Vectors

It's perfectly possible to use the concepts of vectors (or, for that matter, complex numbers) when writing our code, whether or not the program realizes that's what we're doing. However, there are some things that will be made more efficient if we use a dedicated object class for them. The p5 vector class is one example. Compare these two snippets:

```
class Ball{
  constructor(x, y, vx, vy){
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.ax = 0;
    this.ay = 0.01;
  }
  move(){
    this.vx += this.ax;
    this.vy += this.ay;
    this.x += this.vx;
    this.y += this.vy;
  }
  display(){
    circle(this.x, this.y, 20);
  }
}
```

```
class Ball{
  constructor(x, y, vx, vy){
    this.pos = createVector(x, y);
    this.vel = createVector(vx, vy);
    this.acc = createVector(0, 0.01);
  }
  move(){
    this.vel.add(this.acc);
    this.pos.add(this.vel);
  }
  display(){
    circle(this.pos.x, this.pos.y, 20);
  }
}
```

This code saves a few lines, but more importantly it is more readable and therefore easier to write, debug and maintain.

Both of the classes shown above do exactly the same thing. In essence, the p5 vector class is nothing more than a way of storing x and y (and z if required) values, but with a bunch of useful methods built in.

Note: unlike the 'magic' methods of Python (the so-called 'dunder methods' starting with a double underscore, such as __add__ and __mul__), javascript has no way to co-opt built-in operators such as + and * for use in custom objects. You'll just have to get used to the slightly clunkier *v1.add(v2)* notation.

The ball

Let's start by making a ball using the ball class outlined above. In addition to the code for the class itself, we need just a couple more things to make an actual ball appear on the screen:

(the Ball class code has been minimised by clicking the arrow beside the name)

```
class Ball{...}


let ball;

```

Step 1: Declare the *ball* variable.
The ball needs to be accessible within both *setup* and *draw*, so we declare it in the global scope, outside either of those two functions.

```
function setup() {
  createCanvas(400, 400);
  ball = new Ball(width/4, height/4, 1, -1);
}

```

Step 2: Create the ball.
This can't be done at the same time as declaration because it relies on *width* and *height* which are p5 variables.

```
function draw() {
  background(220);
  ball.move();
  ball.display();
}
```

Step 3: Move and display the ball.
This should be done every time the *draw* function loops, to show motion.

Making improvements

I like to get an object on the screen early, because often the necessary improvements or enhancements are obvious once something visual starts to happen. It also makes debugging much easier, since I'll make a few errors at a time and hopefully catch them as I go, instead of having too many to chase down later.

In this case the first obvious enhancement is to stop our ball from disappearing off the screen. Maybe we can have it bounce if it hits an edge by reversing the velocity in the relevant direction.

```
7   move(){
8     this.vel.add(this.acc);
9     this.pos.add(this.vel);
10    if (this.pos.x < 0 || this.pos.x > width){
11      this.vel.x *= -1;
12    }
13    if (this.pos.y < 0 || this.pos.y > height){
14      this.vel.y *= -1;
15    }
16  }
```

Recall that, in JavaScript, || (a double pipe character) stands for the logical operator *or* (&& means *and*).

Notice how the ball appears to bounce just off the screen? That's because it's using the ball's position vector, which corresponds to its centre. See if you can fix that.

Making lots of them

One of the main advantages to using classes is the ability to create multiple objects. We don't even need to finish the class code before we make them, since updating the class code will update all instances for us.

Firstly, instead of declaring a single variable *ball*, let's make an empty array called *balls* (and choose a number to make).

```
22  let n = 4;
23  let balls = [];
```

Then, in the *setup* function, run through a *for* loop pushing new Ball objects into the array. Notice that I've made the initial horizontal speed depend on *i* so they're not all identical.

```
25  function setup() {
26    createCanvas(400, 400);
27    for (let i=0; i<n; i++){
28      balls.push(new Ball(width/4, height/4, i/n, -1));
29    }
30  }
```

Finally, we'll also need a loop in the *draw* function which will update and animate all balls within the array.

```
30  function draw() {
31    background(220);
32    for (let i=0; i<balls.length; i++){
33      balls[i].move();
34      balls[i].display();
35    }
36  }
```

A bit of friction

We can make things more realistic by adding in some friction. We'll need a few more vector methods for this, since I want the friction (well, air resistance, really) to depend on not just the speed but also the direction of motion of the particle at any one moment in time.

```
7   applyFriction(){
8     let fr = this.vel.copy();
9     fr.mult(-0.01);
10    this.vel.add(fr);
11  }
12  move(){
13    this.applyFriction();
```

This uses the *copy* method to create a new vector identical to the velocity vector, then the *mult* method to make it 100 times smaller, and in the opposite direction. Then the *add* method finally applies the friction vector to the velocity. Note that we could have just multiplied the velocity directly by 0.99, but if I want to apply various forces, it'll be a helpful template to use.

A curious artefact

One of the most frustrating things I found when I first started to code was that any time I attempted to make objects bounce around the screen, sooner or later they would end up off an edge, and then stay there, either bouncing erratically up and down or sinking ominously off the screen as I watched, helpless.

But artefacts like this can be really helpful for understanding how your code works. In this case, ask yourself what our code has been instructed to do: if the centre is too close to the edge, as it is here, the code we wrote will multiply the velocity vector by -1, effectively reversing the direction of motion. However, thanks to friction, we won't quite regain the same height that we had previously, so once we fall off the screen, there's no way back.

A nice solution is to implement a 'back-tracking' to take place immediately after a ball goes off-screen, in addition to the reversal of the velocity. That will effectively put the ball back in its last 'good' position, and furnish it with a velocity which will take it further from the edge instead of closer.

```
12  move(){
13    this.applyFriction();
14    this.vel.add(this.acc);
15    this.pos.add(this.vel);
16    if (this.pos.x < 10 || this.pos.x > width - 10){
17      this.vel.x *= -1;
18      this.pos.add(this.vel);
19    }
20    if (this.pos.y < 10 || this.pos.y > height - 10){
21      this.vel.y *= -1;
22      this.pos.add(this.vel);
23    }
24  }
```

Adding a trail

While commenting out the *background* command in *draw* is a quick easy way to see some kind of trail, a better option is to build an array within the *Ball* class which keeps track of the last, say, 100 locations, and shows them on screen along with the ball. We'll need to *push* new positions into the list, and *shift* old positions out from the other end once the list gets too large.

```
2   constructor(x, y, vx, vy){
3     this.pos = createVector(x, y);
4     this.vel = createVector(vx, vy);
5     this.acc = createVector(0, 0.1);
6     this.trail = [];
7   }
```

First we'll need to make a class attribute: an empty array should do the trick. This should be done within *constructor*.

```
13  move(){
14    this.trail.push(this.pos.copy());
15    if (this.trail.length > 100){
16      this.trail.shift();
17    }
```

This code puts a copy of the current position vector in our *trail* array, and removes the earliest term if the trail exceeds a certain length.

```
30  display(){
31    for (let i=0; i<this.trail.length; i++){
32      circle(this.trail[i].x, this.trail[i].y, 20);
33    }
34    circle(this.pos.x, this.pos.y, 20);
35  }
36 }
```

Finally, in the *display* method, we draw a circle for each position stored in the trail before drawing the true object.

So far, this code seems to be turning a bouncing ball into a slinky, since the pen stroke is still drawn around the circle. We can fix that, and even make a few other aesthetic improvements.

Also, if you're getting bored of seeing the same behaviour over and over again, use *random* to set the initial speeds or positions when we first make the balls in *setup*.

Colour and size

In addition to using *noStroke()* to get rid of the line around our circle, we could add in some colour.

```
2   constructor(x, y, vx, vy){
3     this.r = random(255);
4     this.g = random(255);
5     this.b = random(255);
```

First, I add some code to the *constructor* so that each ball is a random colour.

```
33  display(){
34    noStroke();
35    for (let i=0; i<this.trail.length; i++){
36      let r = map(i, 0, this.trail.length, 255, this.r);
37      let g = map(i, 0, this.trail.length, 255, this.g);
38      let b = map(i, 0, this.trail.length, 255, this.b);
39      let size = map(i, 0, this.trail.length, 10, 20);
40      fill(r, g, b);
41      circle(this.trail[i].x, this.trail[i].y, size);
42    }
43    fill(color(this.r, this.g, this.b));
44    circle(this.pos.x, this.pos.y, 20);
45  }
```

Next, I use *map* to set the amount of colour (from white to the same as the ball) so that it depends on how far through the *trail* array we are.

I've also used *map* to do a similar thing with the size, so that circles get smaller the further along they are.

And now, if you *also* disable the drawing of *background* at the start of *draw*, you get a really neat effect:

And finally… freeze!

There's lots of potential for more modifications here, and in future we'll go into more depth with how to use the acceleration, velocity, position set-up to simulate attraction between objects, and using the mouse position to impart a force on objects. For now, the last thing I want to do is code a 'freeze' option, allowing the user, should they particularly like a certain pattern than appears, to capture it.

```
48  let n = 8;
49  let balls = [];
50  let freeze = false;
```

First we'll define a variable in the global scope that will keep track of whether the screen should be frozen or not at any given moment.

```
59  function draw() {
60    //background(220);
61    for (let i=0; i<balls.length; i++){
62      if (!freeze){balls[i].move()};
63      balls[i].display();
64    }
65  }
```

Then we should change what *draw* does depending on the state of that variable. Notice that the *if* function is all on one line, though I would only do it if it aids readability. The operator ! means *not*.

```
67  function mousePressed(){freeze = !freeze}
```

Finally, the built-in *mousePressed* function will run only when the mouse is pressed, and sets *freeze* to *true* if *false*, and *false* if *true*.