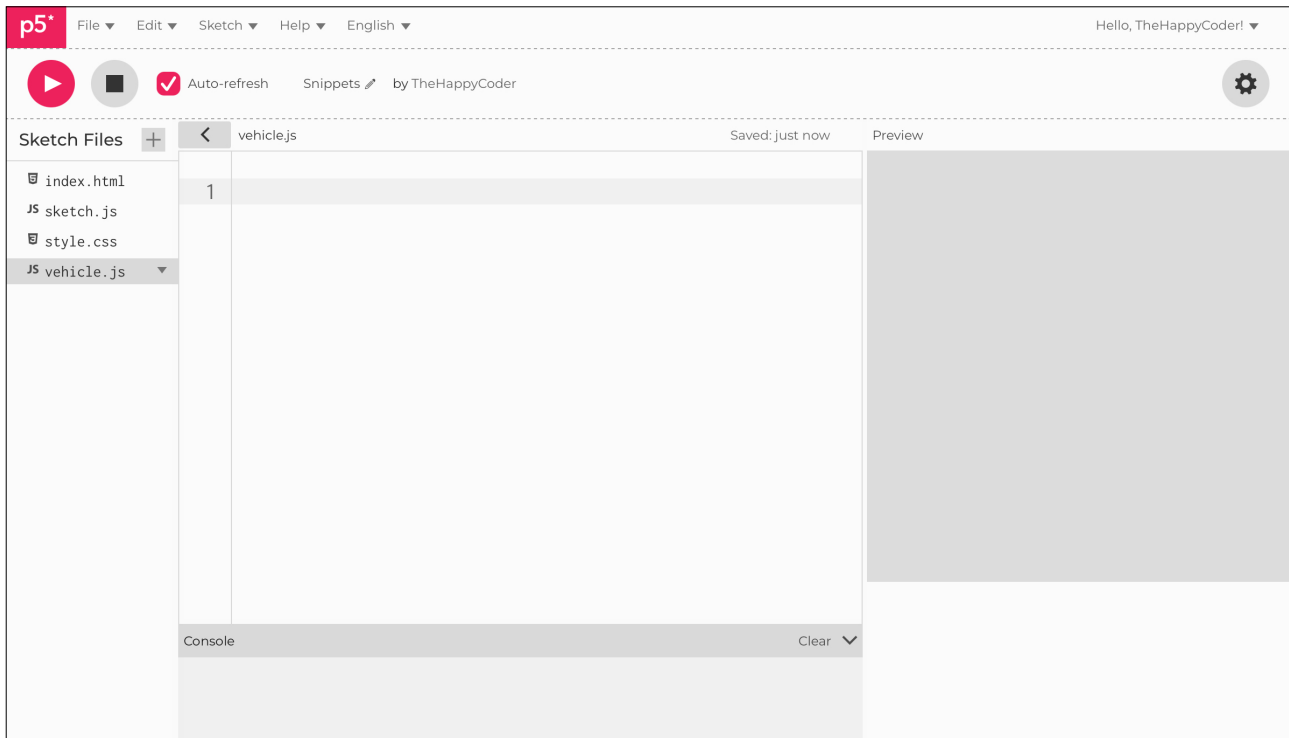
This last step is important; otherwise, your sketch won't know it is there. To access the `vehicle.js` sketch, you click on it in the `Sketch Files` drop-down menu.

You will be flitting between that file and the main `sketch.js` file.

Step 1

Creating a new file called `vehicle.js`, this is covered in a quick exploration of `p5.js`. Make sure that the spelling and case are correct (the `index.html` file is case-sensitive). You can call it anything you like, but you must be consistent, however.

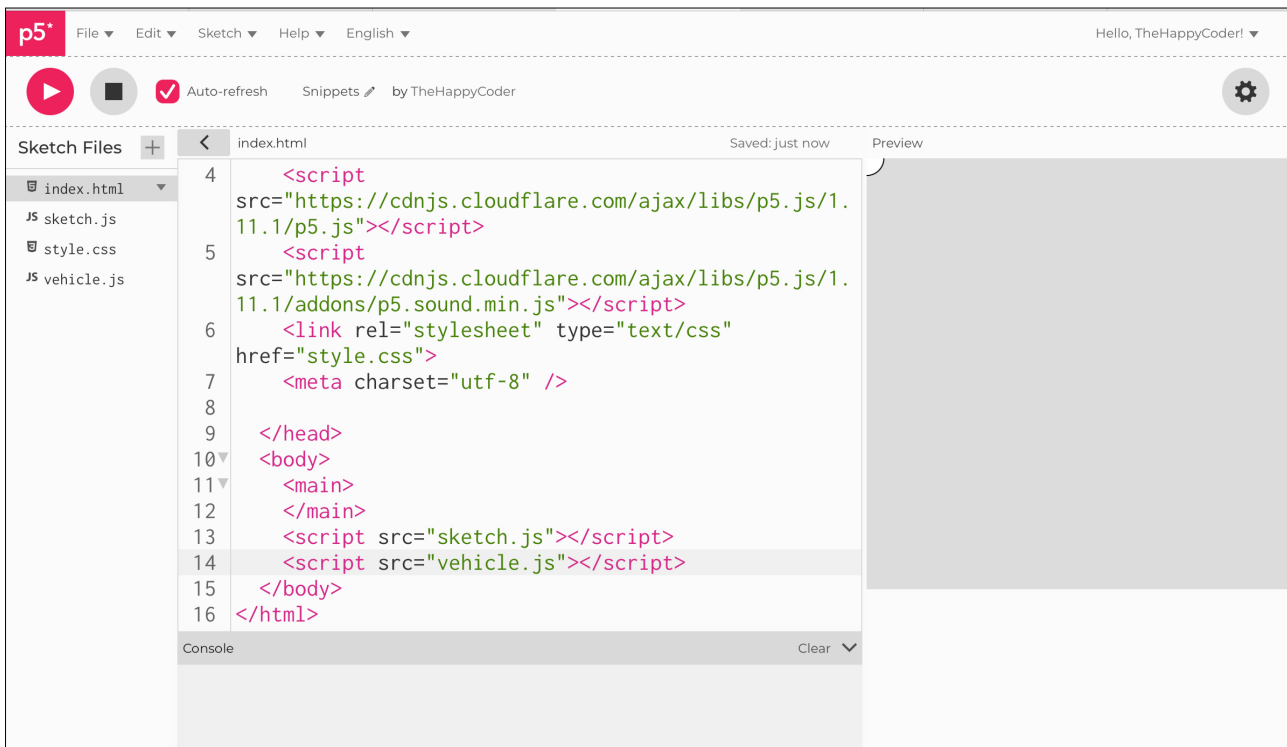
Figure 4: creating a new file called `vehicle.js`



Step 2

Referencing the file in the index.html file, you can simply copy and paste the line of code for `sketch.js` and change it to `vehicle.js`. You need to add it; otherwise, the `index.html` file won't know it is there, and so it remains invisible (it is easy to forget to do this bit after creating the file in the first place, but you soon find out!).

Figure 5: referencing the file in the index.html file





Sketch D1.20 index.html

This is what the `index.html` file looks like now. This step is easy to forget after creating it.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.10.0/p5.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.10.0/addons/p5.sound.min.js"></script>
    <link rel="stylesheet" type="text/css" href="style.css">
    <meta charset="utf-8" />

  </head>
  <body>
    <main>
    </main>
    <script src="sketch.js"></script>
    <script src="vehicle.js"></script>
  </body>
</html>
```

Notes

The spelling and case sensitivity are very important.



Sketch D1.21 new sketch with a target

! New sketch in `sketch.js` (it's based on our bog-standard sketch).

I will indicate whether we are in `sketch.js` or `vehicle.js` by referencing it on the top line of the coding box. You will need to flit from one to the other at times.

We create a target for our vehicle to seek; this is simply based on the `mouseX` and `mouseY` object positions.

sketch.js

```
let target

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

function draw()
{
  background(220)
  target = createVector(mouseX, mouseY)
  circle(target.x, target.y, 32)
}
```

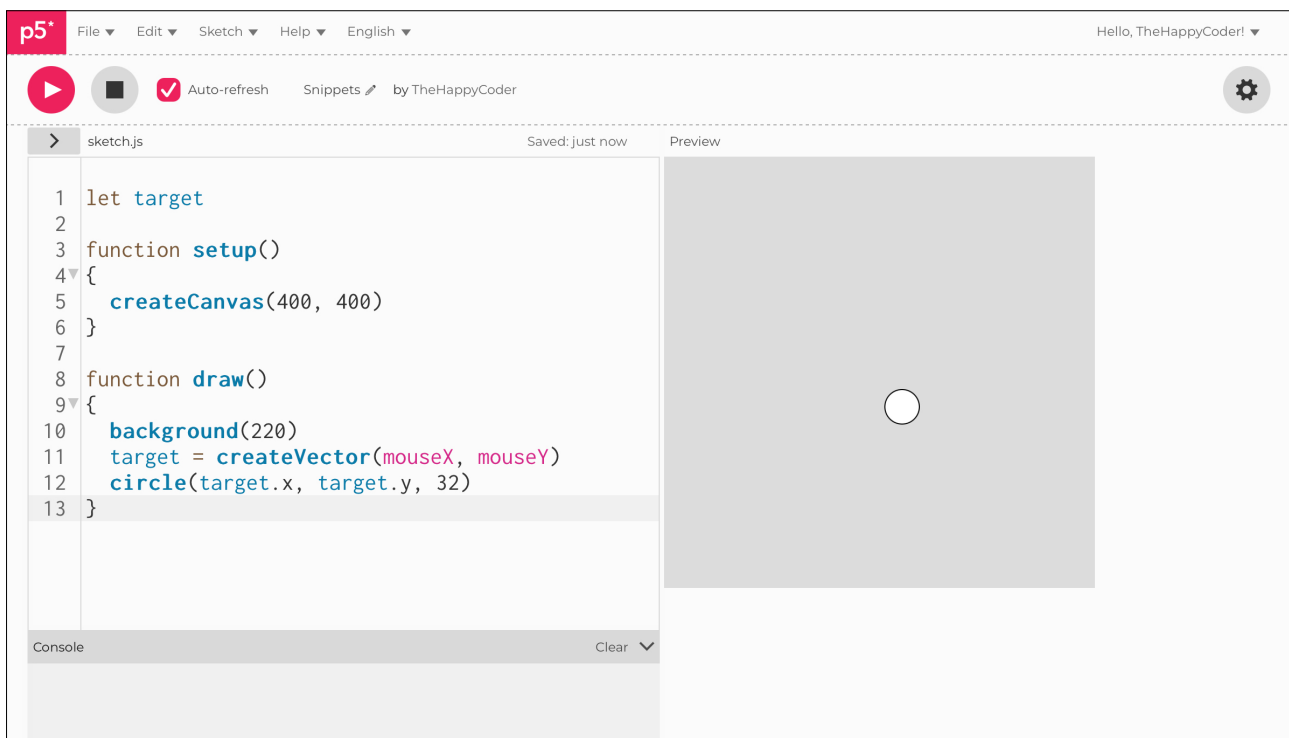
Notes

The circle (which we are calling the **target**) follows the movement of your mouse.

Code Explanation

<code>target = createVector(mouseX, mouseY)</code>	We create a vector with two elements the mouseX and the mouseY
<code>circle(target.x, target.y, 32)</code>	We draw a circle to that vector, this is our target

Figure D1.21





Sketch D1.22 the Vehicle class

! The `vehicle.js` sketch.

Here we create our `Vehicle` class. Notice it has a capital letter for its name. The `constructor()` function has two arguments: the `x` and `y` position of the vehicle. Nothing in the `show()` and `move()` functions yet.

vehicle.js

```
class Vehicle
{
  constructor()
  {

  }

  move()
  {

  }

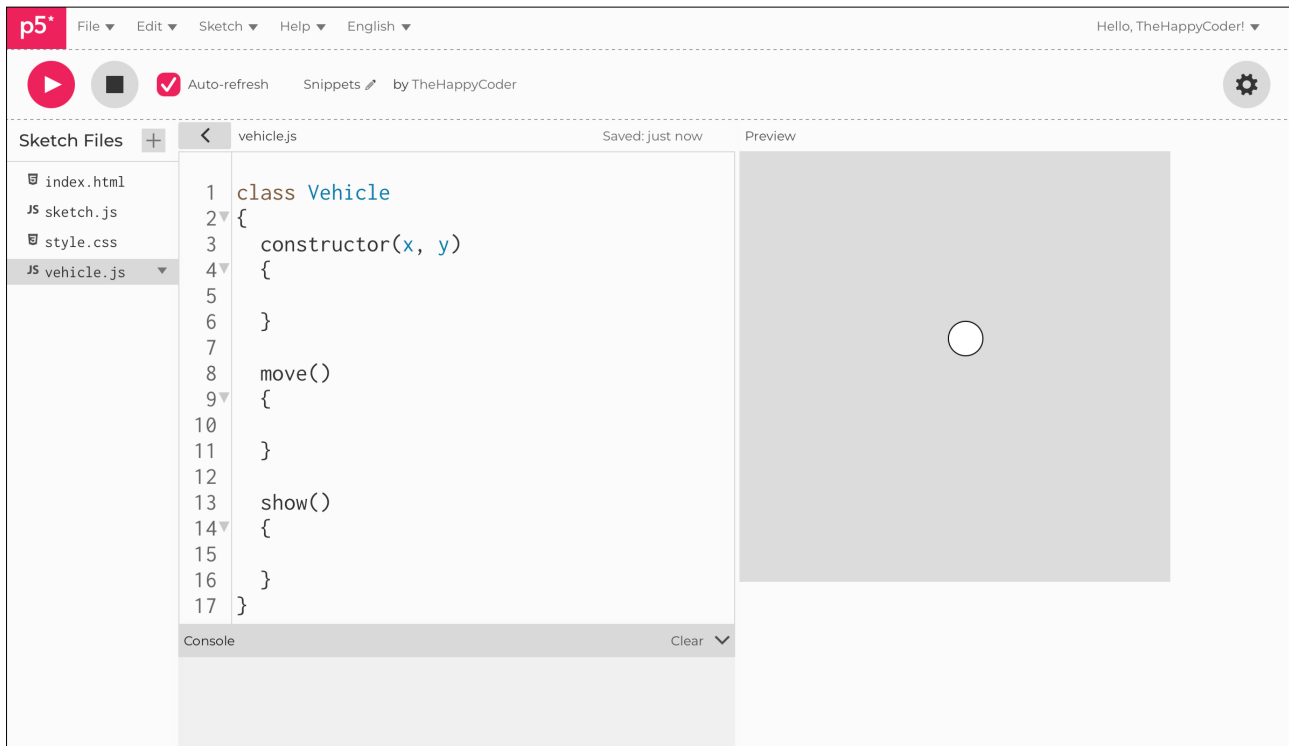
  show()
  {

  }
}
```

Notes

This is a gentle introduction; we will build it gradually. We have our **target** (mouse) and the **Vehicle** class, but nothing much else.

Figure D1.22





Sketch D1.23 a vehicle design

We want a triangle to represent our vehicle. We use the `triangle()` shape function to draw it. The dimensions of the triangle are based on a length `l` (which is `16`). We also create a vector for its position (`this.pos`). Then we translate the triangle by the position; this is because we have created a triangle based on the dimension of `l` and so we move it to the position we want at `x`, `y`. This is necessary for when we start to move the vehicle.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.l = 16
  }

  move()
  {

  }

  show()
  {
    translate(this.pos.x, this.pos.y)
    triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  }
}
```

Notes

We still won't see anything until we create the vehicle in the main sketch (`sketch.js`), so let's hop over there.

Challenge

To get a sense of the triangle's vertices, draw them out on a piece of paper.

Try different ways of using the length `l`.

Code Explanation

<code>translate(this.pos.x, this.pos.y)</code>	This uses the x and y coordinates and translates the triangle to that position
<code>triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)</code>	This spreads each vertex for each corner of the triangle based on the x, y position and the length <code>l</code>



Sketch D1.24 a starting position

! Go to `sketch.js`.

We create a vehicle from the class template and put it in the position of (100, 100).

sketch.js

```
let target
let vehicle

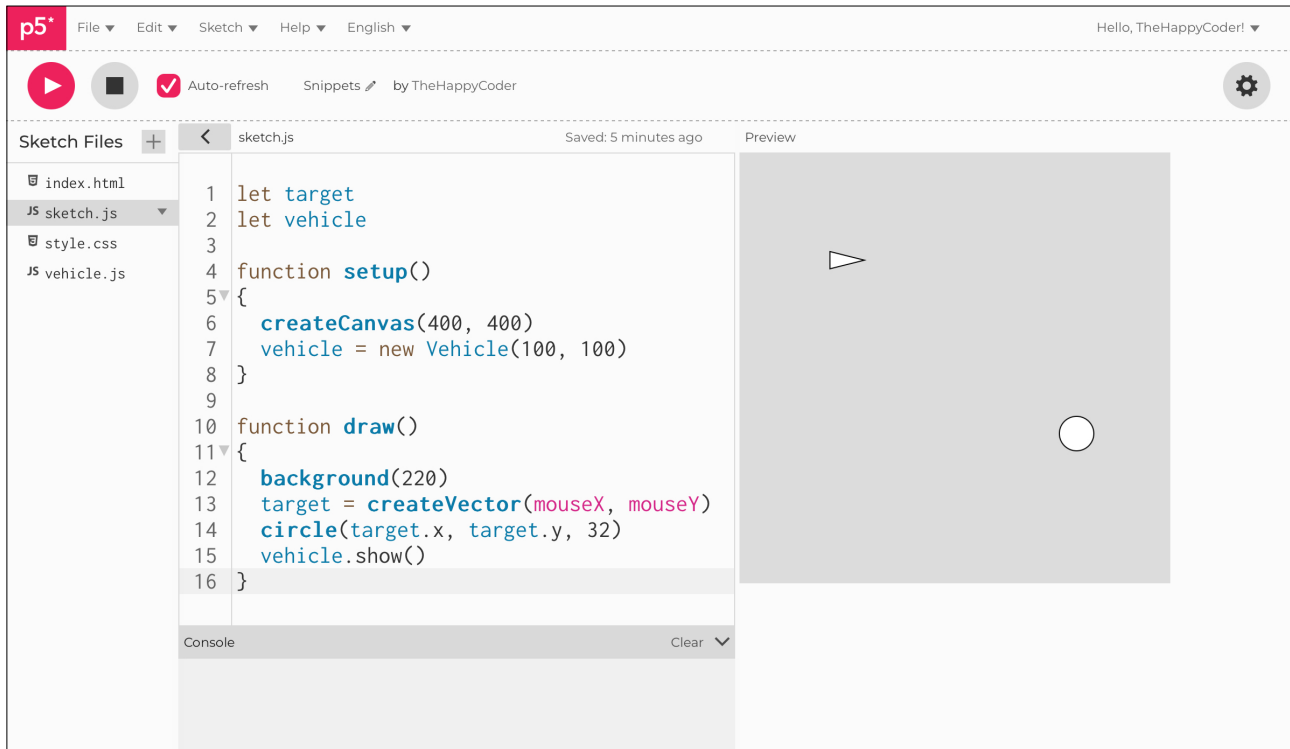
function setup()
{
  createCanvas(400, 400)
  vehicle = new Vehicle(100, 100)
}

function draw()
{
  background(220)
  target = createVector(mouseX, mouseY)
  circle(target.x, target.y, 32)
  vehicle.show()
}
```

Notes

We have our **target** (mouse) and a **vehicle**, but it isn't moving. We need to look at that next.

Figure D1.24





Movement with vectors

With vectors, we describe movement a little bit differently. We could have a whole unit just on this, but for now, I will go through it briefly and simply. If you want to dive a little deeper, then I suggest reading Dan Shiffman's book [The Nature of Code](#), which is also available as a web page.

There are three elements to the movement of a vector object. They are:

position
velocity
acceleration

In short, the acceleration ([acc](#)) is added to the velocity ([vel](#)), which in turn is added to the position ([pos](#)). We start off creating three vectors for each of these components. We may (or may not) give them any initial values.

The acceleration will be the force exerted on the object by the position of the target. Imagine the vehicle moving in a certain direction and the target being at some angle to its path of movement. It will want to turn towards that target. So there is a force acting on the vehicle to turn it from moving in its current direction towards the target. It could just stop turning and move, but if the target is moving and the vehicle is already moving in a particular direction, then the correction is a force in the direction of the target. If that makes sense. This is why later on we apply a force.



Sketch D1.25 velocity and acceleration

! Now back to `vehicle.js`.

The velocity (`vel`) and acceleration (`acc`) vector components are added and are initialised to zero.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)

    this.l = 16
  }

  move()
  {

  }

  show()
  {
    translate(this.pos.x, this.pos.y)
    triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  }
}
```

Notes

This needs more to have any impact on what we are doing; again, we are building it up slowly.

Code Explanation

<code>this.vel = createVector(0, 0)</code>	Velocity is zero in the x and y direction
<code>this.acc = createVector(0, 0)</code>	Acceleration is zero in the x and y direction



Sketch D1.26 adding acc, vel and pos

It may seem a bit backwards, but we first add the acceleration to the velocity, then add the velocity to the position.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
  }

  show()
  {
    translate(this.pos.x, this.pos.y)
    triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  }
}
```

Notes

This means the vehicle doesn't move.

Code Explanation

<code>this.vel.add(this.acc)</code>	We are adding the acceleration to the velocity
<code>this.pos.add(this.vel)</code>	We are adding the velocity to the position



Sketch D1.27 applying a force

The acceleration is a force applied to the vehicle to turn it. We add a new function to the **Vehicle** class called **applyForce()** with an argument which will carry the value of the **force** applied. This **force** is added to the acceleration. This **force** will depend on where the vehicle is and where the target is.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
  }

  show()
  {
    translate(this.pos.x, this.pos.y)
    triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  }
}
```

Notes

Building it up gradually.

Code Explanation

<code>applyForce(force)</code>	This is the new function in the Vehicle class that will apply the force to change the acceleration
<code>this.acc.add(force)</code>	The force is a vector that will change the acceleration vector



Sketch D1.28 seeking the target

We want a function to seek the **target**, so we add one to the Vehicle class. The **target** is already a vector (**mouse.x**, **mouse.y**). The **desired** is a vector which is the difference (subtract) between the target and where the **vehicle** is currently (**this.pos**).

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
  }

  seek(target)
  {
    let desired = p5.Vector.sub(target, this.pos)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
  }

  show()
  {
    translate(this.pos.x, this.pos.y)
    triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  }
}
```



```
}
```

Code Explanation

<code>seek(target)</code>	The function receives an argument the target vector
<code>let desired = p5.Vector.sub(target, this.pos)</code>	Subtracts the two vectors, target and the position of the vehicle



Sketch D1.29 seek, move and show

! Move back to `sketch.js`.

We will add three functions: `seek()`, `move()`, and `show()`. These are the final changes to the main sketch.

sketch.js

```
let target
let vehicle

function setup()
{
  createCanvas(400, 400)
  vehicle = new Vehicle(100, 100)
}

function draw()
{
  background(220)
  target = createVector(mouseX, mouseY)
  circle(target.x, target.y, 32)
  vehicle.seek(target)
  vehicle.show()
  vehicle.move()
}
```

Notes

As yet, there is still nothing to see; we still have a bit more to do in the `Vehicle` class, so this is where we are heading next.



Sketch D1.30 reset acceleration

! Head back to `vehicle.js`.

We are going to reset the acceleration; otherwise, it will keep accumulating. We just want the actual current acceleration on each iteration in the `draw()` function in `sketch.js`.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
  }

  seek(target)
  {
    let desired = p5.Vector.sub(target, this.pos)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.set(0, 0)
  }

  show()
  {
    translate(this.pos.x, this.pos.y)
    triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  }
}
```

```
}  
}
```

Notes

There is a logic to this.

Code Explanation

```
this.acc.set(0, 0)
```

Resets the acceleration to zero so that it doesn't keep accumulating



Sketch D1.31 steering

We want to **steer** the **vehicle** towards the **target**; the **steering** vector is the difference between the desired and the actual motion of the vehicle.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
  }

  seek(target)
  {
    let desired = p5.Vector.sub(target, this.pos)
    let steering = p5.Vector.sub(desired, this.vel)
    this.applyForce(steering)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.set(0, 0)
  }

  show()
  {
    translate(this.pos.x, this.pos.y)
    triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  }
}
```

```
}  
}
```

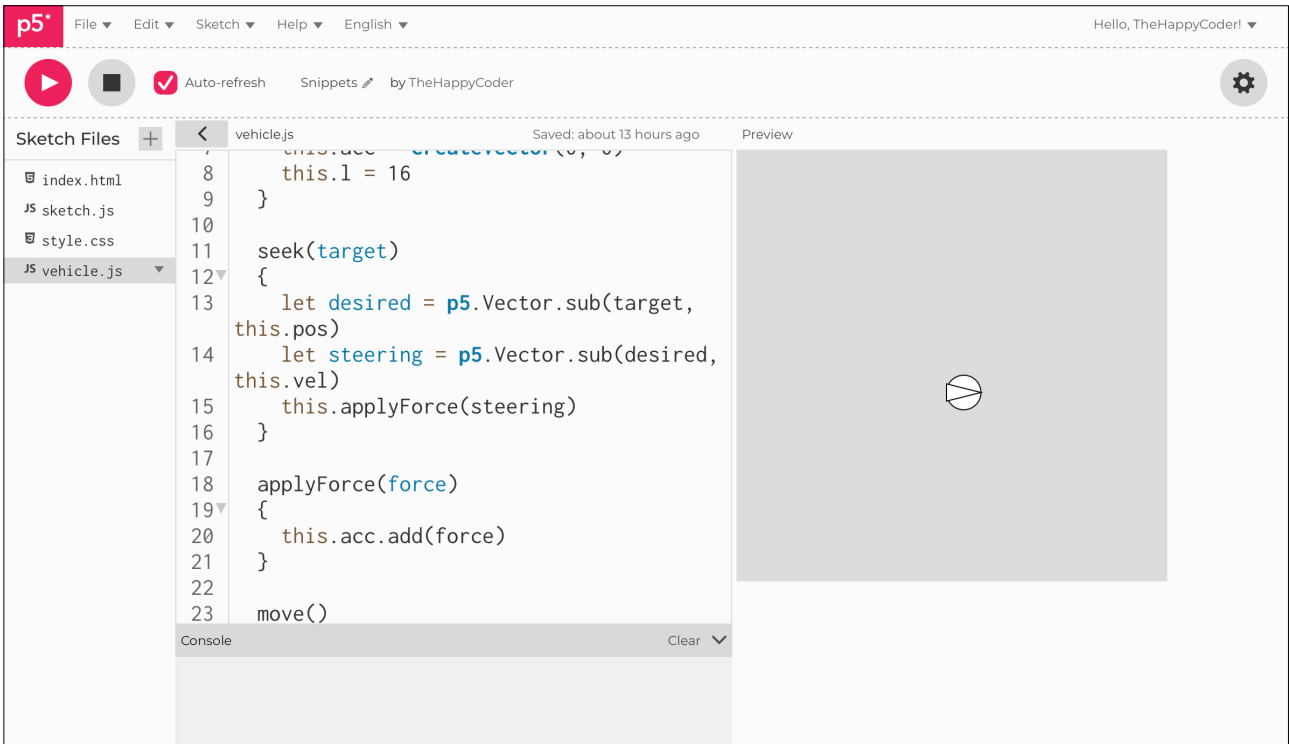
Notes

What you get is the vehicle appearing where the target is. We are building step by step!

Code Explanation

<code>let steering = p5.Vector.sub(desired, this.vel)</code>	The steering is a vector which is the desired vector minus the vehicle velocity vector
<code>this.applyForce(steering)</code>	This vector is the force applied to the vehicle to turn it towards the target

Figure D1.31





Sketch D1.32 rotate to the heading

The function `heading()` calculates the angle between an axis and the vector. We use `push()` and `pop()` to isolate the movement of that particular vehicle.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
  }

  seek(target)
  {
    let desired = p5.Vector.sub(target, this.pos)
    let steering = p5.Vector.sub(desired, this.vel)
    this.applyForce(steering)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.set(0, 0)
  }

  show()
  {
    translate(this.pos.x, this.pos.y)
  }
}
```

`push()`

```
rotate(this.vel.heading())
triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
pop()
}
}
```

Notes

We don't want to simply push the **vehicle** towards the **target**; we want it to turn (**rotate**) and face the direction of travel.

Code Explanation

```
rotate(this.vel.heading())
```

Calculates the angle a 2D vector makes with the positive x-axis and rotates it



Sketch D1.33 getting it working

We need to limit the magnitude of the velocity of the vehicle; we do that with the `setMag()` function. We give it an arbitrary maximum velocity (`maxVelocity`) of 4. What you should get is the vehicle moving towards the target rather than appearing instantly on top of it.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
    this.maxVelocity = 4
  }

  seek(target)
  {
    let desired = p5.Vector.sub(target, this.pos)
    desired.setMag(this.maxVelocity)
    let steering = p5.Vector.sub(desired, this.vel)
    this.applyForce(steering)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.set(0, 0)
  }
}
```

```

show()
{
  translate(this.pos.x, this.pos.y)
  push()
  rotate(this.vel.heading())
  triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  pop()
}
}

```

Notes

What you will now see is the vehicle moving towards the target when you move the target by clicking on the canvas.

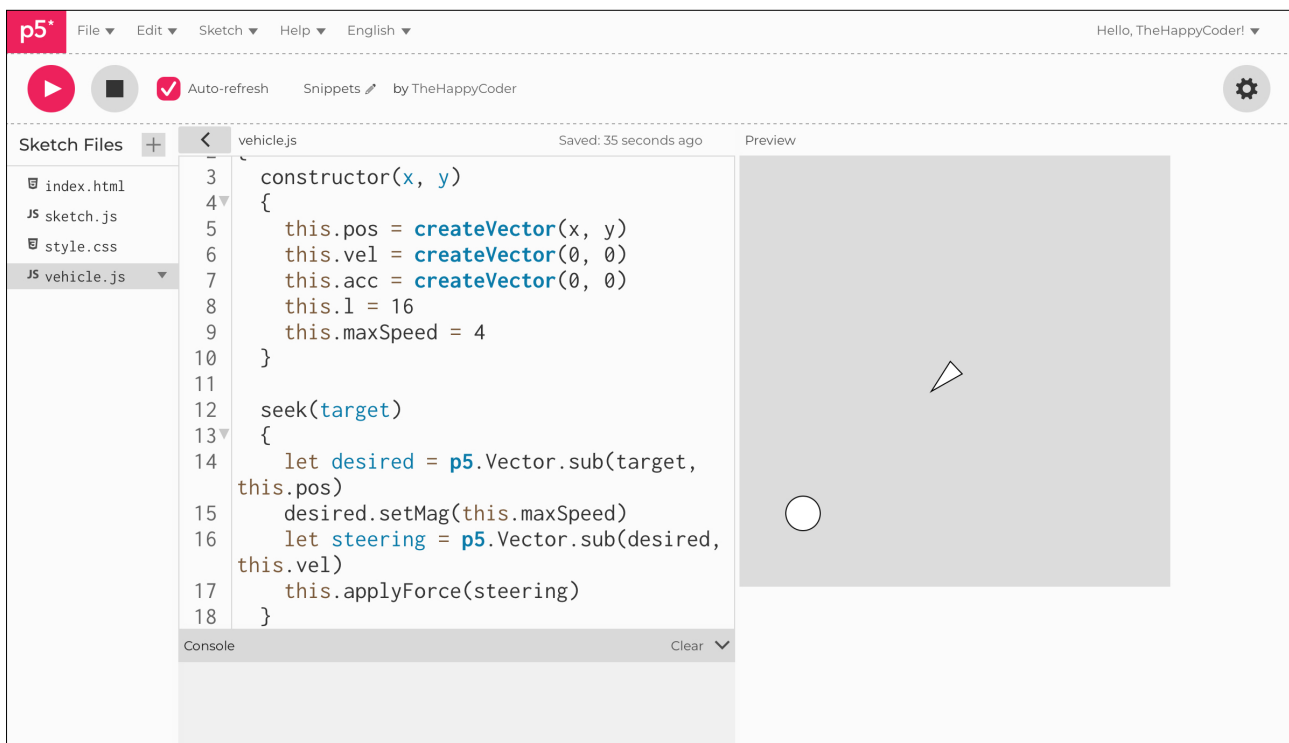
Challenge

Try other values for `maxVelocity`.

Code Explanation

<code>this.maxVelocity = 4</code>	As it says this is the maximum velocity
<code>desired.setMag(this.maxVelocity)</code>	Sets the magnitude of the desired

Figure D1.33





Sketch D1.34 the maximum force

To make it more realistic, rather than just finding the quickest way to get to the target, we limit the maximum force applied to the vehicle (**maxForce**) to **0.01**. To do that, we use a built-in function called **limit()**, which does exactly that. This means it arcs towards the target rather than just stopping, turning, and moving.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
    this.maxVelocity = 4
    this.maxForce = 0.01
  }

  seek(target)
  {
    let desired = p5.Vector.sub(target, this.pos)
    desired.setMag(this.maxVelocity)
    let steering = p5.Vector.sub(desired, this.vel)
    steering.limit(this.maxForce)
    this.applyForce(steering)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.set(0, 0)
  }
}
```

```
}

show()
{
  translate(this.pos.x, this.pos.y)
  push()
  rotate(this.vel.heading())
  triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  pop()
}
}
```

Notes

We see a much more restrained vehicle, with more sweeping arcs rather than point and move.

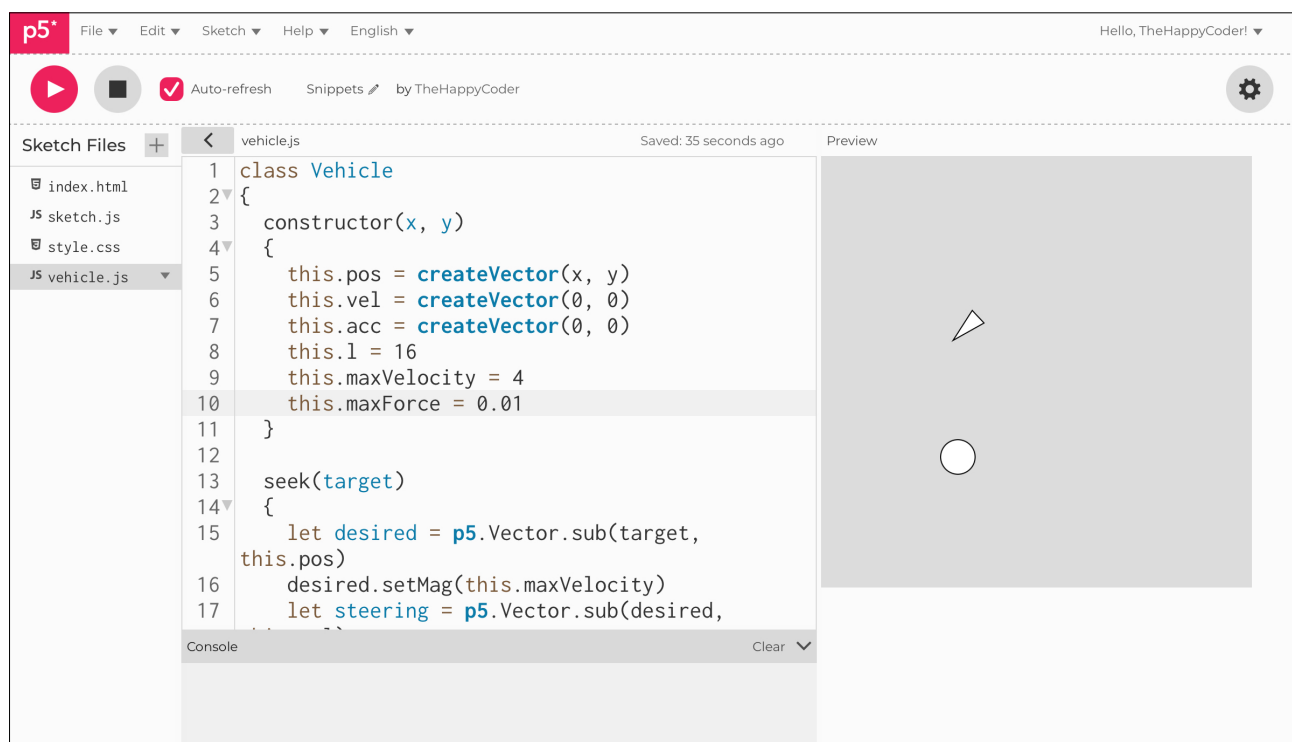
Challenges

1. Try different values of `maxVelocity`.
2. What happens if you remove the `setMag()` line of code (comment it out)?
3. Try different values of maximum force.
4. What happens if you remove the `limit()` line of code (comment it out)?

Code Explanation

<code>this.maxForce = 0.01</code>	Helps to control the acceleration towards the target
<code>steering.limit(this.maxForce)</code>	Limits the amount of force

Figure D1.34





Sketch D1.35 the force is now steering

Changing **desired** to **force** and removing **steering**, this whole **seek(target)** function has a revamp so that the **force** does the steering. This doesn't change the functionality of the code, but you can see that there is more than one way to approach the problem.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
    this.l = 16
    this.maxVelocity = 4
    this.maxForce = 0.01
  }

  seek(target)
  {
    let force = p5.Vector.sub(target, this.pos)
    force.setMag(this.maxVelocity)
    force.sub(this.vel)
    force.limit(this.maxForce)
    this.applyForce(force)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.set(0, 0)
  }
}
```

```

}

show()
{
  translate(this.pos.x, this.pos.y)
  push()
  rotate(this.vel.heading())
  triangle(-this.l, -this.l/2, -this.l, this.l/2, this.l, 0)
  pop()
}
}

```

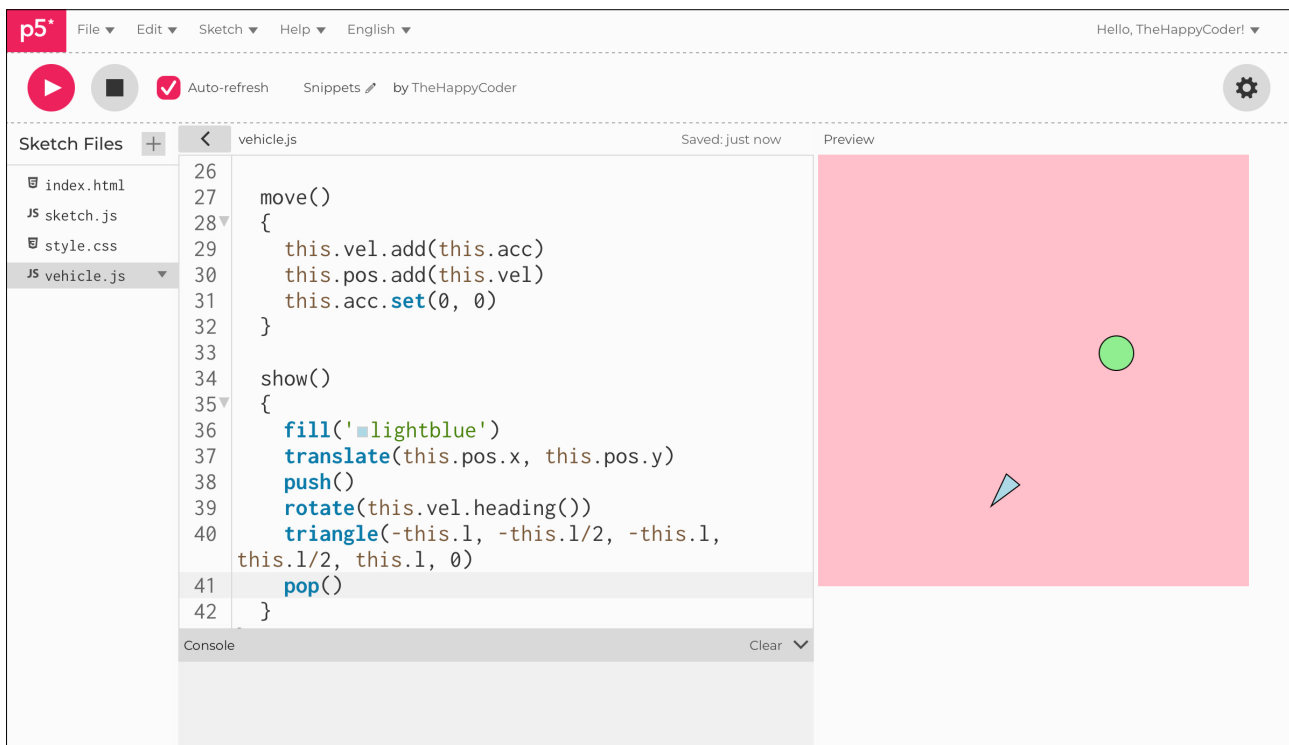
Notes

This is an alternative for you to wrap your head around; it does look a lot more elegant.

Challenges

1. Play around with the values of `maxVelocity` and `maxForce` to get something different.
2. Add sliders for the two variables.
3. Add some colour.

Figure D1.35: with a dash of colour



Algorithmic Intelligence Module D Unit #2 neural evolution explained



Module D Unit #2: neural evolution?

Early on, I described three basic types of learning:

Supervised Learning

Unsupervised Learning

Reinforcement Learning

Neural evolution is a form of reinforcement learning. It is a method whereby a model is not trained on any data but is rewarded by its performance. What we are going to look at is a particular branch of reinforcement learning called Genetic Algorithms.

A genetic algorithm is an approach whereby we use an evolutionary approach to solving a problem. We select the best-performing pairs of models and mate them; their offspring will be better able to tackle the problem we give them.

You could see it as an evolutionary approach of the survival of the fittest. We ascribe a greater fitness level to the model that performs well, a kind of reward. The better it does, the higher the reward.

In a genetic algorithm, this process is repeated until we have a population of objects that perform a task very well, even though they have been given no data about what they are doing or why they are doing it.

You don't need to use a neural network to do this, but neural evolution is an approach whereby we give each object a random model (or brain) and see how it performs. We do the same process of selecting the best ones and mating (crossover) them to produce a child that should have the attribute that we want.

It is important to stress at this point that we do not train it on any data; we don't even change the model; we are simply selecting pairs of well-performing models and using a function called `crossover()`, which is available in `ml5.js`, to produce the next generation of offspring. We will dive into this in a little more detail later.



Smart cars and flappy birds

We are going to look at two examples:

Smart cars

This is using the `seek()` function you coded in [code snippets 4](#) previously, as well as the Perlin `noise` covered in that unit. In this scenario, we have a bunch of vehicles chasing a target that is moving around randomly according to Perlin noise. Each vehicle has its own brain, and that brain is a random neural network. This is a regression task.

Flappy Bird

This is based on the very simple yet very addictive Flappy Bird game. If you are unfamiliar with the game, you have to fly through the gap between a series of pipes without hitting them, and all you can do is jump when you press the space bar (or click the mouse). Here, we give lots of flappy birds their own bird brain (random neural network) and reward those who last the longest. Eventually, you have a bird that can fly through the pipes without hitting them. This is a classification task.



What is a genetic algorithm?

A genetic algorithm is a form of reinforcement learning where an object learns through exploration and experience. A genetic algorithm is based on how nature selects the fittest to survive. Although it takes many aspects of natural selection from nature, these are the key elements for a genetic algorithm.

There are five elements to a genetic algorithm:

Population
Fitness
Selection
Crossover
Mutation

Although you don't have to have all the elements, they should play a part in creating a thriving ecosystem. As we are creating a bunch of vehicles that will evolve over time, I will refer to a vehicle or vehicles rather than living things.

Population

We create our initial population of vehicles in the `setup()` function. Each vehicle will have characteristics or a DNA. They may be random to start with but will evolve over time to be better to survive as they pass their better traits on down through their offspring. Creating a population that has enough variety is also key. I know that they are vehicles, but use your imagination and imagine that they can replicate and reproduce!

Fitness

Each vehicle will have a fitness score attached. Depending on its characteristics. This will determine their chances of selection for reproduction.

Selection

This is where the ones which have the highest fitness scores pass their genes onto the next generation.

Crossover

This is where you take the best bits of one parent with the best bits of the other parent and cross them over, combining them into a new offspring. This new offspring is added to the population, thus improving on what was in the initial population.

Mutation

You may be thinking that you can just stop after crossover, but there is one last phase that makes it more likely to create a healthy and thriving ecosystem: mutation. Mutation. This means that there is a small chance of something random happening that might actually be an improvement. It avoids the possibility of the gene pool becoming stagnant and stale. It introduces more variety.

Convergence

You could reach a point where there are no more significant changes to the population, and it is said to have reached convergence, and it is thus as optimal as it is going to get. At this point, you can stop as you have reached the solution.



Example 1: Smart cars

This is our first example. We will give each vehicle (car) a brain, a random neural network (ml5.js) which will have five inputs and two outputs. It will be given a fitness score depending on how close it gets to the target, which will be a wandering circle. The wandering circle will move around the canvas randomly using Perlin noise.

The inputs are:

The **x** component of the vector between the **vehicle** and the **target**

The **y** component of the vector between the **vehicle** and the **target**

The distance between the **target** and the **vehicle**

The **x** component of the velocity of the **vehicle**

The **y** component of the velocity of the **vehicle**

The outputs are:

The **angle** the **vehicle** must turn to move towards the **target**

The **magnitude** of the power to move it to the **target**

This is a regression task because we want values for the outputs, the angle, and the magnitude. Each generation will improve its movement towards the target not because it has been trained to do so but because the neural network (brain) is predisposed to do that. We only select those who do that and ignore those that don't. Eventually, after many generations, they will nearly all follow the circle even though they don't really know why.

For a diagram of the brain inside the vehicles, see fig. 1.



Example 2: Flappy Bird

Flappy Bird works in the same way. Each bird is given a random neural network (brain) using ml5.js. The neural network will have five inputs and two outputs.

! When you begin the unit, you will see that we need only four inputs because the gap is constant between the top pipe and the bottom pipe, so it only needs to know one of them, but you get the point below.

The inputs are:

The **y** position of the bird

The bird's **velocity**

The **position** of the top pipe

The **position** of the bottom pipe

The **x** position of the pipes

The outputs are:

Jump

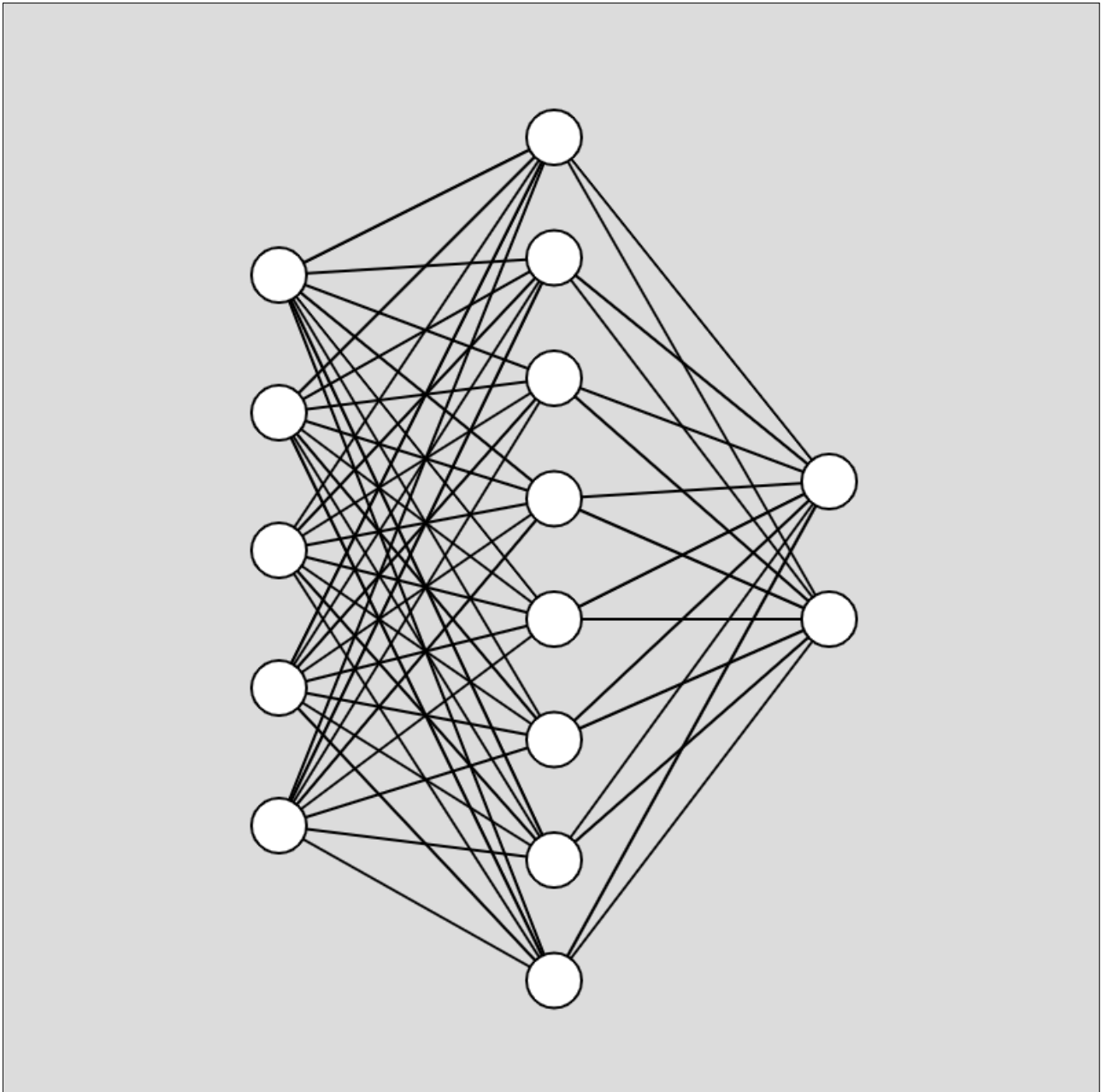
Don't jump

This is a classification task because it is either jump or not jump. The neural network looks like the diagram below (Figure 1). This works through seeing how long the bird lasts before crashing into a pipe, that bird has the best brain. Each bird will get a fitness score depending on how long it lasts. Each generation will get better and better until at least one bird never hits the pipes (hopefully).

This is why AI is so powerful. Humans get tired or lose concentration. The AI doesn't, this doesn't make it perfect but it has that advantage over us, it can be relentless.

First we start with example 1: the smart cars.

Figure 1: the brain!



Algorithmic Intelligence Module D Unit #3 smart cars



Module D Unit # smart cars neural evolution

We are going to give these cars a smart brain (ml5.js). They have to learn to chase after a moving target. All they do is get a reward according to how well they do. This uses a Genetic Algorithm to select those who perform best. We aren't going to train the algorithm but select the ones that perform better, as every car will have its own neural network based on random weights.

Over time, we select the best ones (that get closest to the target) and breed them, thinking of natural selection, evolution. After many populations have come and gone, we are left with ones that have succeeded in breeding the right attributes (weights) to follow a moving target.

We are going to create two classes, target and vehicle. For these, we put them in two new files and add them to the index.html file. This was covered in the **coding snippets part 4**, so you will be familiar with the process of adding files and the purposes of classes. However, I will still emphasise their purpose and function so that you become more familiar with them.

It goes without saying that you will also need to have the line of code for ml5.js in the index.html file. You will also note that we will be switching from one file to another and back again. This means that you might want to keep the side panel open as you code. This way, you can switch quickly from one to the other.

If you don't want the fuss of going from file to file, then you can add the classes in the sketch.js file and just have one long line of code.



The index.html file

Adding ml5.js as per usual. If you still have vehicle.js from the coding snippets then even better.

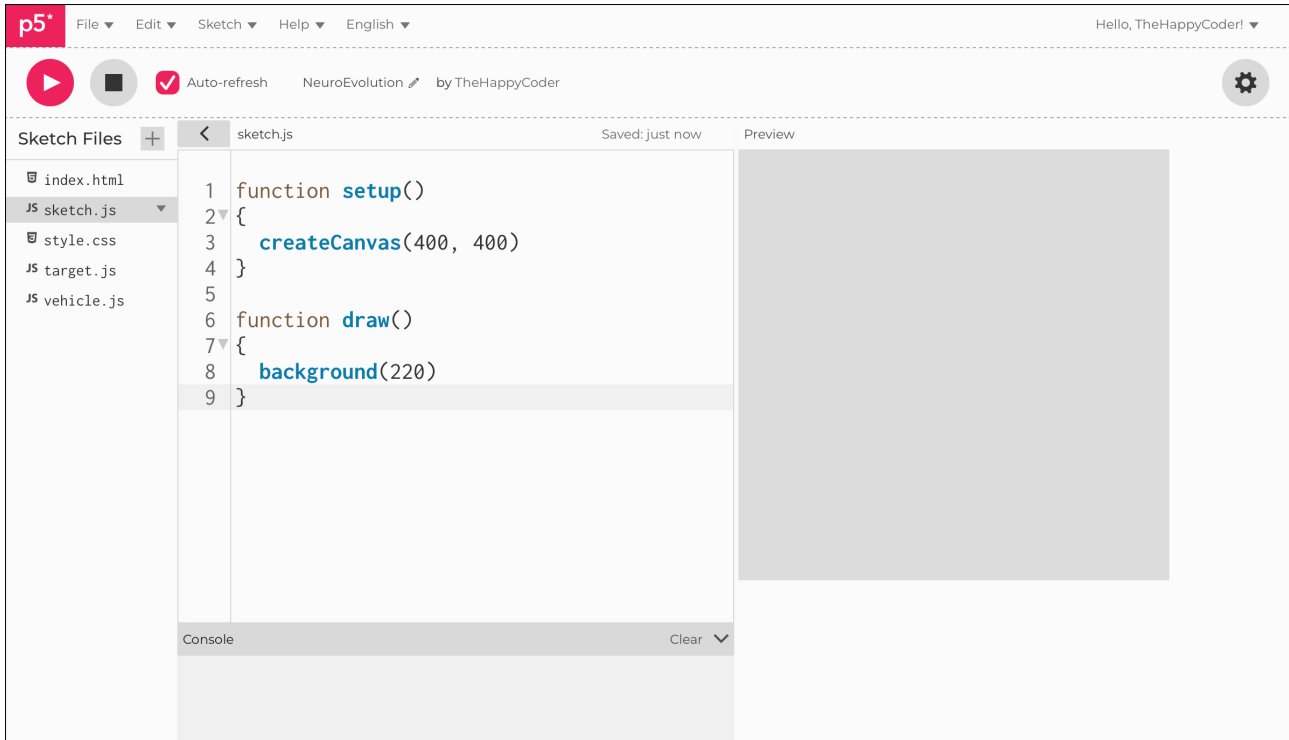
```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Adding target.js and vehicle.js files

Now we are going to add the next two files, **target.js** and **vehicle.js**, to our list of files. See Fig.1.

Figure 1: adding the files

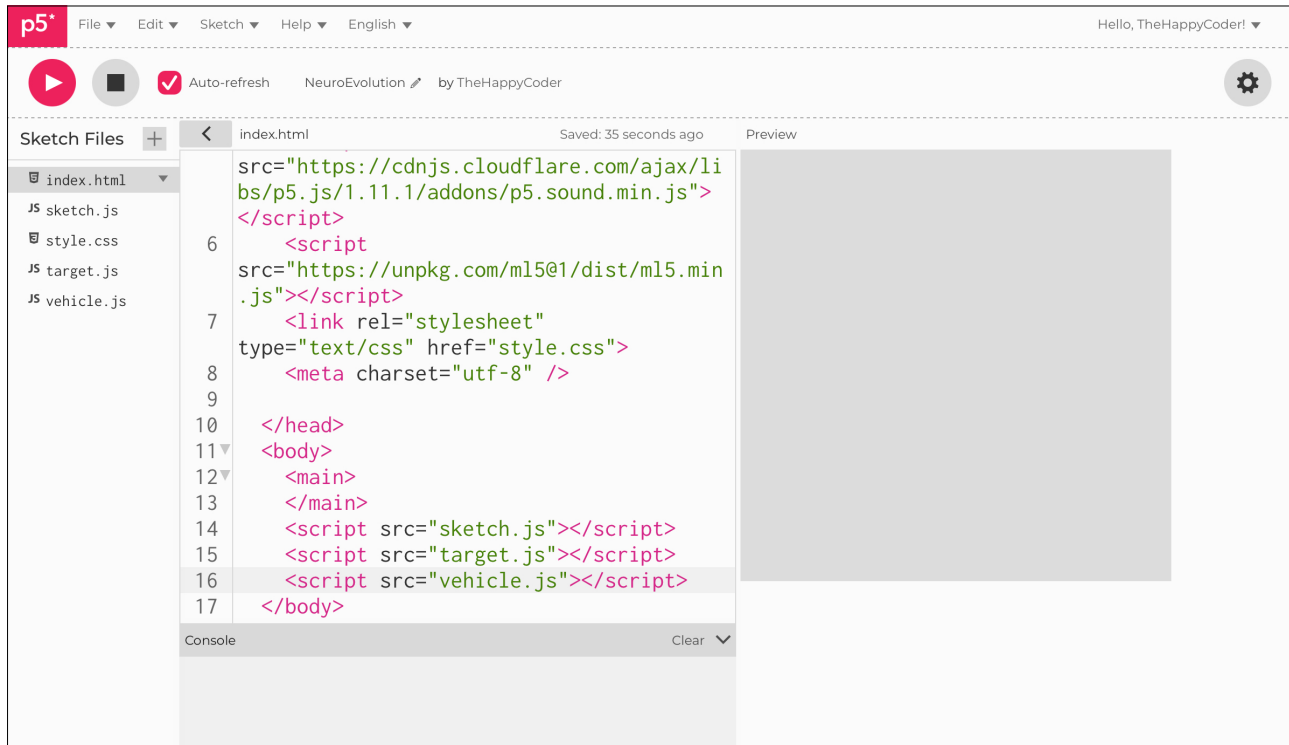




Adding them to the index.html file

This step is easy to forget. Just copy and paste sketch.js and change the names to **target** and **vehicle**. To access the files, just click on them, but these two new ones should be completely empty. See Fig.2.

Figure 2: adding files in index.html





Sketch D3.1 starting sketch

! Starting sketch in `sketch.js`.

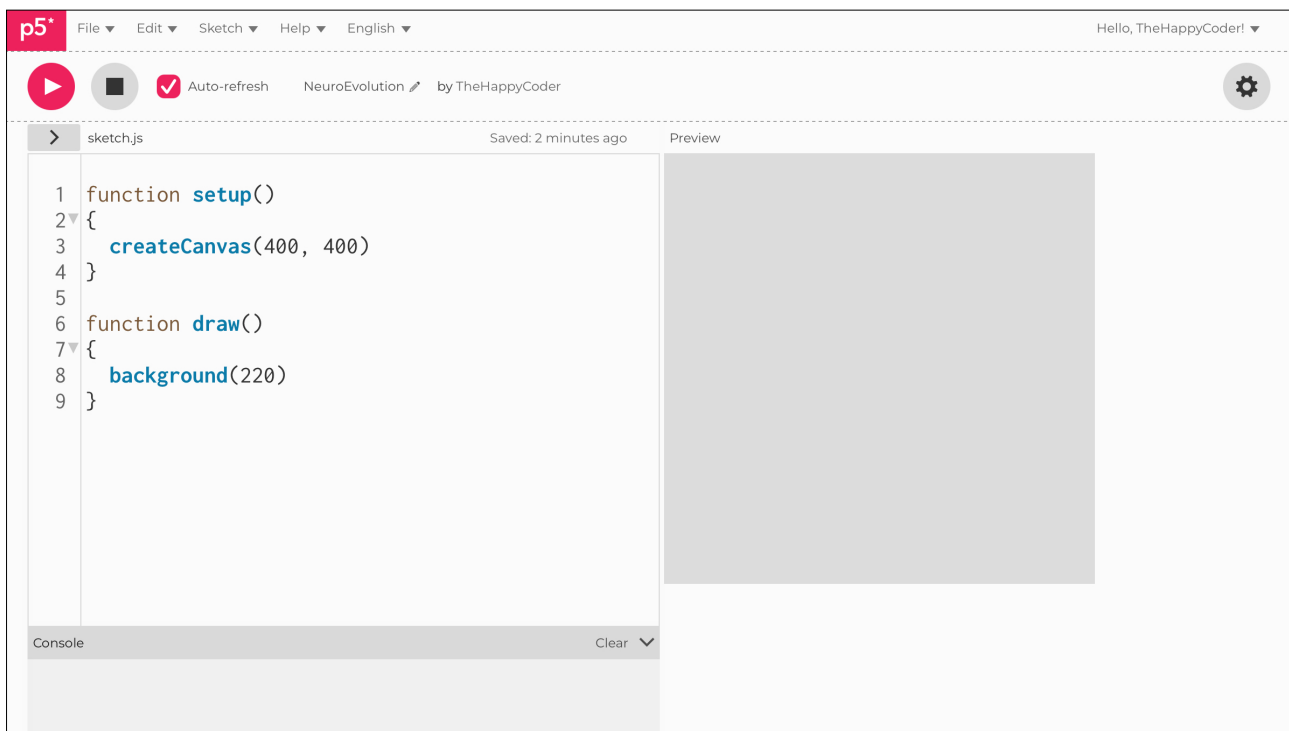
Note that this is for the `sketch.js` file (see the heading).

```
sketch.js

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

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

Figure D3.1





Sketch D3.2 target circle

We are going to create a **target** which will just be a circle. This is what the **vehicles** (cars) will chase around the canvas. The **target** becomes a vector object. From that vector, we pull out the **x** and **y** values, which we will initialise to **width/2** and **height/2**.

sketch.js

```
let target

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

function draw()
{
  background(220)
  target = createVector(width/2, height/2)
  circle(target.x, target.y, 30)
}
```

Notes

You get a static **target** (circle).

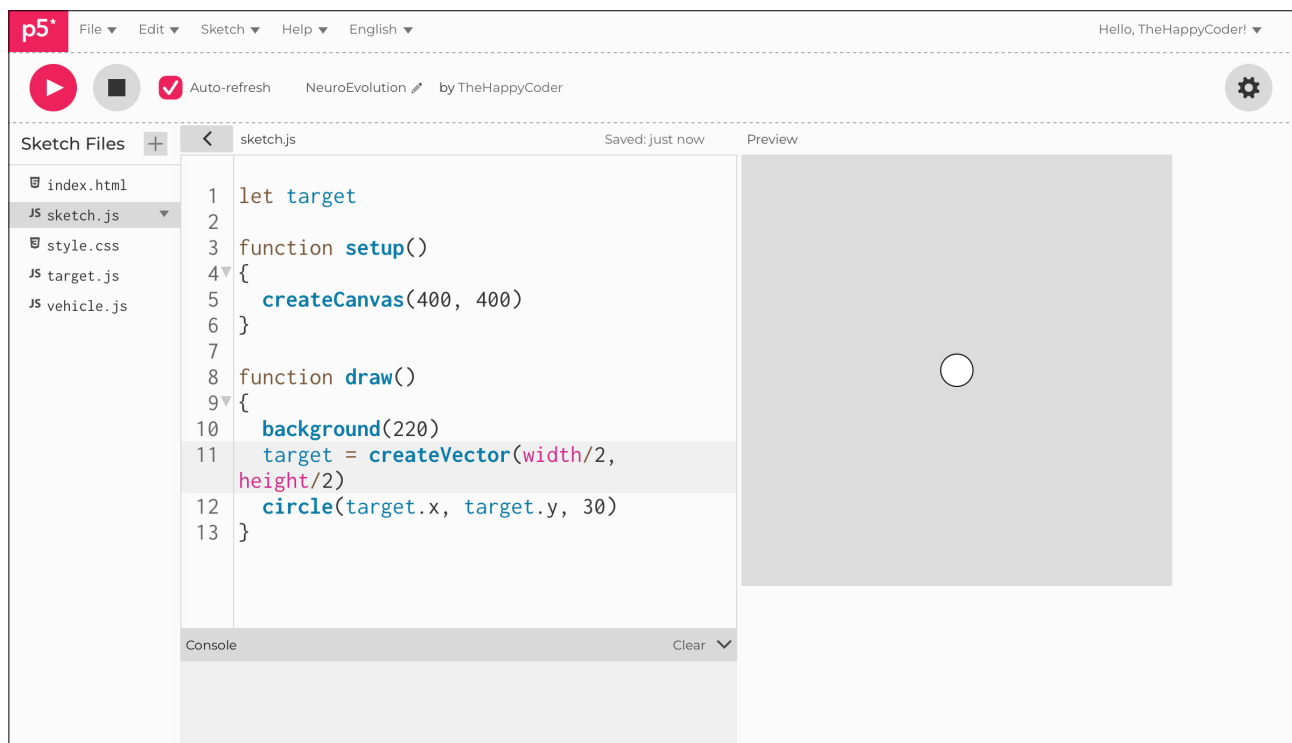
Challenge

Add some colour or even change the shape.

Code Explanation

<code>target = createVector(width/2, height/2)</code>	The target is a vector, its position is in the centre of the canvas
<code>circle(target.x, target.y, 30)</code>	The circle is drawn to the coordinates of the target (target.x and target.y)

Figure D3.2





Sketch D3.3 target class

! Now we move over to the `target.js` file.

We don't want a stationary `target`; we want to give it movement, and in this instance, a random motion called Perlin `noise`. For this, we need to create a class called `Target`.

target.js

```
class Target
{
  constructor()
  {
    this.xoff = 0
    this.yoff = 1000
    this.position = createVector(0, 0)
  }
}
```

Notes

If you haven't completed the unit `coding snippets 4`, then here is a brief explanation of `Perlin Noise`. Think of it as a continuous random number where the next random value is dependent on the previous one. What this means is that there isn't a wild discrepancy from one value to the next. This creates the illusion of a smooth, gentle wandering of the circle.

Imagine a wavy line and the starting point on this wavy line is at 0 (xoff), which will return a value between 0 and +1. If you move along the line to position 1000 (yoff), you get another value between 0 and +1. We then inch along this line and we get a random value just a bit more or a bit less than the previous value. We do this for the `x` offset and the `y` offset; otherwise, we would have the same value for both if they both started at 0 or 1000, for example.

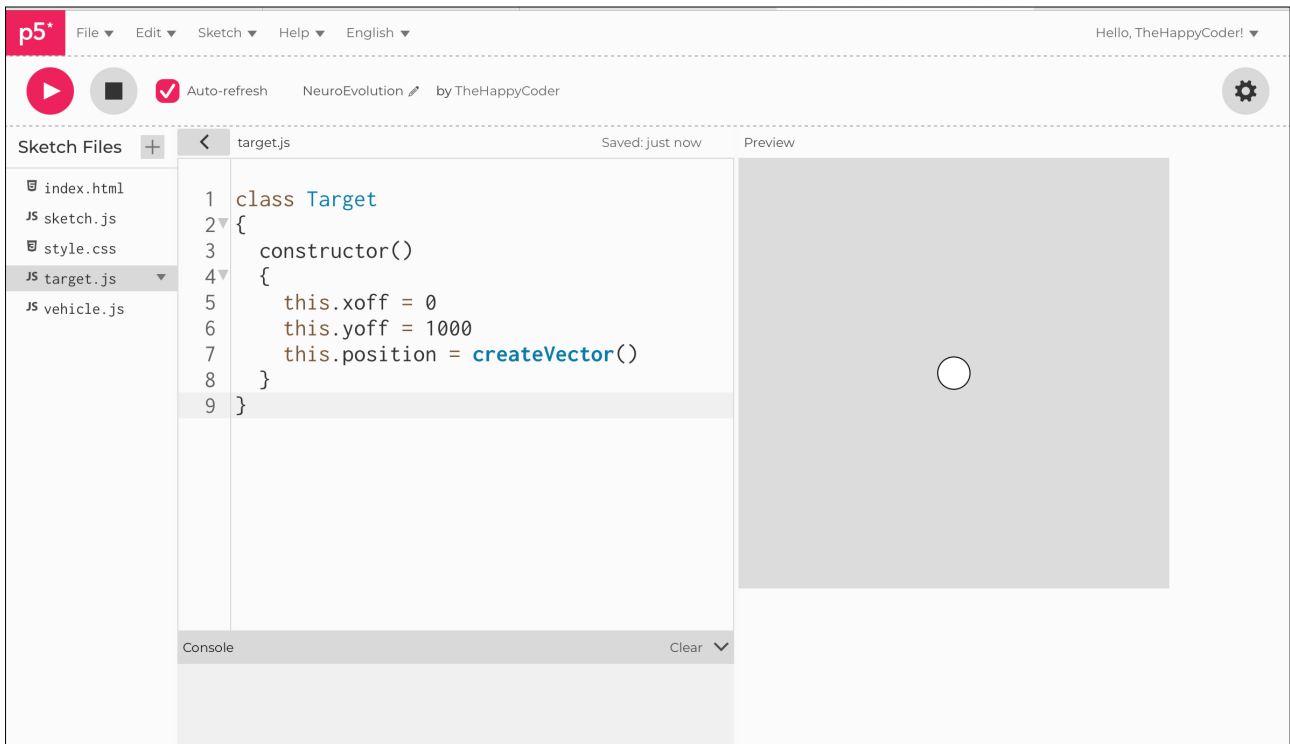
Challenges

1. Have a different starting position on the Perlin `noise` timeline for the `x` and `y` components.
2. What do you think would happen if they were the same?

Code Explanation

<code>this.xoff = 0</code>	The initial x value on the perlin noise time line
<code>this.yoff = 1000</code>	The initial y value on the perlin noise time line
<code>this.position = createVector()</code>	Create a vector for the position of the target

Figure D3.3





Sketch D3.4 perlin noise

We are going to use this randomness to move the **target** (circle) around the canvas. So we need to connect the position of the target to the **noise** (Perlin) value and then increment along that random wavy line by **0.01** for **x** and **y**. We add our two favourite functions, **show()** and **move()**.

target.js

```
class Target
{
  constructor()
  {
    this.xoff = 0
    this.yoff = 1000
    this.position = createVector(0, 0)
  }

  move()
  {
    this.position.x = noise(this.xoff) * width
    this.position.y = noise(this.yoff) * height
    this.xoff += 0.01
    this.yoff += 0.01
  }

  show()
  {
    circle(this.position.x, this.position.y, 30)
  }
}
```

Notes

We multiply by `width` and `height` because the `noise()` function returns values between `0` and `1`, and we want the `target` to move between `0` and the `width` and `height` of the canvas.

Challenge

How would you use the `map()` function?

Code Explanation

<code>this.position.x = noise(this.xoff)*width</code>	We get our target x position from the perlin <code>noise()</code> function (times the width)
<code>this.position.y = noise(this.yoff)*height</code>	We get our target y position from the perlin <code>noise()</code> function (times the height)
<code>this.xoff += 0.01</code>	Increment along the noise time line for x
<code>this.yoff += 0.01</code>	Increment along the noise time line for y



Sketch D3.5 moving target

! Back to `sketch.js`. Comment out and remove the two lines of code for the circle.

We are going to use the `Target` class to create a target object and then run the `update()` and `show()` functions from that object. Now you have a `target` (circle) wandering around the canvas.

sketch.js

```
let target

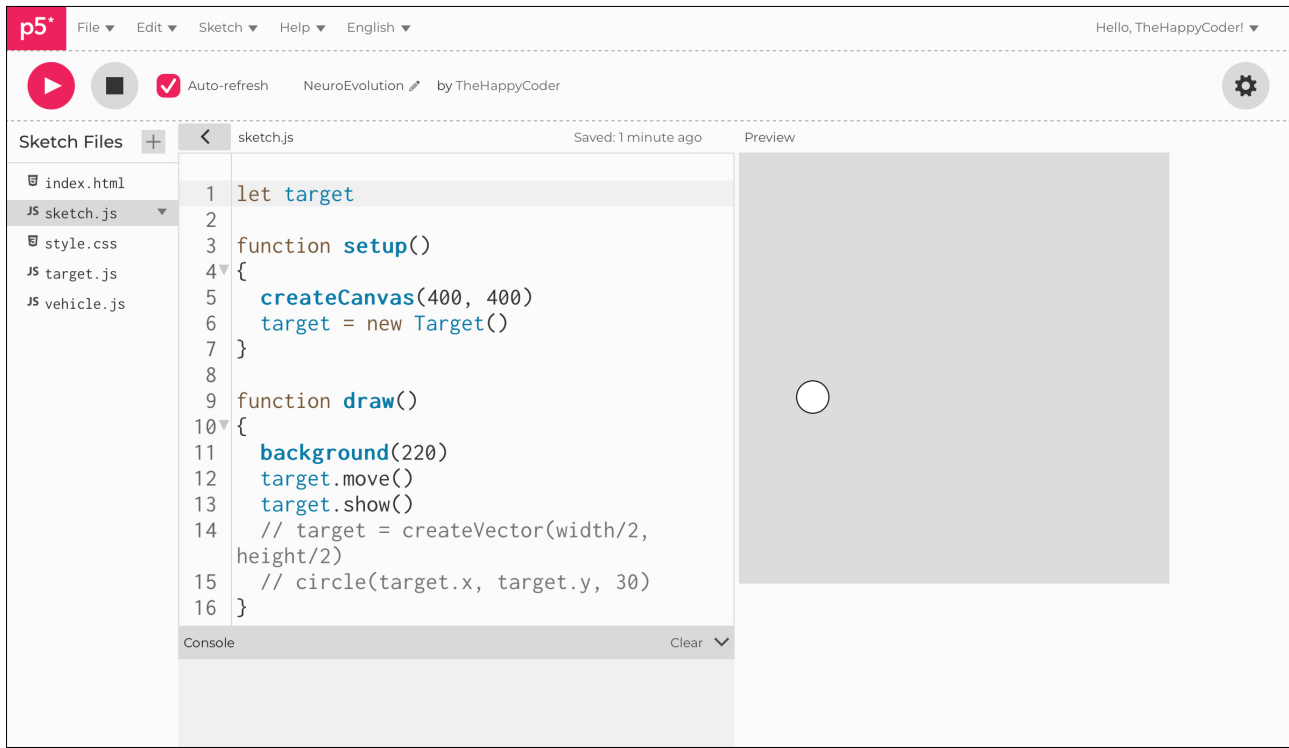
function setup()
{
  createCanvas(400, 400)
  target = new Target()
}

function draw()
{
  background(220)
  target.move()
  target.show()
  // target = createVector(width/2, height/2)
  // circle(target.x, target.y, 30)
}
```

Notes

The **target** should be moving around the canvas in a gentle swooping way.

Figure D3.5





Sketch D3.6 a vehicle

! Hop over to `vehicle.js`.

Now we need a vehicle. We will create a `Vehicle` class, and in its constructor function, we will have `position`, `velocity`, and `acceleration`. This should be fairly familiar to you by now.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)
  }

  show()
  {
    push()
    translate(this.position.x, this.position.y)
    rect(0, 0, 10, 5)
    pop()
  }
}
```

Notes

When we create a `vehicle`, the `constructor()` function will receive two arguments, the `(x, y)` coordinates. Nothing will appear just yet as we haven't actually created a `vehicle`. We previously called them `pos`, `vel`, and `acc`, just nice to be different.



Sketch D3.7 random position

! Back to [sketch.j](#).

Let's create one vehicle for now at some random position.

sketch.js

```
let target
let vehicle

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  vehicle = new Vehicle(random(width), random(height))
}

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

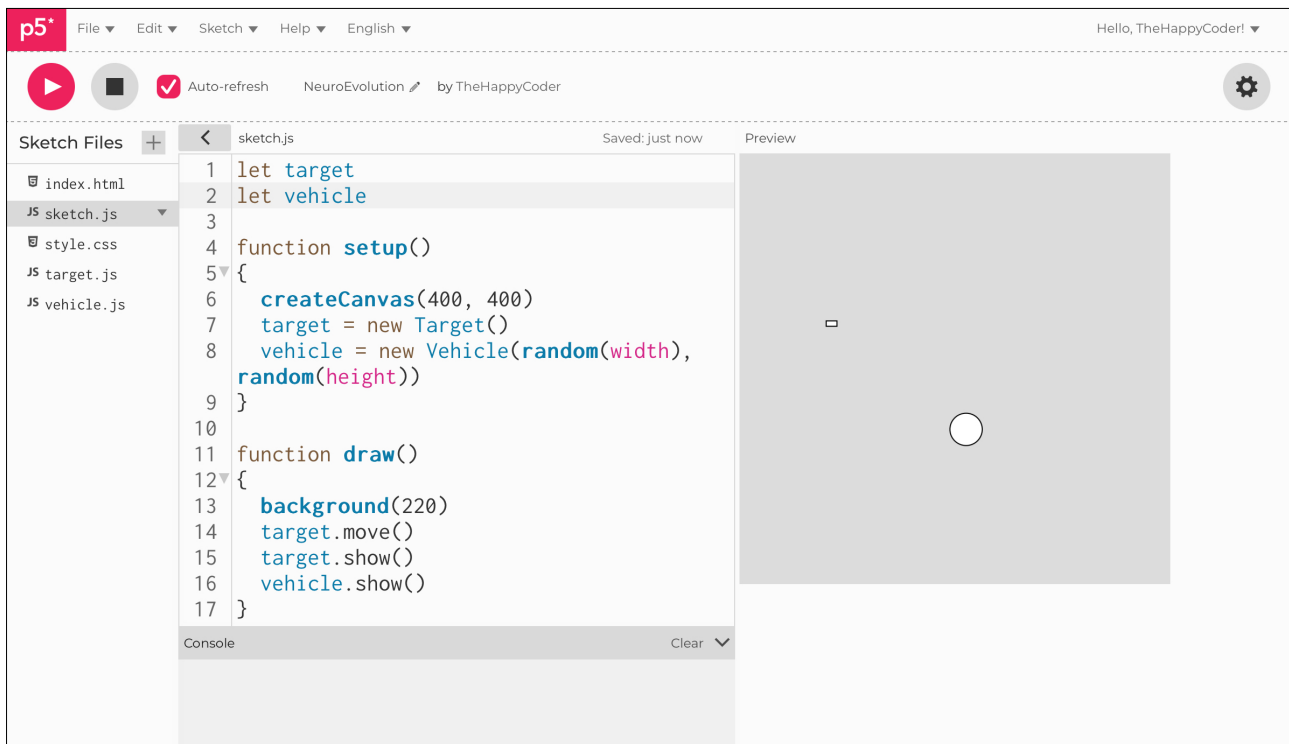
Notes

Every time you run the sketch, you will get another random **vehicle** somewhere on the canvas.

Challenge

Give the vehicle some colour.

Figure D3.7





Sketch D3.8 moving towards the target

! Moving back to **vehicle.j**.

We will now get the **vehicle** moving using the usual formula of adding the components together for movement.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)
    this.maxSpeed = 4
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.velocity.limit(this.maxSpeed)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
  }

  show()
  {
    push()
    translate(this.position.x, this.position.y)
    rect(0, 0, 10, 5)
    pop()
  }
}
```

Notes

We won't see anything happen yet (except the **target** wandering) because we haven't the **move()** function in **sketch.js**.

Code Explanation

```
this.acceleration.mult(0)
```

The acceleration is zeroed so that the vehicle doesn't run away with itself



Sketch D3.9 the seek function

! Because we have covered most of this in the **coding snippets 4** unit with the **seek()** function, I won't labour this part.

We will want to get it to move towards the target and also to rotate. This was covered briefly in the **coding snippets 4** unit using the **heading()** function. Also, we do need to have a maximum force. The vehicle will be steered towards the target by a force of some magnitude; we turn the **vehicle** in that direction. This is where we add the **applyForce()** function and the **seek()** function.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)
    this.maxSpeed = 4
    this.maxForce = 0.2
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.velocity.limit(this.maxSpeed)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
  }

  applyForce(force)
  {
    this.acceleration.add(force)
  }

  seek(target)
  {
    let desired = p5.Vector.sub(target, this.position)
    desired.setMag(this.maxSpeed)
```

```
    let steer = p5.Vector.sub(desired, this.velocity)
    steer.limit(this.maxForce)
    this.applyForce(steer)
  }
```

```
  show()
  {
```

```
    let angle = this.velocity.heading()
```

```
    push()
```

```
    translate(this.position.x, this.position.y)
```

```
    rotate(angle)
```

```
    rect(0, 0, 10, 5)
```

```
    pop()
```

```
  }
```

```
}
```

Notes

In the `show()` function, we find the `angle` from the `heading()` function and then rotate the vehicle according to that `angle`. The `seek()` and `applyForce()` functions are the same as we explored in [coding snippets 4](#).

Code Explanation

<code>let angle = this.velocity.heading()</code>	We get the angle from the direction of the velocity vector
<code>rotate(angle)</code>	We rotate the vehicle by that angle



Sketch D3.10 seek and ye shall find

! Return to `sketch.j`.

We create a vector for the `position` of the `target` and then seek that vector.

sketch.js

```
let target
let vehicle

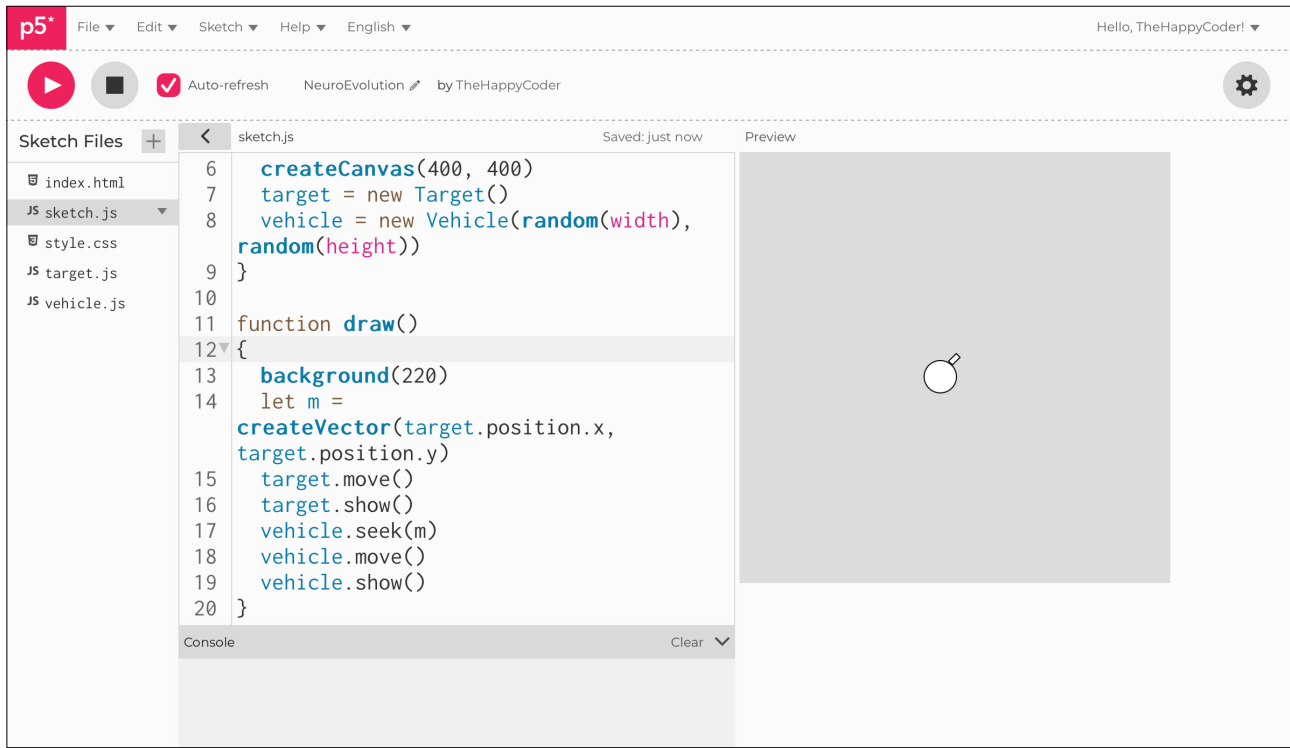
function setup()
{
  createCanvas(400, 400)
  target = new Target()
  vehicle = new Vehicle(random(width), random(height))
}

function draw()
{
  background(220)
  let m = createVector(target.position.x, target.position.y)
  target.move()
  target.show()
  vehicle.seek(m)
  vehicle.move()
  vehicle.show()
}
```

Notes

The **vehicle** now follows the **target**; job done! It is quite mesmerising to watch.

Figure D3.10





Sketch D3.11 more vehicles

We are now going to create **50** of these vehicles. We will replace a number of lines of code to move from the single **vehicle** to an array of **vehicles**.

sketch.js

```
let target

let vehicles = []

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  for (let i = 0; i < 50; i++)
  {
    vehicles[i] = new Vehicle(random(width), random(height))
  }
}

function draw()
{
  background(220)
  let m = createVector(target.position.x, target.position.y)
  target.move()
  target.show()
  for (let vehicle of vehicles)
  {
    vehicle.seek(m)
    vehicle.move()
    vehicle.show()
  }
}
```

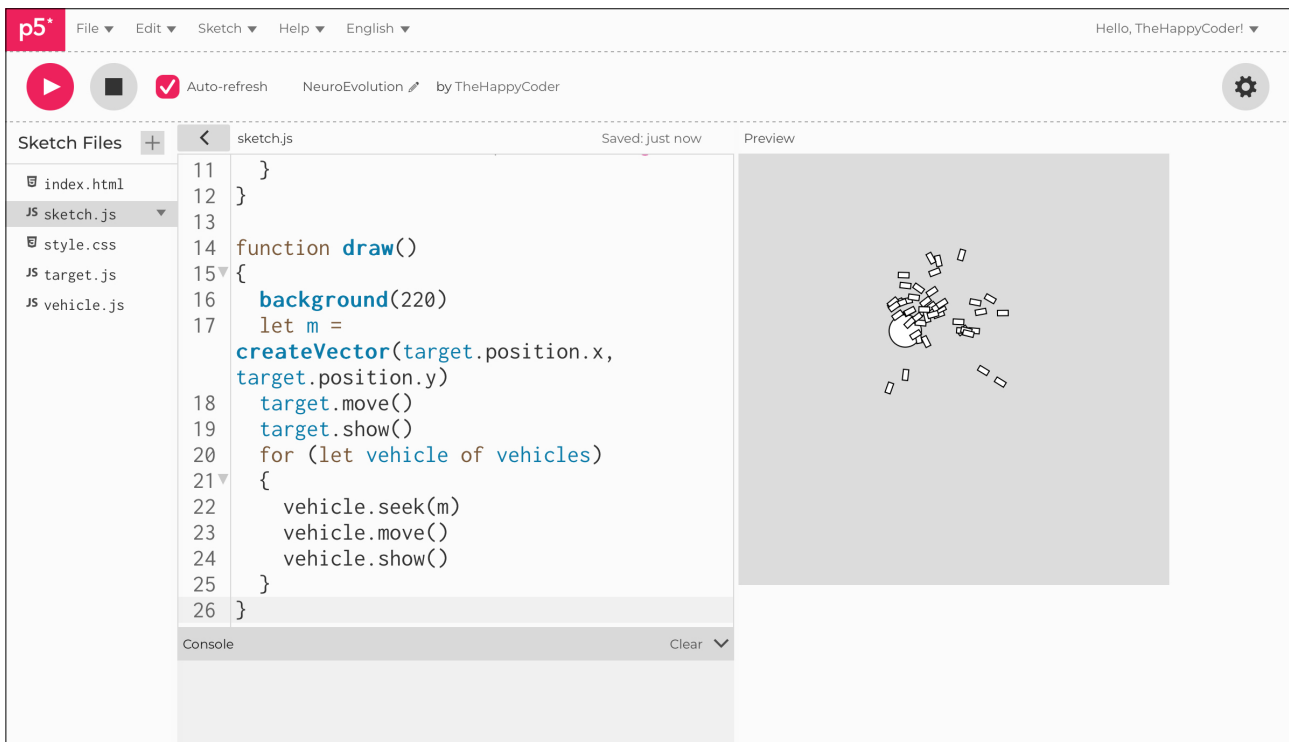
Notes

You should have a whole bunch of **vehicles** following the target, as shown in **Figure D3.11**. Each **vehicle** will start from a different random position but eventually coalesce into one vehicle.

Challenge

Try different numbers of vehicles

Figure D3.11





A neural evolution brain

Although much of the **neural network** stuff we have already covered is used here in this module and unit, there is a subtle difference. We can see it in the lines of code in the next sketch as we add the **neural network** brain to the **vehicles**. First of all, we give the brain the inputs and outputs:

The five inputs are:

The x component of the vector between the vehicle and the target
The y component of the vector between the vehicle and the target
The distance between the target and the vehicle
The x component of the velocity of the vehicle
The y component of the velocity of the vehicle

The outputs are:

The angle the vehicle must turn to move towards the target
The magnitude of the power to move it to the target

Then we are going to give it the task: **regression**. However, here is the big difference: we will inform the neural network that this is a neural evolution network with the **neuroEvolution: true** line of code. Then, finally, we inform the neural network that we are not doing any training with the **noTraining: true** line of code.

Remember that this is a **Reinforcement Learning** challenge; there is no data for the neural network to learn from. We are simply going to select the best brains (neural networks) that appear to solve the challenge without hardcoding the vehicles with the **seek()** function, which really is the simple solution to this challenge.

Make sure you have ml5.js in the index.html file, just checking!

Note

Although I have called this module Neural Evolution, the reference in the code is neuroEvolution. In Dan Shiffman's book, 'The Nature of Code' and his YouTube channel, the Coding Train he refers to it as NeuroEvolution. Both mean the same thing but in the coding it must be neuroEvolution.



Sketch D3.12 giving the vehicle a brain

! Bob over to `vehicle.js`.

In the `Vehicle` class, we give the vehicle a `brain`. The `brain` is the neural network which has five inputs and two outputs. There will be `50 brains` driving their own vehicles.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)
    this.maxSpeed = 4
    this.maxForce = 0.2

    ml5.setBackend("cpu")
    this.brain = ml5.neuralNetwork({
      inputs: 5,
      outputs: 2,
      task: "regression",
      neuroEvolution: true,
      noTraining: true
    })
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.velocity.limit(this.maxSpeed)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
  }

  applyForce(force)
  {
    this.acceleration.add(force)
  }
}
```

```

seek(target)
{
  let desired = p5.Vector.sub(target, this.position)
  desired.setMag(this.maxSpeed)
  let steer = p5.Vector.sub(desired, this.velocity)
  steer.limit(this.maxForce)
  this.applyForce(steer)
}

show()
{
  let angle = this.velocity.heading()
  push()
  translate(this.position.x, this.position.y)
  rotate(angle)
  rect(0, 0, 10, 5)
  pop()
}
}

```

Notes

Notice that this is a regression task; it is also not a normal neural network. This means we can access the `mutate()` and `crossover()` functions later as part of the neural evolution network we will implement later.

Code Explanation

<code>neuroEvolution: true,</code>	This is a neural evolution challenge
<code>noTraining: true</code>	We are not doing any training



Sketch D3.13 steering the vehicles

From the inputs, we need to know how much to steer the **vehicles**. For that, we need to know their **velocity**, their **position** relative to the target, and therefore the **desired** velocity. This is all about magnitude and direction at the end of the day. We do need to do a bit of normalising so that things don't get out of control, hence dividing things by the **width** or the **maxSpeed**.

We are going to rewrite the entire **seek(target)** function in the **Vehicle** class. Nothing will happen (except an error message) because we have done nothing about the outputs, which will determine the force to move the vehicle.

There is a lot to unpack in the **seek()** function. These are our five inputs for the neural network, hence the refactoring.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)
    this.maxSpeed = 4
    this.maxForce = 0.2
    ml5.setBackend("cpu")
    this.brain = ml5.neuralNetwork({
      inputs: 5,
      outputs: 2,
      task: "regression",
      neuroEvolution: true,
      noTraining: true
    })
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.velocity.limit(this.maxSpeed)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
```

```

}

applyForce(force)
{
  this.acceleration.add(force)
}

seek(target)
{
  let v = p5.Vector.sub(target.position, this.position)
  let d = v.mag() / width
  v.normalize()
  let inputs = [
    v.x,
    v.y,
    d,
    this.velocity.x / this.maxSpeed,
    this.velocity.y / this.maxSpeed
  ]
}

show()
{
  let angle = this.velocity.heading()
  push()
  translate(this.position.x, this.position.y)
  rotate(angle)
  rect(0, 0, 10, 5)
  pop()
}
}

```

Notes

You will get an error if you try to run it now, so don't try. The variable **v** is a vector of the difference between any one vehicle's position and the target position. We then get the distance **d** from the magnitude of that difference (**mag()**). We then normalise the vector **v** before creating the five inputs from all that data.

Code Explanation

<code>v = p5.Vector.sub(target.position, this.position)</code>	Subtracting two vectors to get a third vector v
<code>d = v.mag() / width</code>	Distance is the magnitude of the vector v, reducing the magnitude by the width
<code>v.normalize()</code>	Normalising v
<code>v.x</code>	The x component of the vector v
<code>v.y</code>	The y component of the vector v
<code>this.velocity.x / this.maxSpeed</code>	Reducing the x component of the velocity by the maxSpeed
<code>this.velocity.y / this.maxSpeed</code>	Reducing the y component of the velocity by the maxSpeed



Sketch D3.14 the outputs

! Back to `sketch.js`.

Now for the outputs. We have two outputs, the angle and the magnitude. The predicted values are random. They take in the values from the inputs, then generate outputs that are meaningless. Also, you will notice that in the `brain` there is an argument for `noTraining`. This is because we are not using any data to train the neural network; we are just choosing to select the best performing ones. So when you run this, they just disappear off into the distance.

! We will remove:

```
let m = createVector(target.position.x, target.position.y)
```

and replace with:

```
vehicle.seek(target)
```

sketch.js

```
let target
let vehicles = []

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  for (let i = 0; i < 50; i++)
  {
    vehicles[i] = new Vehicle(random(width), random(height))
  }
}

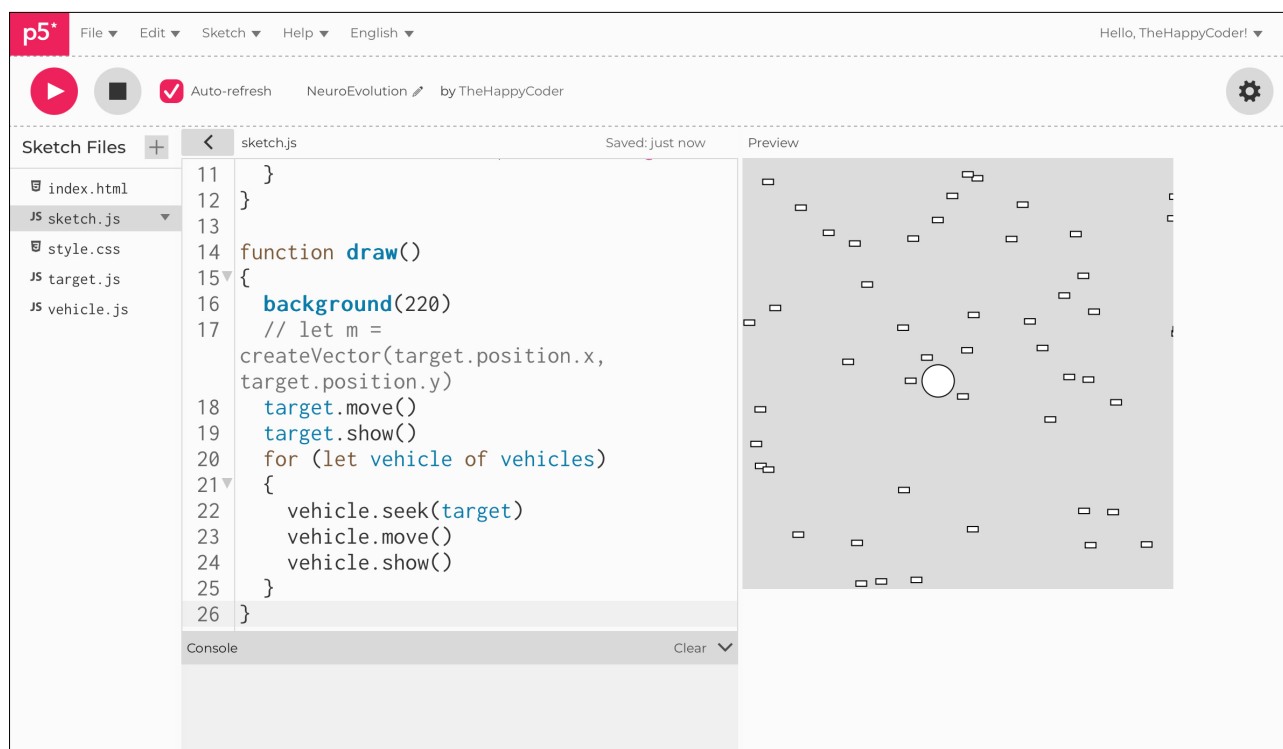
function draw()
{
  background(220)
  // let m = createVector(target.position.x, target.position.y)
  target.move()
  target.show()
  for (let vehicle of vehicles)
  {
    vehicle.seek(target)
```

```
    vehicle.move()
    vehicle.show()
  }
}
```

Notes

You will get static vehicles.

Figure D3.14





Sketch D3.15 they move

! Skip over to **vehicle.js**.

Now let's get them moving.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)
    this.maxSpeed = 4
    this.maxForce = 0.2
    ml5.setBackend("cpu")
    this.brain = ml5.neuralNetwork({
      inputs: 5,
      outputs: 2,
      task: "regression",
      neuroEvolution: true,
      noTraining: true
    })
  }

  move(target)
  {
    this.velocity.add(this.acceleration)
    this.velocity.limit(this.maxSpeed)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
  }

  applyForce(force)
  {
    this.acceleration.add(force)
  }
}
```

```

seek(target)
{
  let v = p5.Vector.sub(target.position, this.position)
  let d = v.mag() / width
  v.normalize()
  let inputs = [
    v.x,
    v.y,
    d,
    this.velocity.x / this.maxSpeed,
    this.velocity.y / this.maxSpeed
  ]

  let outputs = this.brain.predictSync(inputs)
  let angle = outputs[0].value * TWO_PI
  let magnitude = outputs[1].value
  let force = p5.Vector.fromAngle(angle)
  force.setMag(magnitude)
  this.applyForce(force)
}

show()
{
  let angle = this.velocity.heading()
  push()
  translate(this.position.x, this.position.y)
  rotate(angle)
  rect(0, 0, 10, 5)
  pop()
}
}

```

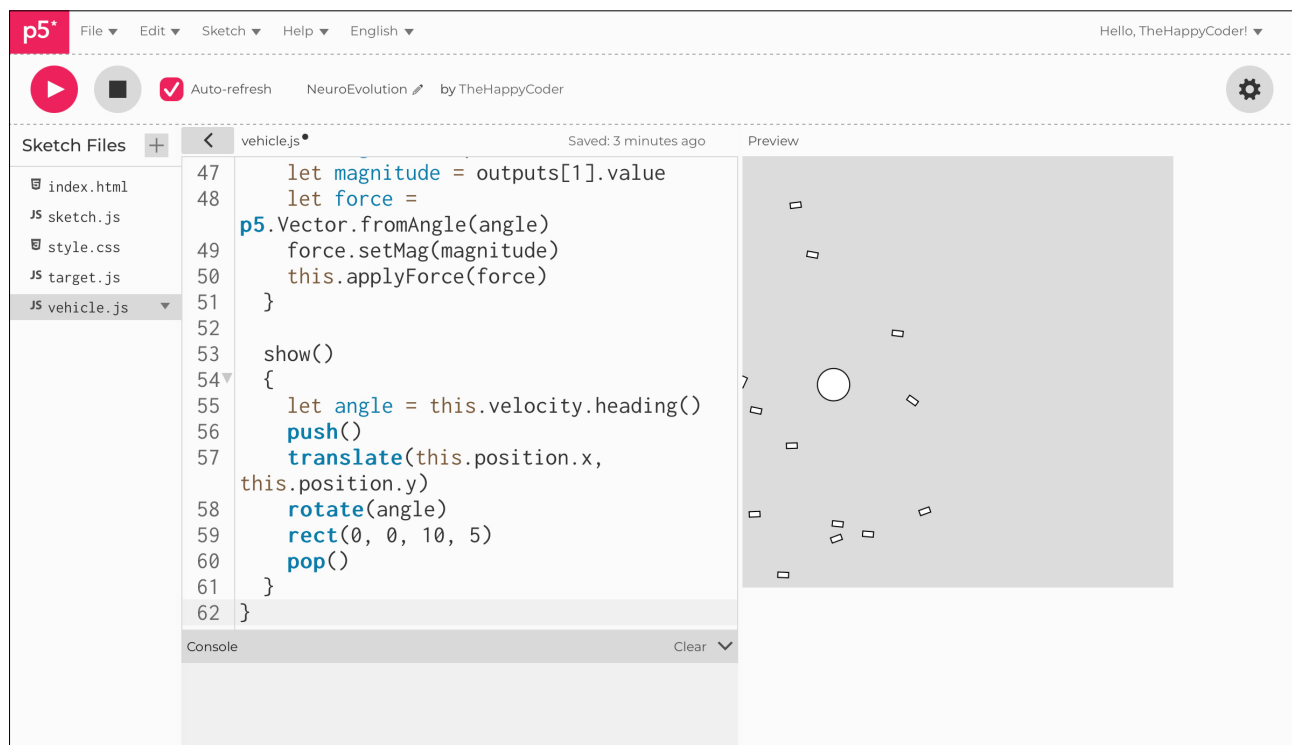
Notes

They move, but they just wander off into the sunset! This is because they have no reason to do otherwise.

Code Explanation

<code>outputs = this.brain.predictSync(inputs)</code>	<code>predictSync()</code> means it waits for the inputs before predicting
<code>angle = outputs[0].value * TWO_PI</code>	The outputs index [0] is the angle which we multiply by 2π (to scale the output value)
<code>magnitude = outputs[1].value</code>	The outputs index [1] is the magnitude
<code>force = p5.Vector.fromAngle(angle)</code>	The function <code>fromAngle()</code> creates a vector from an angle in radians

Figure D3.15





Sketch D3.16 fitness scores

For this to evolve, we need to attribute a fitness score to each vehicle. We need to know a number of things; one of the main indicators is how close the vehicle gets to the target, so we move an imaginary circle, which will help us know when the target and the vehicle overlap. We call the vehicle radius `this.l`, and for the target, we will call it `this.r`.

vehicle.js

```
class Vehicle
{
  constructor(x, y)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)

    this.l = 5
    this.fitness = 0

    this.maxSpeed = 4
    this.maxForce = 0.2
    ml5.setBackend("cpu")
    this.brain = ml5.neuralNetwork({
      inputs: 5,
      outputs: 2,
      task: "regression",
      neuroEvolution: true,
      noTraining: true
    })
  }

  move()
  {
    this.velocity.add(this.acceleration)
    this.velocity.limit(this.maxSpeed)
    this.position.add(this.velocity)
    this.acceleration.mult(0)

    let d2 = p5.Vector.dist(this.position, target.position)
    if (d2 < this.l + target.r)
    {
      this.fitness++
    }
  }
}
```

```

    }
  }

  applyForce(force)
  {
    this.acceleration.add(force)
  }

  seek(target)
  {
    let v = p5.Vector.sub(target.position, this.position)
    let d = v.mag() / width
    v.normalize()
    let inputs = [
      v.x,
      v.y,
      d,
      this.velocity.x / this.maxSpeed,
      this.velocity.y / this.maxSpeed
    ]
    let outputs = this.brain.predictSync(inputs)
    let angle = outputs[0].value * TWO_PI
    let magnitude = outputs[1].value
    let force = p5.Vector.fromAngle(angle)
    force.setMag(magnitude)
    this.applyForce(force)
  }

  show()
  {
    let angle = this.velocity.heading()
    push()
    translate(this.position.x, this.position.y)
    rotate(angle)
    rect(0, 0, this.l*2, this.l)
    pop()
  }
}

```

Notes

We haven't actually defined **r** yet (see next sketch). We call this d2, so as not to confuse with the d variable already used. We cannot use dist or distance, these words are already taken in p5.js for something else.

Code Explanation

<code>d2 = p5.Vector.dist(this.position, target.position)</code>	We calculate the distance between the position of each vehicle and the target
<code>if (d2 < this.l + target.r)</code>	If the distance is less than the l value (5) and the r value (15 in next sketch) combined then...
<code>this.fitness++</code>	Increase the fitness score of that particular vehicle



Sketch D3.17 we need a radius

! Mosey on over to `target.js`.

We need a variable for the radius `r` of the circle (`target`) to measure the `fitness`.

target.js

```
class Target
{
  constructor()
  {
    this.xoff = 0
    this.yoff = 1000
    this.position = createVector(0, 0)
    this.r = 15
  }

  move()
  {
    this.position.x = noise(this.xoff) * width
    this.position.y = noise(this.yoff) * height
    this.xoff += 0.01
    this.yoff += 0.01
  }

  show()
  {
    circle(this.position.x, this.position.y, this.r * 2)
  }
}
```

Notes

We now have a variable so we can compare the distance between the vehicle and the target.



Sketch D3.18 overlap

! Back we go to `sketch.js`.

So we have added a fitness to each vehicle, and those that pass within a distance of `r + l` (they overlap) their fitness will increase. This is all prep for later on as we add in more features. Yet we do need to normalise the fitness of the vehicle; otherwise, you will get huge differences. We do this very simply by adding up all the fitnesses (`sum`) and dividing an individual vehicle's fitness by that total value (`sum`) of all the fitnesses together. This means that they all add up to `1`, which is important later on.

sketch.js

```
let target
let vehicles = []

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  for (let i = 0; i < 50; i++)
  {
    vehicles[i] = new Vehicle(random(width), random(height))
  }
}

function draw()
{
  background(220)
  target.move()
  target.show()
  for (let vehicle of vehicles)
  {
    vehicle.seek(target)
    vehicle.move()
    vehicle.show()
  }
}

function normaliseFitness()
{

```



```
let sum = 0
for (let vehicle of vehicles)
{
    sum += vehicle.fitness
}
for (let vehicle of vehicles)
{
    vehicle.fitness = vehicle.fitness / sum
}
}
```

Notes

The **vehicle of vehicles** goes through the array of **vehicles**, one by one. We have created this new function, but nothing will happen until we call it.

Challenge

You could change the name to (thing of vehicles), and so what else would you have to change?

Code Explanation

<code>sum = 0</code>	The total (sum) starts with zero
<code>sum += vehicle.fitness</code>	Go through all the vehicles adding their fitness scores as we go along
<code>vehicle.fitness = vehicle.fitness / sum</code>	Change the fitness score of each vehicle by dividing by the total (sum)



Weighted selection

The next step is to select the highest-scoring vehicles, the ones with the best fitness. However, this isn't very natural, even if it seems very logical. This is an approach you might consider whereby you walk through the population array, picking out the highest pairs of values and mating them, and removing them from the array before looking for the next highest pair.

Instead, we will use what is called **weighted selection**, which still gives every vehicle a chance to be selected but will favour those who have the highest fitness. The way that **Dan Shiffman** explained it in his brilliant book **Nature of Code** is summarised thus:

Imagine that you are in a relay race; the race starts, and you hand over the baton to the next runner, and so on, until your team reaches the end of the race. But instead of each runner running the same distance on each leg of the relay, the fittest runs a longer leg based on their fitness, and the least fit runs a shorter leg, and this is where the analogy breaks down a bit; the winner is the runner that crosses the line first.

This means that it is possible for any runner (depending on the distance of the race) to potentially win, although the fittest runner has a higher probability of winning because it is more likely that they will be there at the end when they cross the finishing line.

If you read the code below, you can see how that works, even if it isn't very intuitive. I will include a link to the website that has the book online for you to read.



Sketch D3.19 the relay race

We want to reproduce with the better vehicles based on their fitness. But first, we need to decide which ones are the best and select their brains. For this, we use a technique called **weighted selection**. This is a more realistic way of natural selection. The best ones will still have the best chance, so it isn't purely random, but it does give other lesser vehicles to reproduce as well.

It uses a probability, **random(1)**, which is the length of the race between **0** and **1**. It then goes through the array, reducing that distance (called **start**) until it reaches the end (**0**). Whichever vehicle crosses the line at the end is the winner. The clock stops (the array stops, hence **index--**), then continues again with the array from where we left off, with a new random number (race distance) and sees who the winner is on this occasion until we have gone through the whole array of vehicles (**population**).

Remember that the **fitness score** of all the vehicles adds up to **1**.

sketch.js

```
let target
let vehicles = []

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  for (let i = 0; i < 50; i++)
  {
    vehicles[i] = new Vehicle(random(width), random(height))
  }
}

function draw()
{
  background(220)
  target.move()
  target.show()
  for (let vehicle of vehicles)
  {
    vehicle.seek(target)
    vehicle.move()
    vehicle.show()
  }
}
```

```
}

function normaliseFitness()
{
    let sum = 0
    for (let vehicle of vehicles)
    {
        sum += vehicle.fitness
    }
    for (let vehicle of vehicles)
    {
        vehicle.fitness = vehicle.fitness / sum
    }
}
```

```
function weightedSelection()
{
    let index = 0
    let start = random(1)
    while (start > 0)
    {
        start = start - vehicles[index].fitness
        index++
    }
    index--
    return vehicles[index].brain
}
```

Notes

This is a bit challenging to get your head around at first. This approach is memory-efficient but computationally more challenging; this is only a problem if there is a very large population where another approach might be better.

Challenge

Can you think of a better way?

Code Explanation

<code>index = 0</code>	We begin with the first in the array of population, this is our first runner
<code>start = random(1)</code>	We create a random number between 0 and 1, this is the length of the race
<code>while (start > 0)</code>	We keep going until we reach the end of the race
<code>start = start - vehicles[index].fitness</code>	We remove the fitness score for that vehicle (runner) reducing the start random value (race distance)
<code>index++</code>	Go to the next vehicle (runner)
<code>index--</code>	When finished stop the race
<code>return vehicles[index].brain</code>	Get the detail of the best vehicle (runner)



Reproduction

We want a **lifespan** for the current vehicles, say **250** frames, so that just before they die off, they reproduce a better version of themselves. They don't all mate; only those who have a reasonably high fitness.

From the **weighted selection**, we chose two parents and created a child through crossover, where the child will carry some of the genes from each parent. This child will have a mutation rate. This is important; otherwise, the gene pool becomes stagnant. There needs to be some element of randomness. This is often seen in nature. If those mutations are useful, they can be incorporated; otherwise, they die off.

The ml5.js machine learning library has a built-in function for **crossover()** and **mutation()** which is a relatively new addition but ever so helpful if you are developing simulations or games.

The **crossover()** function simply takes some of the weights of one successful parent's neural network with the weights of another successful parent's neural network and, in some way, combines them to produce a new neural network with these combined weights. There are various ways this can be done, but that is for another discussion. For now, let us just make use of it.



Sketch D3.20 the next generation

To create the next generation, we need to create a new function and we will call `reproduction()`. Here, we will create the next generation of children, which will be the same length as our first generation. This is a holding array called `nextVehicles`, and once we have filled it, we will rename it `vehicles`.

sketch.js

```
let target
let vehicles = []

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  for (let i = 0; i < 50; i++)
  {
    vehicles[i] = new Vehicle(random(width), random(height))
  }
}

function draw()
{
  background(220)
  target.move()
  target.show()
  for (let vehicle of vehicles)
  {
    vehicle.seek(target)
    vehicle.move()
    vehicle.show()
  }
}

function normaliseFitness()
{
  let sum = 0
  for (let vehicle of vehicles)
  {
```

```

    sum += vehicle.fitness
  }
  for (let vehicle of vehicles)
  {
    vehicle.fitness = vehicle.fitness / sum
  }
}

```

```

function weightedSelection()
{
  let index = 0
  let start = random(1)
  while (start > 0)
  {
    start = start - vehicles[index].fitness
    index++
  }
  index--
  return vehicles[index].brain
}

```

```

function reproduction()
{
  let nextVehicles = []
  for (let i = 0; i < vehicles.length; i++)
  {
    let parentA = weightedSelection()
    let parentB = weightedSelection()
    let child = parentA.crossover(parentB)
    child.mutate(0.01)
    nextVehicles[i] = new Vehicle(random(width), random(height), child)
  }
  vehicles = nextVehicles
}

```


Notes

Notice that we have an extra argument for creating the next generation of vehicle, the **child**. We will need to add it into the **constructor()** function in the **Vehicle** class.

Code Explanation

<code>nextVehicles = []</code>	We create an empty array for the new vehicles.
<code>parentA = weightedSelection()</code>	Select the first parent, call the <code>weightedSelection()</code> function
<code>parentB = weightedSelection()</code>	Select the second parent, call the <code>weightedSelection()</code> function
<code>child.mutate(0.01)</code>	When the child is created apply a tiny mutation rate of 0.01
<code>nextVehicles[i] = new Vehicle(random(width), random(height), child)</code>	Create a new vehicle from the child
<code>vehicles = nextVehicles</code>	Rename all the new vehicles: <code>vehicles</code>



Sketch D3.21 birth of a child

! Go over to `vehicle.js`.

You may have noticed that when we create a new `Vehicle()`, we have included a third argument called `child`. This is effectively the `brain`, but when we originally created a vehicle, we only used two arguments, the `x` and the `y` position. So we need to factor that in when creating our second generation and onwards. We need to check if there is a `brain` in the first place, and if we have, then we update it with the new one.

We get around this by checking to see if there is already a brain and replacing it. If there isn't a brain, we give it one. We alter the `constructor()` function of the `Vehicle` class as appropriate.

! We create an `if...else()` statement to check whether the vehicle already has a brain and if not, then create one. We have moved the neural network (brain) inside the `else()` statement, hence it is highlighted.

```
vehicle.js

class Vehicle
{
  constructor(x, y, brain)
  {
    this.position = createVector(x, y)
    this.velocity = createVector(0, 0)
    this.acceleration = createVector(0, 0)
    this.l = 5
    this.fitness = 0
    this.maxSpeed = 4
    this.maxForce = 0.2

    if (brain)
    {
      this.brain = brain
    }
    else
    {
      ml5.setBackend("cpu")
      this.brain = ml5.neuralNetwork({
        inputs: 5,
        outputs: 2,
        task: "regression",
```

```

        neuroEvolution: true,
        noTraining: true
    })
}

}

move()
{
    this.velocity.add(this.acceleration)
    this.velocity.limit(this.maxSpeed)
    this.position.add(this.velocity)
    this.acceleration.mult(0)
    let d2 = p5.Vector.dist(this.position, target.position)
    if (d2 < this.l + target.r)
    {
        this.fitness++
    }
}

applyForce(force)
{
    this.acceleration.add(force)
}

seek(target)
{
    let v = p5.Vector.sub(target.position, this.position)
    let d = v.mag() / width
    v.normalize()
    let inputs = [
        v.x,
        v.y,
        d,
        this.velocity.x / this.maxSpeed,
        this.velocity.y / this.maxSpeed
    ]
    let outputs = this.brain.predictSync(inputs)
    let angle = outputs[0].value * TWO_PI

```

```

    let magnitude = outputs[1].value
    let force = p5.Vector.fromAngle(angle)
    force.setMag(magnitude)
    this.applyForce(force)
  }

  show()
  {
    let angle = this.velocity.heading()
    push()
    translate(this.position.x, this.position.y)
    rotate(angle)
    rect(0, 0, this.l*2, this.l)
    pop()
  }
}

```

Notes

If you run it, you shouldn't get any errors, but neither will you see the next **generation** just yet. We need to have a **lifespan** so that the previous **generation** dies out and the next is birthed.

Code Explanation

constructor(x, y, brain)	Added the brain (child) component argument to the constructor function
if (brain)	Check to see if the vehicle already has a brain
this.brain = brain	If it does then replace it with the new brain (child)



Sketch D3.22 lifeCounter

! Return to [sketch.js](#).

All we have got left to do is give the vehicle a **lifespan** and then **reproduce** it. We need to count the number of **iterations** rather than actual frames (which is another way of doing it); this way, we could have a countdown. We are also going to keep track of how many generations there are and display this later on.

sketch.js

```
let target
let vehicles = []
let lifeSpan = 250
let lifeCounter = 0
let generations = 0

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  for (let i = 0; i < 50; i++)
  {
    vehicles[i] = new Vehicle(random(width), random(height))
  }
}

function draw()
{
  background(220)
  target.move()
  target.show()
  for (let vehicle of vehicles)
  {
    vehicle.seek(target)
    vehicle.move()
    vehicle.show()
  }
  lifeCounter++
  if (lifeCounter > lifeSpan)
  {
```

```

    normaliseFitness()
    reproduction()
    lifeCounter = 0
    generations++
  }
}

function normaliseFitness()
{
  let sum = 0
  for (let vehicle of vehicles)
  {
    sum += vehicle.fitness
  }
  for (let vehicle of vehicles)
  {
    vehicle.fitness = vehicle.fitness / sum
  }
}

function weightedSelection()
{
  let index = 0
  let start = random(1)
  while (start > 0)
  {
    start = start - vehicles[index].fitness
    index++
  }
  index--
  return vehicles[index].brain
}

function reproduction()
{
  let nextVehicles = []
  for (let i = 0; i < vehicles.length; i++)
  {

```

```

    let parentA = weightedSelection()
    let parentB = weightedSelection()
    let child = parentA.crossover(parentB)
    child.mutate(0.01)
    nextVehicles[i] = new Vehicle(random(width), random(height), child)
  }
  vehicles = nextVehicles
}

```

Notes

You should now see the **generations** appearing. They should also start to **follow** the target after a little while (you have to be patient). If you are impatient, then look at the next sketch.

Challenges

1. Add colour or images (instead of rectangles) to the scene.
2. How would you go about creating a 3D version (not easy but not impossible)?

Code Explanation

lifeSpan = 250	The life of a vehicle
lifeCounter = 0	Keeps track of how many iterations there have been
generations = 0	Counts the number of generations
lifeCounter++	Counts the number of iterations, adding 1 each time
if (lifeCounter > lifeSpan)	Check is to see if the vehicles have reached the end of their lifespan
normaliseFitness()	The fitness is normalised at the end of each generation
reproduction()	The reproduction happens at the end of the generation
lifeCounter = 0	Reset the counter
generations++	Add another generation to the tally



What next?

In a sense, you have done it. If you watch long enough, you will see the changes take place. The **vehicles** will start to follow the **target** as they evolve through the improved fitness levels of those who are successful. This, however, is a slow process, and there is a way of speeding it up and also having a bit more information on the number of **generations** you are at.

We can use a **slider** to speed up the process. Remember that the computer can do this much faster than we are watching it. This is so we can see it happening in real time, as it were.

You can be as creative as you want. I created some bees following a honey pot.



Sketch D3.23 adding a slider

Adding a **slider**, what we are doing is increasing the rate of iterations as another **for()** loop within **draw()**. We can also see the number of **generations**; this helps get an idea of how many **generations** have passed. The create slider function has three arguments: the minimum value (**1**), the maximum value (**20**), and the increment value (**1**).

sketch.js

```
let target
let vehicles = []
let lifeSpan = 250
let lifeCounter = 0
let generations = 0
let timeSlider

function setup()
{
  createCanvas(400, 400)
  target = new Target()
  for (let i = 0; i < 50; i++)
  {
    vehicles[i] = new Vehicle(random(width), random(height))
  }
  timeSlider = createSlider(1, 20, 1)
  timeSlider.position(10, 420)
}

function draw()
{
  background(220)
  target.show()
  for (let vehicle of vehicles)
  {
    vehicle.show()
  }
  for (let i = 0; i < timeSlider.value(); i++)
  {
    for (let vehicle of vehicles)
    {
```

```
    vehicle.seek(target)
    vehicle.move()
  }
  target.move()
  lifeCounter++
}
```

```
if (lifeCounter > lifeSpan)
{
  normaliseFitness()
  reproduction()
  lifeCounter = 0
  generations++
}
```

```
  textSize(40)
  text(generations, 10, 380)
```

```
}
```

```
function normaliseFitness()
{
  let sum = 0
  for (let vehicle of vehicles)
  {
    sum += vehicle.fitness
  }
  for (let vehicle of vehicles)
  {
    vehicle.fitness = vehicle.fitness / sum
  }
}
```

```
function weightedSelection()
{
  let index = 0
  let start = random(1)
  while (start > 0)
  {
    start = start - vehicles[index].fitness
```

```
    index++
  }
  index--
  return vehicles[index].brain
}

function reproduction()
{
  let nextVehicles = []
  for (let i = 0; i < vehicles.length; i++)
  {
    let parentA = weightedSelection()
    let parentB = weightedSelection()
    let child = parentA.crossover(parentB)
    child.mutate(0.01)
    nextVehicles[i] = new Vehicle(random(width), random(height), child)
  }
  vehicles = nextVehicles
}
```

Notes

Moving the **slider** along until around **150** iterations shows a huge improvement in their attraction to the target.

Challenge

Try other values

Figure D3.23



Algorithmic Intelligence Module D Unit #4 flappy bird



Module D Unit #4: flappy bird neural evolution

Flappy Bird is a relatively simple and yet very addictive game. It is a game that many emulate in various coding languages, and p5.js is no exception. In this module, we will start from scratch but not make a full working game. The basic game has a bird that flaps its wings and goes up when you tap the space bar (or click the mouse) and falls towards the ground when you stop. It has to fly through a series of gaps created by random pipes. The idea is to make it through all the pipes without hitting one for as long as possible.

We are going to give the bird a brain, a neural network, and see if we can train it to play the game through using a genetic algorithm approach, which is a form of reinforcement learning. This is another neural evolution solution similar to the smart cars example previously. That one was a regression task; this one is a classification task.

In the same process as before, those birds that last the longest will have the best fitness scores and will pass on their genes (weights) to the next generation (population).

Each bird will have a brain that has a random set of weights. We want to use. The following data in our neural network as inputs:

- The y position of the bird
- The bird's velocity
- The position of the top (or bottom) pipe
- The x position of the pipes

The reason that we can have four and not five inputs is that the distance between the position of the top and bottom pipe is a constant (gap). This is a classification problem because the output is either to jump up or not to jump; our outputs are:

- jump
- not jump

To summarise our bird brain, we could have a hidden layer of, say, eight nodes, so the feedforward neural network will have the following:

- 4 inputs
- 8 hidden nodes
- 2 outputs
- plus 2 biases



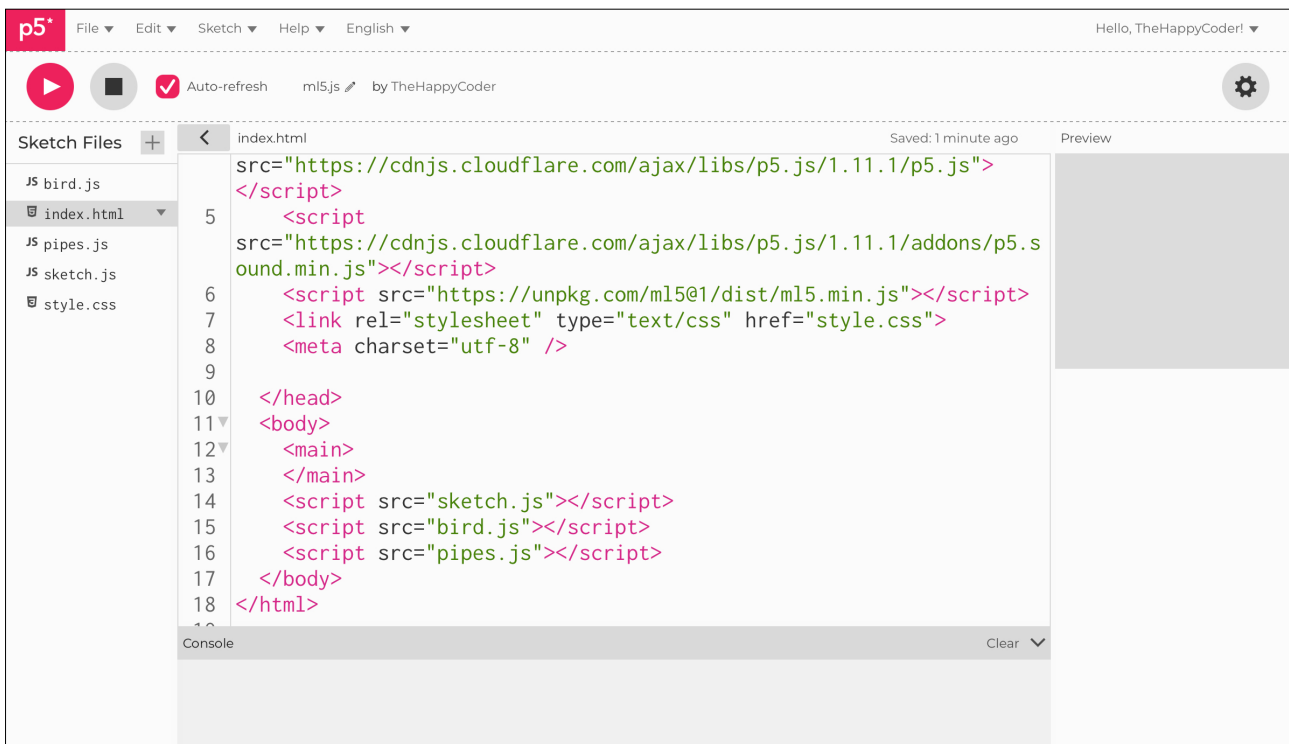
The basic game

First, we need to code the basic game before we do anything clever with neural networks and genetic algorithms. We need two files on top of the sketch.js main file. We also need to add the ml5.js line of code to the index.html file. The two named files are:

bird.js
pipes.js

You will have done this in the previous unit, so I won't repeat the process here in any detail. What you should have is shown in Fig.1 below.

Figure 1





The index.html file (don't forget)

Adding the ml5.js line of code, as well as the files.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
  <script src="bird.js"></script>
  <script src="pipes.js"></script>
</body>
</html>
```




Sketch D4.1 starting sketch

Our main starting sketch in `sketch.js`, with a wider and thinner canvas of **600** by **200**.

```
sketch.js

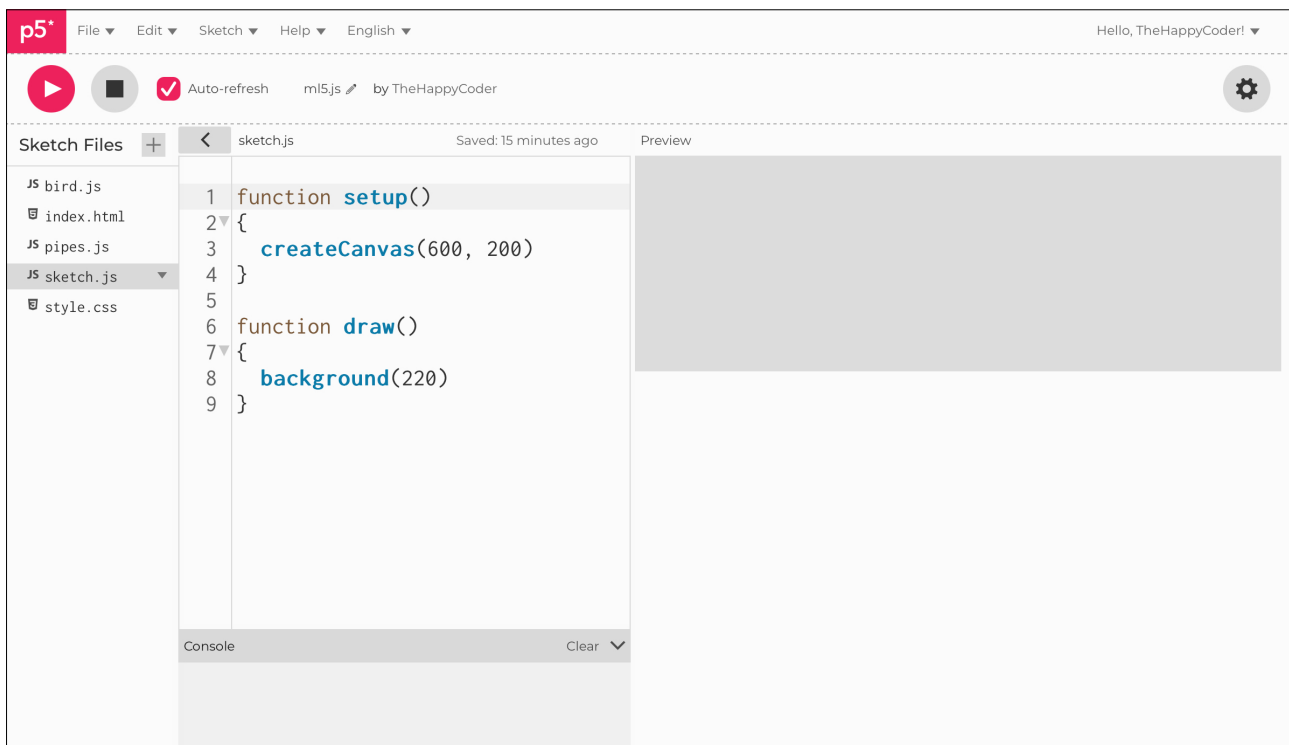
function setup()
{
  createCanvas(600, 200)
}

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

Notes

We have a long, thin canvas because of the nature of the game; you can do it with **400** by **400** if you wish.

Figure D4.1





Sketch D4.2 Bird class

! Move to `bird.js`.

In `bird.js`, we are going to create a `Bird` class with a `constructor()` function. It will have an `x` horizontal value, which will stay constant as the pipes effectively come towards it rather than the other way round. A `y` vertical value will vary when the mouse is clicked. The `velocity` will be zero initially. There will be some `gravity` so that it falls to the ground when the mouse is not pressed, and a `force` that pushes it upwards against `gravity` when the mouse is pressed.

`bird.js`

```
class Bird
{
  constructor()
  {
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
  }
}
```

Notes

Defining and initialising all the variables we need.

Code Explanation

<code>this.x = 50</code>	Fixed distance from the lefthand edge
<code>this.y = 120</code>	Starting vertical position
<code>this.velocity = 0</code>	Starting velocity
<code>this.gravity = 0.5</code>	Arbitrary gravity value acting downwards
<code>this.force = -10</code>	Flap force acting upwards when mouse clicked, hence negative value



Sketch D4.3 gravity

When the bird flaps its wings (metaphorically speaking), the **force** (a form of acceleration) is added to the vertical **velocity**. **Gravity** is acting on the bird all the time (another acceleration), and the resultant **velocity** is added to the **y** component of its position. And to give some realism(!), we add some damping (**0.95**). Also, when it lands on the floor, it just stays there. The bird is represented by a circle. We will now have our usual functions **move()** and **show()** plus another one called **flap()** which is the equivalent to the `applyForce()` previously.

bird.js

```
class Bird
{
  constructor()
  {
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
  }

  flap()
  {
    this.velocity += this.force
  }

  move()
  {
    this.velocity += this.gravity
    this.y += this.velocity
    this.velocity *= 0.95
    if (this.y > height)
    {
      this.y = height
      this.velocity = 0
    }
  }

  show()
```

```

{
  stroke(0)
  noFill()
  circle(this.x, this.y, 20)
}
}

```

Notes

At this point, we are just creating the game and doing it quickly. We want to focus on the main event, which is the neural evolution part. For now, just take the time to understand the code without there being too much explanation at this point from me.

Code Explanation

<code>this.velocity += this.force</code>	This is the upward force (negative 10) when the bird flaps its wings
<code>this.velocity += this.gravity</code>	Adding gravity to the velocity
<code>this.y += this.velocity</code>	Adding the velocity to the vertical position
<code>this.velocity *= 0.95</code>	Adding a bit of simulation
<code>if (this.y > height)</code>	Checks to see if it has hit the floor
<code>this.y = height</code>	Stays on the floor
<code>this.velocity = 0</code>	Reduce the velocity to zero on the floor until it flaps



Sketch D4.4 pipes

! Go to the `pipes.js` file.

Now, to add the `pipes`, we have two pipes, one at the top and one at the bottom. We have a `constructor()` function for the pipes, which will have a vertical space between the pipes (`100`), a random top position for a pipe, and a corresponding bottom position for a pipe. Each pipe will be `20` wide, and the velocity of the pipes has the value of `2`.

`pipes.js`

```
class Pipe
{
  constructor()
  {
    this.spacing = 100
    this.top = random(height - this.spacing)
    this.bottom = this.top + this.spacing
    this.x = width
    this.w = 20
    this.velocity = 2
  }
}
```

Notes

This just sets the variables for the simple pipes.

Code Explanation

<code>this.spacing = 100</code>	This is the gap between the top and bottom pipe and is what flappy bird is going to fly through
<code>this.top = random(height - this.spacing)</code>	The position of the top pipe, at some random position
<code>this.bottom = this.top + this.spacing</code>	The bottom pipe is relevant to the top pipe plus the gap (spacing)
<code>this.x = width</code>	The pipe starts at the far righthand edge
<code>this.w = 20</code>	The width of the pipe
<code>this.velocity = 2</code>	The velocity of the pipe moving



Sketch D4.5 drawing the pipes

We will draw the pipes as simple rectangles. Remember, we aren't trying to recreate all the full features of the game, just the minimum. The **velocity** for the pipes is not the same as the velocity for the bird going up and down. We add the **show()** and **move()** functions.

pipes.js

```
class Pipe
{
  constructor()
  {
    this.spacing = 100
    this.top = random(height - this.spacing)
    this.bottom = this.top + this.spacing
    this.x = width
    this.w = 20
    this.velocity = 2
  }

  show()
  {
    fill(51)
    noStroke()
    rect(this.x, 0, this.w, this.top)
    rect(this.x, this.bottom, this.w, height - this.bottom)
  }

  move()
  {
    this.x -= this.velocity
  }
}
```

Notes

We won't see the bird or the pipes just yet; we will need to add these functions to the main sketch.

Code Explanation

<code>rect(this.x, 0, this.w, this.top)</code>	Top pipe rectangle
<code>rect(this.x, this.bottom, this.w, height - this.bottom)</code>	Bottom pipe rectangle
<code>this.x -= this.velocity</code>	Moves the pipes from right to left hence the negative increment



Sketch D4.6 collision

We need to know when the bird collides with the pipes. We will put this collision in the `pipe.js` class. `vColl` means vertical collision and logically `hColl` means horizontal collision. If you think through where the bird (as a single point `(x, y)`) is in relation to the pipes, you can understand the collision part. The argument (`bird`) will be the position of that one particular bird, as there will be many of them at the start.

pipes.js

```
class Pipe
{
  constructor()
  {
    this.spacing = 100
    this.top = random(height - this.spacing)
    this.bottom = this.top + this.spacing
    this.x = width
    this.w = 20
    this.velocity = 2
  }

  show()
  {
    fill(51)
    noStroke()
    rect(this.x, this.w, this.top)
    rect(this.x, this.bottom, this.w, height - this.bottom)
  }

  move()
  {
    this.x -= this.velocity
  }

  collides(bird)
  {
    let vColl = bird.y < this.top || bird.y > this.bottom
    let hColl = bird.x > this.x && bird.x < this.x + this.w
    return vColl && hColl
  }
}
```



```
}
```

Notes

You will realise that nothing happens when you try to run this because we haven't put anything in `draw()` just yet. So, let's get something happening on the canvas, let's make a playable, if rudimentary, Flappy Bird.

Code Explanation

<code>vColl = bird.y < this.top bird.y > this.bottom</code>	Checks to see if either one of these two conditions are true
<code>hColl = bird.x > this.x && bird.x < this.x + this.w</code>	Checks to see if both conditions are true at the same time
<code>return vColl && hColl</code>	Returns true or false



Sketch D4.7 drawing the bird!

! Return to `sketch.js`.

We need to create the bird, which will be a simple circle, and lots of pipes moving across the canvas. Each bird will be a separate bird, whereas we will have an array to keep track of the pipes.

sketch.js
<pre>let bird let pipes = [] function setup() { createCanvas(600, 200) bird = new Bird() pipes.push(new Pipe()) } function draw() { background(220) }</pre>

Notes

Nowt (northern for nothing) to see yet.

Code Explanation

<code>let bird</code>	Our bird variable
<code>let pipes = []</code>	An empty array of pipes
<code>bird = new Bird()</code>	Create a new bird
<code>pipes.push(new Pipe())</code>	Adding new pipes to the array



Sketch D4.8 mouse pressed

To make the bird fly, we click the mouse (usually, we press the space bar).

sketch.js

```
let bird
let pipes = []

function setup()
{
  createCanvas(600, 200)
  bird = new Bird()
  pipes.push(new Pipe())
}

function mousePressed()
{
  bird.flap()
}

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

Notes

Still nothing to see yet.

Code Explanation

function mousePressed()	Checks to see if mouse is clicked (pressed)
bird.flap()	Calls the flap() function in Bird class



Sketch D4.9 oops

Now, to draw the pipes and the bird, we will have to say **oops** when we hit the pipe and remember to press the mouse to fly/jump. We draw a **pipe** every **100** frames using modulo (%) and by counting the number of frames.

sketch.js

```
let bird
let pipes = []

function setup()
{
  createCanvas(600, 200)
  bird = new Bird()
  pipes.push(new Pipe())
}

function mousePressed()
{
  bird.flap()
}

function draw()
{
  background(220)

  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].collides(bird))
    {
      text("OOPS", 50, height/2)
    }
  }
  bird.move()
  bird.show()
  if (frameCount % 100 === 0)
  {
    pipes.push(new Pipe())
  }
}
```

```

}
}

```

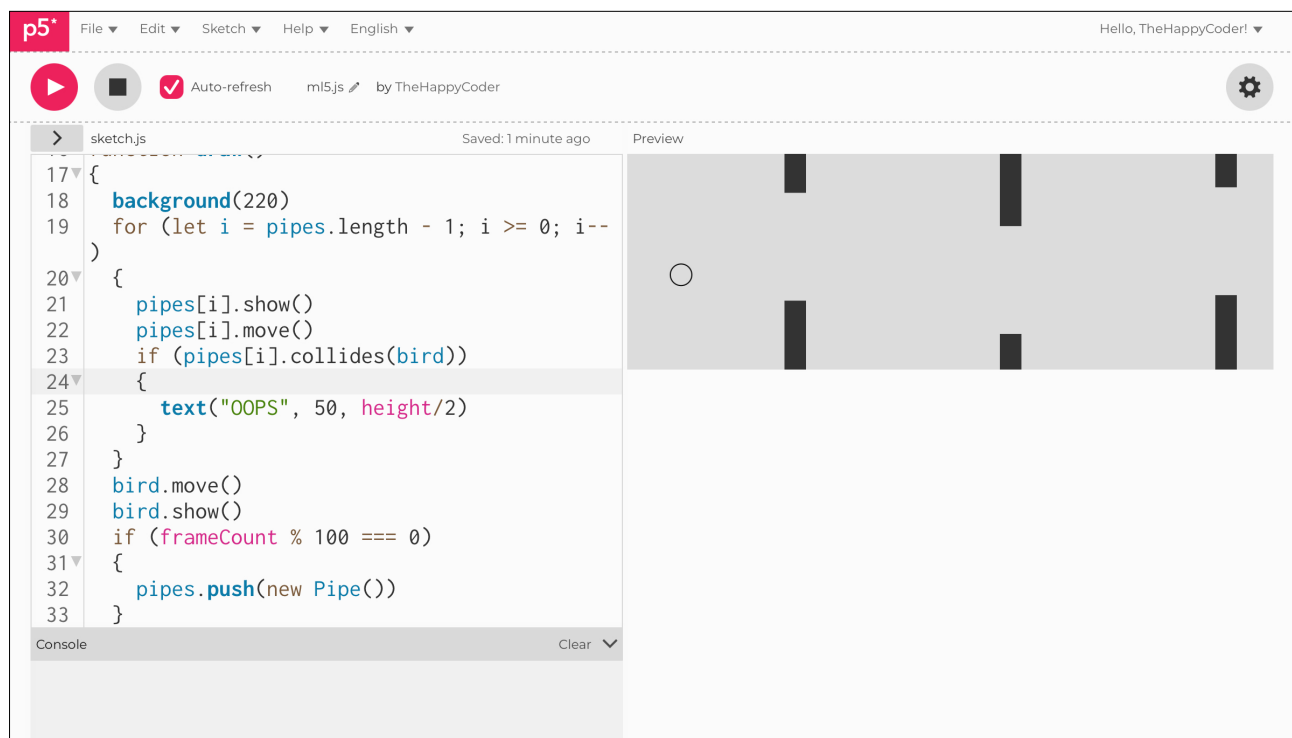
Notes

If you click the **mouse** (or mousepad), the **bird** (circle) should move upwards, and when it hits a pipe, you get an **oops** on the canvas. The **for()** loop for the pipes works backwards so that it encounters the next pipe in the array, which is being added to constantly. The array will just keep on growing, something we will address later on. The frame rate is how often the screen refreshes; it is a continuous number.

Code Explanation

<code>for (let i = pipes.length - 1; i >= 0; i--)</code>	This has the effect of always being then next pipe approaching the bird
<code>pipes[i].show()</code>	Draws each set of pipes created
<code>pipes[i].move()</code>	Moves that pipe
<code>if (pipes[i].collides(bird))</code>	Checks to see if the collision conditions have been met
<code>text("OOPS", 50, height/2)</code>	If it has then text oops on canvas
<code>if (frameCount % 100 === 0)</code>	The % gives you the remainder, so if it is exactly divisible by a hundred then. . .
<code>pipes.push(new Pipe())</code>	. . .make a new pipe

Figure D4.9





Sketch D4.10 culling the array

We have a slight problem. When the pipes go off the edge of the canvas, they need to be removed; otherwise, the array will just get bigger and bigger. Eventually, over time, the game will slow down. We use `splice()` to remove them from the array.

sketch.js

```
let bird
let pipes = []

function setup()
{
  createCanvas(600, 200)
  bird = new Bird()
  pipes.push(new Pipe())
}

function mousePressed()
{
  bird.flap()
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].collides(bird))
    {
      text("OOPS", 50, height/2)
    }
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  bird.move()
}
```

```
bird.show()
if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}
```

Notes

You will get an error because we haven't created the `offscreen()` function in `pipe.js` yet.

Code Explanation

<code>for (let i = pipes.length - 1; i >= 0; i--)</code>	Cycles backwards through the array of pipes
<code>if (pipes[i].offscreen())</code>	Checks to see if the pipe has gone off the canvas
<code>pipes.splice(i, 1)</code>	It it has remove that pipe



Sketch D4.11 splice the pipe

! Wander over to `pipes.js`.

The pipe is removed from the array if the `x` value of that pipe (`this.x`) is less than `-20` (the width of the pipe); in other words, it has fully gone off the edge of the canvas. Here we add the `offscreen()` function.

pipes.js

```
class Pipe
{
  constructor()
  {
    this.spacing = 100
    this.top = random(height - this.spacing)
    this.bottom = this.top + this.spacing
    this.x = width
    this.w = 20
    this.velocity = 2
  }

  show()
  {
    fill(51)
    noStroke()
    rect(this.x, 0, this.w, this.top)
    rect(this.x, this.bottom, this.w, height - this.bottom)
  }

  move()
  {
    this.x -= this.velocity
  }

  collides(bird)
  {
    let vColl = bird.y < this.top || bird.y > this.bottom
    let hColl = bird.x > this.x && bird.x < this.x + this.w
    return vColl && hColl
  }
}
```



```
    offscreen()
    {
        return this.x < -this.w
    }
}
```

Notes

It should work perfectly now. This is all to do with managing its memory. Your browser only has so much memory, and clogging it up like this is not great and may crash or grind to a halt eventually.

Code Explanation

```
return this.x < -this.w
```

Checks to see if this condition is true



Adding the brain

All we have done so far is make a simplified functional version of the game. We are now going to develop the idea of a genetic algorithmic approach to getting the bird to fly through the pipes perfectly without hardcoding it, but instead using a neural evolution technique.

This is very similar to the smart cars, so it isn't completely new for you (assuming you completed that unit). Just make sure you have the ml5.js installed in the index.html file. I will still walk you through it just in case you haven't or have forgotten.



Sketch D4.12 simple bird brain

! Go to `bird.js`.

We are going to give the bird a `brain` as an ml5.js neural network. We place this brain in the `constructor()` part of the class. When you run this, you should still get the same, but check for any error messages in the console to make sure you have typed everything in OK.

bird.js

```
class Bird
{
  constructor()
  {
    ml5.setBackend("webgl")
    this.brain = ml5.neuralNetwork(
    {
      inputs: 4,
      outputs: ["flap", "no flap"],
      task: "classification",
      noTraining: true,
      neuroEvolution: true
    })

    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
  }

  flap()
  {
    this.velocity += this.force
  }

  move()
  {
    this.velocity += this.gravity
    this.y += this.velocity
    this.velocity *= 0.95
  }
}
```

```

    if (this.y > height)
    {
        this.y = height
        this.velocity = 0
    }
}
show()
{
    stroke(0)
    noFill()
    circle(this.x, this.y, 20)
}
}

```

Notes

We haven't connected the bird and brain together yet; the bird has no brain!

Code Explanation

inputs: 4	There are four inputs
outputs: ["flap", "no flap"]	Our two outputs
task: "classification"	Task declared
noTraining: true	Not doing any training
neural evolution: true	Declaring it is a neuro evolution network



Sketch D4.13 next pipe please

This is where we need to think a bit about what is happening. Once the pipe has gone past the bird, it is no longer relevant. We already delete it when it goes off the canvas. But we need a way of checking the distance to the next one in front of the bird, not just the one at the front of the array. So in `bird.js`, we need a function to measure that. The `break` command simply terminates that checking loop. The new function `think(pipes)` has the argument from the array of pipes.

bird.js

```
class Bird
{
  constructor()
  {
    ml5.setBackend("webgl")
    this.brain = ml5.neuralNetwork(
    {
      inputs: 4,
      outputs: ["flap", "no flap"],
      task: "classification",
      noTraining: true,
      neural evolution: true
    })
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
  }
}
```

```
think(pipes)
{
  let nextPipe = null
  for (let pipe of pipes)
  {
    if (pipe.x + pipe.w > this.x)
    {
      nextPipe = pipe
      break
    }
  }
}
```

```

    }
  }

  flap()
  {
    this.velocity += this.force
  }

  move()
  {
    this.velocity += this.gravity
    this.y += this.velocity
    this.velocity *= 0.95
    if (this.y > height)
    {
      this.y = height
      this.velocity = 0
    }
  }

  show()
  {
    stroke(0)
    noFill()
    circle(this.x, this.y, 20)
  }
}

```

Notes

Just check each pipe in turn.

Code Explanation

<code>nextPipe = null</code>	The term null means empty or no value
<code>for (let pipe of pipes)</code>	Goes through the array of pipes pulling out a pipe at a time
<code>nextPipe = pipe</code>	The nextPipe now has a value
<code>break</code>	Stops the loop



Sketch D4.14 input data

Now we have the next pipe, we can add it to the data inputs for our neural network. These are our four inputs.

bird.js

```
class Bird
{
  constructor()
  {
    ml5.setBackend("webgl")
    this.brain = ml5.neuralNetwork(
    {
      inputs: 4,
      outputs: ["flap", "no flap"],
      task: "classification",
      noTraining: true,
      neuroEvolution: true,
    })
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
  }

  think(pipes)
  {
    let nextPipe = null
    for (let pipe of pipes)
    {
      if (pipe.x + pipe.w > this.x)
      {
        nextPipe = pipe
        break
      }
    }
  }

  let inputs = [
    this.y,
```

```

        this.velocity,
        nextPipe.top,
        nextPipe.x - this.x
    ]
}

flap()
{
    this.velocity += this.force
}

move()
{
    this.velocity += this.gravity
    this.y += this.velocity
    this.velocity *= 0.95
    if (this.y > height)
    {
        this.y = height
        this.velocity = 0
    }
}

show()
{
    stroke(0)
    noFill()
    circle(this.x, this.y, 20)
}
}

```


Notes

We now have our four **inputs**, and the **nextPipe** will be the pipe it sees.

Code Explanation

<code>this.y</code>	Input 1: the vertical position
<code>this.velocity</code>	Input 2: the current velocity of the bird
<code>nextPipe.top</code>	Input 3: the vertical position of the top pipe
<code>nextPipe.x - this.x</code>	Input 4: how far away is the next pipe



Sketch D4.15 normalise

To **normalise** all the **inputs**, we can divide any vertical measurements by the height, which are the first three **inputs**. The fourth **input** is the horizontal distance, so one is divided by the width. This gives us values between **0** and **1**.

bird.js

```
class Bird
{
  constructor()
  {
    ml5.setBackend("webgl")
    this.brain = ml5.neuralNetwork(
    {
      inputs: 4,
      outputs: ["flap", "no flap"],
      task: "classification",
      noTraining: true,
      neuroEvolution: true,
    })
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
  }

  think(pipes)
  {
    let nextPipe = null
    for (let pipe of pipes)
    {
      if (pipe.x + pipe.w > this.x)
      {
        nextPipe = pipe
        break
      }
    }
    let inputs = [
```

```

        this.y / height,
        this.velocity / height,
        nextPipe.top / height,
        (nextPipe.x - this.x) / width
    ]
}

flap()
{
    this.velocity += this.force
}

move()
{
    this.velocity += this.gravity
    this.y += this.velocity
    this.velocity *= 0.95
    if (this.y > height)
    {
        this.y = height
        this.velocity = 0
    }
}

show()
{
    stroke(0)
    noFill()
    circle(this.x, this.y, 20)
}
}

```

Notes

Everything is now normalised.



Sketch D4.16 synchronicity

We want the code to run **synchronously** so that the model waits for the **inputs** before carrying on. We use something a bit useful in ml5.js called **classifySync()**, which is the same as **classify** but runs synchronously. If the result is classified as **flap**, then the function **flap()** is run; otherwise, nothing will happen.

bird.js

```
class Bird
{
  constructor()
  {
    ml5.setBackend("webgl")
    this.brain = ml5.neuralNetwork(
    {
      inputs: 4,
      outputs: ["flap", "no flap"],
      task: "classification",
      noTraining: true,
      neuroEvolution: true,
    })
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
  }

  think(pipes)
  {
    let nextPipe = null
    for (let pipe of pipes)
    {
      if (pipe.x + pipe.w > this.x)
      {
        nextPipe = pipe
        break
      }
    }
  }
}
```

```

    let inputs = [
      this.y / height,
      this.velocity / height,
      nextPipe.top / height,
      (nextPipe.x - this.x) / width
    ]

    let results = this.brain.classifySync(inputs)
    if (results[0].label === "flap")
    {
      this.flap()
    }
  }

  flap()
  {
    this.velocity += this.force
  }

  move()
  {
    this.velocity += this.gravity
    this.y += this.velocity
    this.velocity *= 0.95
    if (this.y > height)
    {
      this.y = height
      this.velocity = 0
    }
  }

  show()
  {
    stroke(0)
    noFill()
    circle(this.x, this.y, 20)
  }
}

```

Notes

Do remember that there is no training; each bird has a different set of random weights. The outputs will be effectively random despite having the data inputs; it has no idea it has to jump or even why it should jump; it does not even know the rules of the game.

Code Explanation

<code>results = this.brain.classifySync(inputs)</code>	From the input data we classify the output (results)
<code>if (results[0].label === "flap")</code>	If the first result is flap then. . .
<code>this.flap()</code>	. . .flap like a bird



Sketch D4.17 population of birds

! Return to [sketch.js](#).

We only currently have one bird. We need a population of birds all trying to be the best bird they can be. We will make an array of **200** new birds. We need an array of birds, and so we use a **for()** loop to create this population of birds.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
}

function mousePressed()
{
  bird.flap()
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].collides(bird))
    {
      text("00PS", 50, height/2)
    }
    if (pipes[i].offscreen())
```

```

    {
        pipes.splice(i, 1)
    }
}
bird.move()
bird.show()
if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}

```

Notes

No point in running it yet; we have a few more amendments to make.

Code Explanation

let birds = []	An empty array of birds replaces singular bird
for (let i = 0; i < population; i++)	Creating a population of birds
birds[i] = new Bird()	Adding the birds to the array



Sketch D4.18 to flap or not to flap

We are now going to get the bird to think whether it should flap or not, update its decision, and then show us what it can do. However, it is still too early to run this just yet; we are still laying the foundation. Put `bird.move()` and `bird.show()` inside a new `for-of()` loop and call the `think(pipes)` function.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
}

function mousePressed()
{
  bird.flap()
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].collides(bird))
    {
      text("OOPS", 50, height/2)
    }
    if (pipes[i].offscreen())
```

```
    {
      pipes.splice(i, 1)
    }
  }

  for (let bird of birds)
  {
    bird.think(pipes)
    bird.move()
    bird.show()
  }

  if (frameCount % 100 === 0)
  {
    pipes.push(new Pipe())
  }
}
```

Notes

We cycle through all the birds one at a time.



Sketch D4.19 GPU v CPU

We can use the **GPU** or the **CPU**. I won't go into the difference here except to say that we want to use the **CPU** for better performance. The **GPU** is better when there is a heavy demand for graphics in some games, for instance. The **.tf** reference is for TensorFlow, which underpins ml5.js. TensorFlow is used with Python for machine learning, but there is a JavaScript version called TensorFlow.js, which is compatible with ml5.js and p5.js.

! If it is still running slow or very jerky, then suggest reducing the number of birds, or try **webgl**.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function mousePressed()
{
  bird.flap()
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
```

```
    if (pipes[i].collides(bird))
    {
        text("OOPS", 50, height/2)
    }
    if (pipes[i].offscreen())
    {
        pipes.splice(i, 1)
    }
}
for (let bird of birds)
{
    bird.think(pipes)
    bird.move()
    bird.show()
}
if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}
```

Notes

I found that this did make quite a difference depending on what device/machine I used. The other alternative might be [webgl](#). Play around and see what works best for you.



Sketch D4.20 bird fitness

! Hop over to `bird.js`.

Next job is to define and find the `fitness` of each bird. We need to add two more features, `fitness` and `alive`, in the `constructor()` function. The bird's `fitness` obviously increases the longer it is `alive`; hence, `fitness` is a number, but `alive` is a boolean; it is either alive or not, think of `oops` being a bit more terminal. We increment the `fitness` in the `move()` function.

bird.js

```
class Bird
{
  constructor()
  {
    ml5.setBackend("webgl")
    this.brain = ml5.neuralNetwork(
    {
      inputs: 4,
      outputs: ["flap", "no flap"],
      task: "classification",
      noTraining: true,
      neuroEvolution: true,
    })
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
    this.fitness = 0
    this.alive = true
  }

  think(pipes)
  {
    let nextPipe = null
    for (let pipe of pipes)
    {
      if (pipe.x + pipe.w > this.x)
      {
```

```

        nextPipe = pipe
        break
    }
}
let inputs = [
    this.y / height,
    this.velocity / height,
    nextPipe.top / height,
    (nextPipe.x - this.x) / width
]
let results = this.brain.classifySync(inputs)
if (results[0].label === "flap")
{
    this.flap()
}

flap()
{
    this.velocity += this.force
}

move()
{
    this.velocity += this.gravity
    this.y += this.velocity
    this.velocity *= 0.95
    if (this.y > height)
    {
        this.y = height
        this.velocity = 0
    }
    this.fitness++
}

show()
{
    stroke(0)

```

```
    noFill()
    circle(this.x, this.y, 20)
  }
}
```

Notes

When the bird hits the pipe, we will want to eliminate it, remove it (sounds less brutal).

Code Explanation

<code>this.fitness = 0</code>	Each bird has a fitness of zero
<code>this.alive = true</code>	Each bird is initialised as being alive at the start
<code>this.fitness++</code>	Increment the bird's fitness on each iteration.



Sketch D4.21 the bird is dead

! Return to [sketch.js](#).

However, we need a dead bird (sorry) when it hits a pipe. So, we have to do a bit of refactoring to incorporate this here and also in the next sketch. We have a nested loop to achieve this.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function mousePressed()
{
  bird.flap()
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].collides(bird))
    {
      text("00PS", 50, height/2)
    }
  }
}
```



```

    if (pipes[i].offscreen())
    {
        pipes.splice(i, 1)
    }
}
for (let bird of birds)
{
    if (bird.alive)
    {
        bird.think(pipes)
        bird.move()
        bird.show()
    }
}
if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}

```

Notes

The only birds left are those which are alive (true), obviously.

Code Explanation

```
if (bird.alive)
```

Checks to see if this bird is alive (true) or dead (false)



Sketch D4.22 collision

Now, when the bird collides with the pipe. We do a bit more reconfiguring. Also, we remove the lines of code which are commented out (`//`). We are no longer operating the birds with the mouse. We eliminate the birds if they collide with the pipe.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

// function mousePressed()
// {
//   bird.flap()
// }

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()

    // if (pipes[i].collides(bird))
    // {
    //   text("00PS", 50, height/2)
    // }

    if (pipes[i].offscreen())
```

```

    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
    if (bird.alive)
    {
      for (let pipe of pipes)
      {
        if (pipe.collides(bird))
        {
          bird.alive = false
        }
      }

      bird.think(pipes)
      bird.move()
      bird.show()
    }
  }
  if (frameCount % 100 === 0)
  {
    pipes.push(new Pipe())
  }
}

```

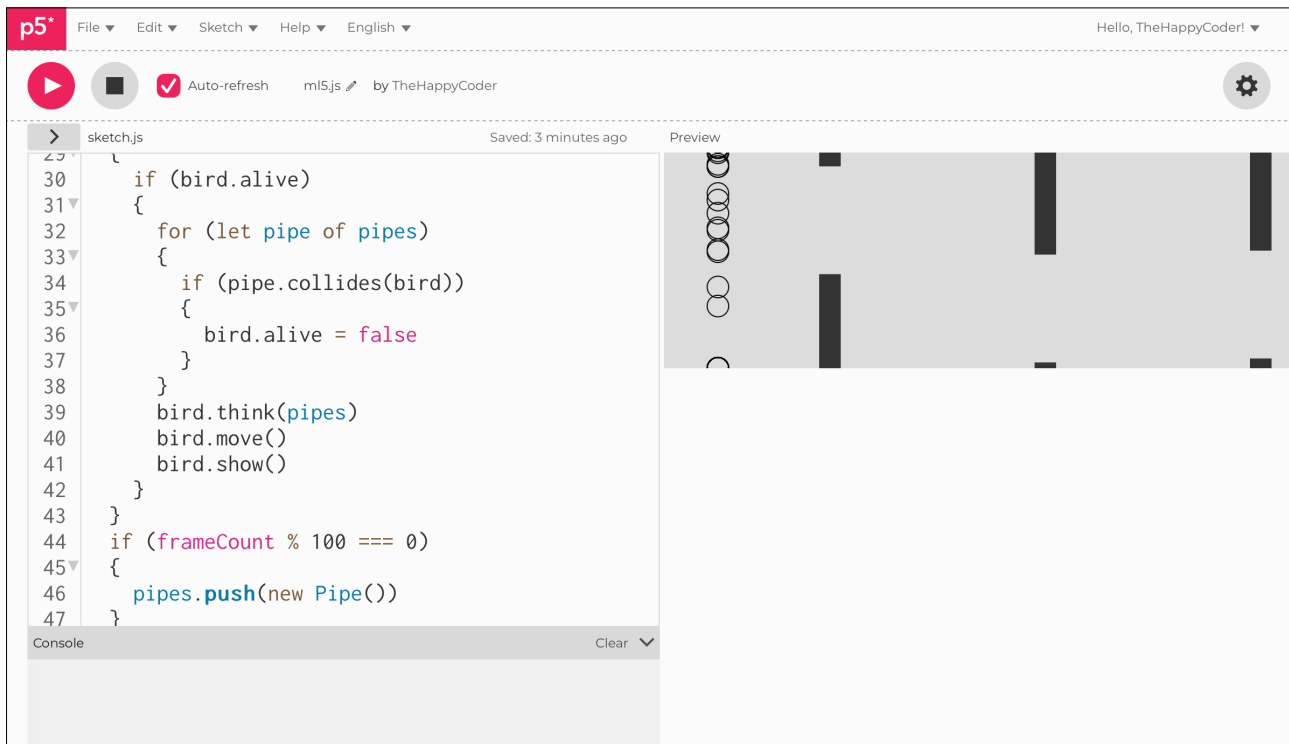
Notes

We now have a working population of birds trying to get through the pipes; however, they only last for one population, and once they have all died, there are none more. We want to reproduce the next generation.

Code Explanation

<code>for (let pipe of pipes)</code>	Goes through each pipe in turn
<code>if (pipe.collides(bird))</code>	Checks for any collisions
<code>bird.alive = false</code>	If a collision occurs that bird is now dead (false)

Figure D4.22





Sketch D4.23 is anyone there

There is no predetermined lifespan. So they live as long as they don't hit a pipe, and when they have all died out, then we select the best ones. We create a function to check if there are any birds left; if there are still some birds flapping, it returns false and carries on checking.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
    if (bird.alive)
    {
```

```
    for (let pipe of pipes)
    {
        if (pipe.collides(bird))
        {
            bird.alive = false
        }
    }
    bird.think(pipes)
    bird.move()
    bird.show()
}
}
if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}
```

```
function allBirdsDead()
{
    for (let bird of birds)
    {
        if (bird.alive)
        {
            return false
        }
    }
    return true
}
```

Notes

Not an imaginative name for a function, but it does what it says on the tin!

Code Explanation

<code>for (let bird of birds)</code>	Check through all the birds
<code>if (bird.alive)</code>	If a there is an alive bird then. . .
<code>return false</code>	. . .the function <code>allBirdsDead()</code> is false, until. . .
<code>return true</code>	. . .there are no alive birds



Sketch D4.24 mating the best ones

What we want to do now is select the fittest birds and mate them to make even better birds for the next round. This is an evolutionary approach to selection. We will use the `weightedSelection()` algorithm that we used in smart cars. To remind you, the fittest have the advantage, but the others always have a chance, so it is a bit more like how natural selection occurs.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
    if (bird.alive)
    {
```



```

    for (let pipe of pipes)
    {
        if (pipe.collides(bird))
        {
            bird.alive = false
        }
    }
    bird.think(pipes)
    bird.move()
    bird.show()
}

if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}

```

```

function allBirdsDead()
{
    for (let bird of birds)
    {
        if (bird.alive)
        {
            return false
        }
    }
    return true
}

```

```

function weightedSelection()
{
    let index = 0
    let start = random(1)
    while (start > 0)
    {
        start = start - birds[index].fitness
        index++
    }
}

```

```
}  
index--  
return birds[index].brain  
}
```

Notes

We continue to put the pieces together; we still have to **normalise** the fitness values. We went through the code for weighted selection in some detail in smart cars.



Sketch D4.25 normalising fitness

Now, to normalise the fitness, we use the `for-of()` loop to add up all the fitness scores of all the birds (`sum`) and then go through each one and divide by said `sum`. This means that we have fitness scores between `0` and `1`, and they all add up to `1`.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
    if (bird.alive)
    {
      for (let pipe of pipes)
```

```

    {
        if (pipe.collides(bird))
        {
            bird.alive = false
        }
    }
    bird.think(pipes)
    bird.move()
    bird.show()
}

if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}

function allBirdsDead()
{
    for (let bird of birds)
    {
        if (bird.alive)
        {
            return false
        }
    }
    return true
}

function weightedSelection()
{
    let index = 0
    let start = random(1)
    while (start > 0)
    {
        start = start - birds[index].fitness
        index++
    }
}

```

```
    index--  
    return birds[index].brain  
}  
  
function normaliseFitness()  
{  
    let sum = 0  
    for (let bird of birds)  
    {  
        sum += bird.fitness  
    }  
    for (let bird of birds)  
    {  
        bird.fitness = bird.fitness / sum  
    }  
}
```

Notes

This should be fairly straightforward. Another piece of the jigsaw. The code was covered in smart cars.



Sketch D4.26 crossover

Now we tackle **crossover**, where we combine the DNA or genes of two parents that are successful or have a high **fitness** level. This could be very problematic because all you are doing is choosing the weights of one neural network or the other. What ml5.js has is a built-in function for this called **crossover()**, and we create a **reproduction()** function to calculate all this.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].move()
    pipes[i].show()
    // pipes[i].move()
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
    if (bird.alive)
```

```

    {
      for (let pipe of pipes)
      {
        if (pipe.collides(bird))
        {
          bird.alive = false
        }
      }
      bird.think(pipes)
      bird.move()
      bird.show()
    }
  }
  if (frameCount % 100 === 0)
  {
    pipes.push(new Pipe())
  }
}

```

```

function allBirdsDead()
{
  for (let bird of birds)
  {
    if (bird.alive)
    {
      return false
    }
  }
  return true
}

```

```

function weightedSelection()
{
  let index = 0
  let start = random(1)
  while (start > 0)
  {
    start = start - birds[index].fitness
  }
}

```

```
    index++
  }
  index--
  return birds[index].brain
}

function normaliseFitness()
{
  let sum = 0
  for (let bird of birds)
  {
    sum += bird.fitness
  }
  for (let bird of birds)
  {
    bird.fitness = bird.fitness / sum
  }
}
```

```
function reproduction()
{
  let parentA = weightedSelection()
  let parentB = weightedSelection()
  let child = parentA.crossover(parentB)
}
```

Notes

You can see how the `crossover()` function works.



Sketch D4.27 mutation

To **mutate**, we can use another new function included in ml5.js called (guess what) **mutate()**. Mutation is important because it creates some variety to prevent a poor initial selection from causing the birds to die out prematurely. The number indicates how often a weight is altered. **0.01** is **1%** of the population.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].move()
    pipes[i].show()
    // pipes[i].move()
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
    if (bird.alive)
```

```

    {
      for (let pipe of pipes)
      {
        if (pipe.collides(bird))
        {
          bird.alive = false
        }
      }
      bird.think(pipes)
      bird.move()
      bird.show()
    }
  }
  if (frameCount % 100 === 0)
  {
    pipes.push(new Pipe())
  }
}

```

```

function allBirdsDead()
{
  for (let bird of birds)
  {
    if (bird.alive)
    {
      return false
    }
  }
  return true
}

```

```

function weightedSelection()
{
  let index = 0
  let start = random(1)
  while (start > 0)
  {
    start = start - birds[index].fitness
  }
}

```

```

    index++
  }
  index--
  return birds[index].brain
}

function normaliseFitness()
{
  let sum = 0
  for (let bird of birds)
  {
    sum += bird.fitness
  }
  for (let bird of birds)
  {
    bird.fitness = bird.fitness / sum
  }
}

function reproduction()
{
  let parentA = weightedSelection()
  let parentB = weightedSelection()
  let child = parentA.crossover(parentB)
  child.mutate(0.01)
}

```

Notes

We do need some mutation.



Sketch D4.28 next generation

Once we have **crossover** and **mutation**, we need a new population. So we need a new (empty) array of birds, and we will call this **nextBirds = []**. This is part of the **reproduction()** function. After we have gone through the population and created the next birds from the children, we then call this new generation of birds the current population. This means we are rinsing and repeating with each successive population replacing the previous one, which we hope is better.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].move()
    pipes[i].show()
    // pipes[i].move()
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
```

```

    if (bird.alive)
    {
        for (let pipe of pipes)
        {
            if (pipe.collides(bird))
            {
                bird.alive = false
            }
        }
        bird.think(pipes)
        bird.move()
        bird.show()
    }
}

if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}
}

```

```

function allBirdsDead()
{
    for (let bird of birds)
    {
        if (bird.alive)
        {
            return false
        }
    }
    return true
}

```

```

function weightedSelection()
{
    let index = 0
    let start = random(1)
    while (start > 0)
    {

```

```

    start = start - birds[index].fitness
    index++
  }
  index--
  return birds[index].brain
}

function normaliseFitness()
{
  let sum = 0
  for (let bird of birds)
  {
    sum += bird.fitness
  }
  for (let bird of birds)
  {
    bird.fitness = bird.fitness / sum
  }
}

function reproduction()
{
  let nextBirds = []
  for (let i = 0; i < population; i++)
  {
    let parentA = weightedSelection()
    let parentB = weightedSelection()
    let child = parentA.crossover(parentB)
    child.mutate(0.01)
    nextBirds[i] = new Bird(child)
  }
  birds = nextBirds
}

```

Notes

A bit of refactoring, but a similar outcome to smart cars in the previous unit.



Sketch D4.29 a new brain

! Fly over to [bird.js](#).

The new birds still have no brains, so we need to alter the **Bird class** to correct this omission. If the bird has no brain, then create one; if it does (because it is the next generation), then use it. The **constructor()** function now receives an argument (**brain**), which is the new child brain from the **crossover**.

bird.js

```
class Bird
{
  constructor(brain)
  {
    if (brain)
    {
      this.brain = brain
    }
    else
    {
      ml5.setBackend("webgl")
      this.brain = ml5.neuralNetwork(
        {
          inputs: 4,
          outputs: ["flap", "no flap"],
          task: "classification",
          noTraining: true,
          neuroEvolution: true
        })
    }

    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
    this.fitness = 0
    this.alive = true
  }

  think(pipes)
```

```

{
  let nextPipe = null
  for (let pipe of pipes)
  {
    if (pipe.x + pipe.w > this.x)
    {
      nextPipe = pipe
      break
    }
  }
  let inputs = [
    this.y / height,
    this.velocity / height,
    nextPipe.top / height,
    (nextPipe.x - this.x) / width
  ]
  let results = this.brain.classifySync(inputs)
  if (results[0].label === "flap")
  {
    this.flap()
  }
}

flap()
{
  this.velocity += this.force
}

move()
{
  this.velocity += this.gravity
  this.y += this.velocity
  this.velocity *= 0.95
  if (this.y > height)
  {
    this.y = height
    this.velocity = 0
  }
}

```



```
    this.fitness++  
  }  
  
  show()  
  {  
    stroke(0)  
    noFill()  
    circle(this.x, this.y, 20)  
  }  
}
```

Notes

We have a brain for every generation.



Sketch D4.30 reset and off we go again

! Return to `sketch.js`.

Now, to complete the code to make this work. If all the birds are dead, then we need to reset the pipes. To do that, we create a new function called `resetPipes()`.

sketch.js

```
let birds = []
let pipes = []
let population = 200

function setup()
{
  createCanvas(600, 200)
  for (let i = 0; i < population; i++)
  {
    birds[i] = new Bird()
  }
  pipes.push(new Pipe())
  ml5.tf.setBackend("cpu")
}

function draw()
{
  background(220)
  for (let i = pipes.length - 1; i >= 0; i--)
  {
    pipes[i].show()
    pipes[i].move()
    if (pipes[i].offscreen())
    {
      pipes.splice(i, 1)
    }
  }
  for (let bird of birds)
  {
    if (bird.alive)
    {
```

```

    for (let pipe of pipes)
    {
        if (pipe.collides(bird))
        {
            bird.alive = false
        }
    }
    bird.think(pipes)
    bird.move()
    bird.show()
}

if (frameCount % 100 === 0)
{
    pipes.push(new Pipe())
}

if (allBirdsDead())
{
    normaliseFitness()
    reproduction()
    resetPipes()
}

}

function resetPipes()
{
    pipes.splice(0, pipes.length - 1)
}

function allBirdsDead()
{
    for (let bird of birds)
    {
        if (bird.alive)
        {
            return false
        }
    }
}

```

```

    return true
}

function weightedSelection()
{
    let index = 0
    let start = random(1)
    while (start > 0)
    {
        start = start - birds[index].fitness
        index++
    }
    index--
    return birds[index].brain
}

function normaliseFitness()
{
    let sum = 0
    for (let bird of birds)
    {
        sum += bird.fitness
    }
    for (let bird of birds)
    {
        bird.fitness = bird.fitness / sum
    }
}

function reproduction()
{
    let nextBirds = []
    for (let i = 0; i < population; i++)
    {
        let parentA = weightedSelection()
        let parentB = weightedSelection()
        let child = parentA.crossover(parentB)
        child.mutate(0.01)
    }
}

```

```
    nextBirds[i] = new Bird(child)
  }
  birds = nextBirds
}
```

Notes

The reason for deleting the pipes except the last one (`pipes.length - 1`) is because if we don't, it will try to find the distance to the next pipe, but there isn't one briefly. This stops us from getting an error message.

Code Explanation

```
pipes.splice(0, pipes.length - 1)
```

Deletes all the pipes except for the last one



Sketch D4.31 eliminating silly birds

! Here in `bird.js`.

One final tweak: we can also eliminate any birds that fly off the screen or hit the ground in `bird.js`.

bird.js

```
class Bird
{
  constructor(brain)
  {
    if (brain)
    {
      this.brain = brain
    }
    else
    {
      ml5.setBackend("webgl")
      this.brain = ml5.neuralNetwork(
        {
          inputs: 4,
          outputs: ["flap", "no flap"],
          task: "classification",
          noTraining: true,
          neuroEvolution: true
        })
    }
    this.x = 50
    this.y = 120
    this.velocity = 0
    this.gravity = 0.5
    this.force = -10
    this.fitness = 0
    this.alive = true
  }

  think(pipes)
  {
```

```

let nextPipe = null
for (let pipe of pipes)
{
  if (pipe.x + pipe.w > this.x)
  {
    nextPipe = pipe
    break
  }
}
let inputs = [
  this.y / height,
  this.velocity / height,
  nextPipe.top / height,
  (nextPipe.x - this.x) / width
]
let results = this.brain.classifySync(inputs)
if (results[0].label == "flap")
{
  this.flap()
}
}

flap()
{
  this.velocity += this.force
}

move()
{
  this.velocity += this.gravity
  this.y += this.velocity
  this.velocity *= 0.95
  if (this.y > height || this.y < 0)
  {
    this.alive = false
  }
  this.fitness++
}

```

```
show()  
{  
  stroke(0)  
  noFill()  
  circle(this.x, this.y, 20)  
}  
}
```

Notes

If you leave it running for a while, you should see the results as a single bird (although technically not necessarily) working its way through the maze of pipes successfully.

Challenges

There are a lot of things you could do to improve or change this flappy bird neural evolution programme.

1. Build a slider to speed it up, as we did with the smart cars.
2. Add an image of the flappy bird (see Games in the resources tab).

Code Explanation

<code>if (this.y > height this.y < 0)</code>	If the bird has gone below the bottom edge or gone above the top edge then. . .
<code>this.alive = false</code>	. . .the bird is dead