

Snake p5 Project

Introduction

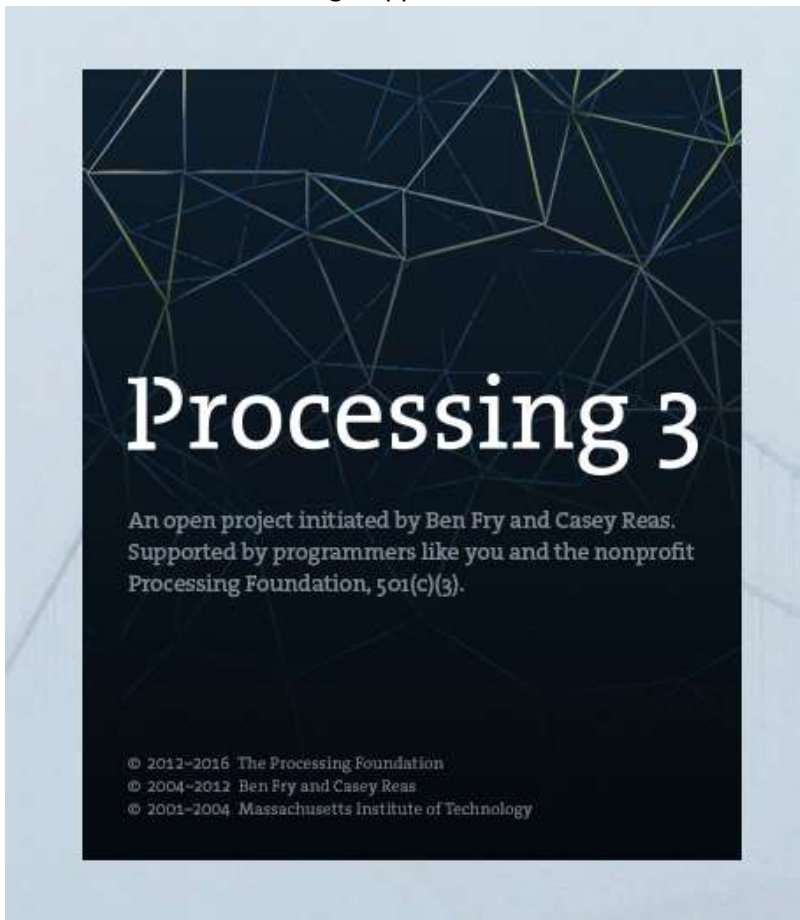
You are going to make a game called Snake ([https://en.wikipedia.org/wiki/Snake_\(video_game\)](https://en.wikipedia.org/wiki/Snake_(video_game))), you might not remember this game but your parents probably will because it used to be the only game most people could play on their mobile phones This is broadly based upon a coding challenge by Daniel Shiffman and is on YouTube here: <https://www.youtube.com/watch?v=AaGK-fj-BAM>.

Step 1: Getting started

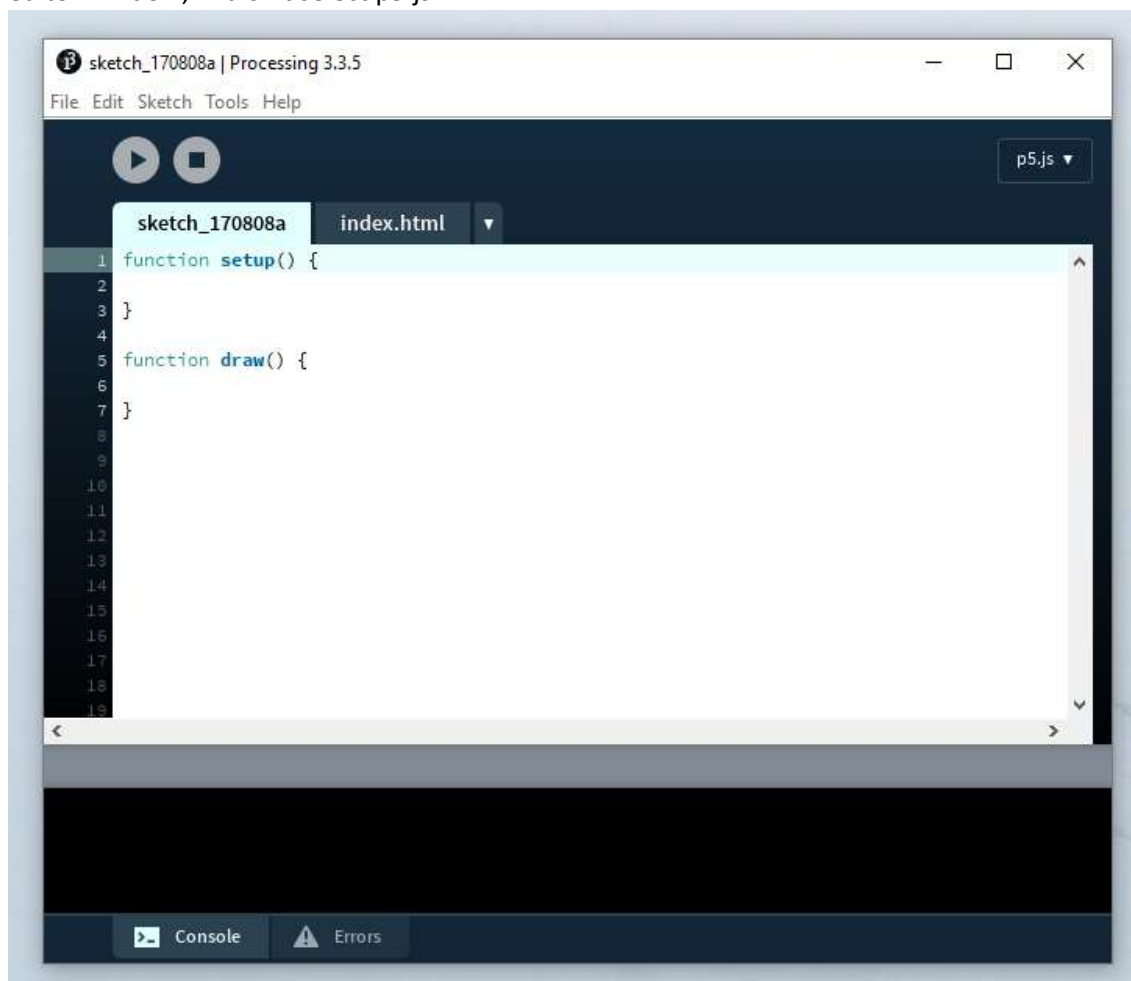
1. Open processing:



2. This will start the Processing 3 application:



3. Make sure your new project is a p5.js project by checking that p5.js is on the top right of the editor window, if it isn't select p5.js:



Explanation


p5.js is a JavaScript library that has a goal of making coding accessible for artists, designers, educators and – for our purposes – beginners. It is mostly used to help us draw on the web.

Step 2: Our playing field

When we start a new p5 project we are given two blocks of code (which are functions) in one file, and a blank HTML file where our work will be displayed when we're finished. One of the functions is called **setup** and the other is called **draw**.

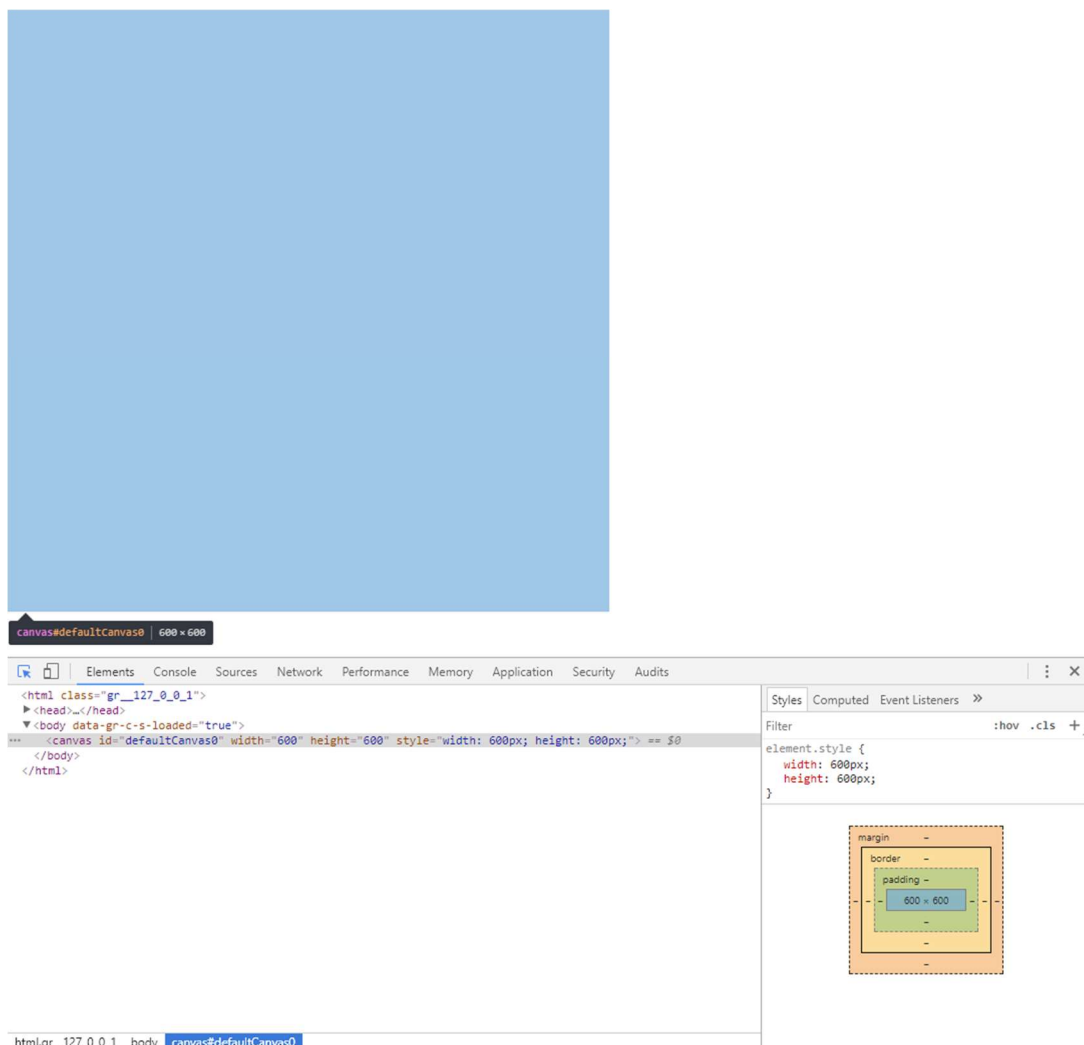
Neither of these functions takes any arguments but they are what we use to tell p5 what to do. We first need to make a playing field and we'll do this within the setup function by creating a canvas.

1. Add `createCanvas(600, 600);` to the setup function:



```
1 function setup() {  
2   createCanvas(600, 600);  
3 }  
4  
5 function draw() {  
6  
7 }  
8  
9
```

2. If you click the play button a new page will appear in your default browser, but there's nothing there, or is there? If we examine the page with the inspector we can see an invisible area:



3. Let's make the playing field visible by setting a background colour, add **background(50, 50, 100);** to your draw function.

```
1 function setup() {  
2   createCanvas(600, 600);  
3 }  
4  
5 function draw() {  
6   background(50, 50, 100);  
7 }  
8
```

Explanation

Colours in p5 can be a little confusing: we describe colours regarding how much red, green and blue they contain.

But instead of describing the amount of the colour from 1 – 10 or even 1 – 100, we describe them from 0 – 255. This is because computers used to have a limited number of colours they could display, they also had a small amount of memory they could use to describe colours so they used a different numbering system to the one we're used to.

Because we use the decimal system we're used to counting like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20... Computers aren't limited to this number base, in fact they prefer something based on multiples of 2. You could say that they're primarily interested in just two numbers: 0 and 1. This is called the binary number system and it's what computers understand best. So to count in binary you'd start with 0 and go on from there like this: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1111, 10000, 10001, 10010, 10011, 10100... There is a really good reference to learn more about counting in binary here: <http://www.wikihow.com/Count-in-Binary>

Humans like us find it difficult to understand binary so we've invented a compromise – after all, who'd know that saying, “ten-thousand and one hundred”, means 20? So, instead of using binary we use something called hexadecimal. But, I doubtless hear you say, how can we use 16 numbers when we're limited to 10 numbers (0 – 9), well, we borrow some letters!

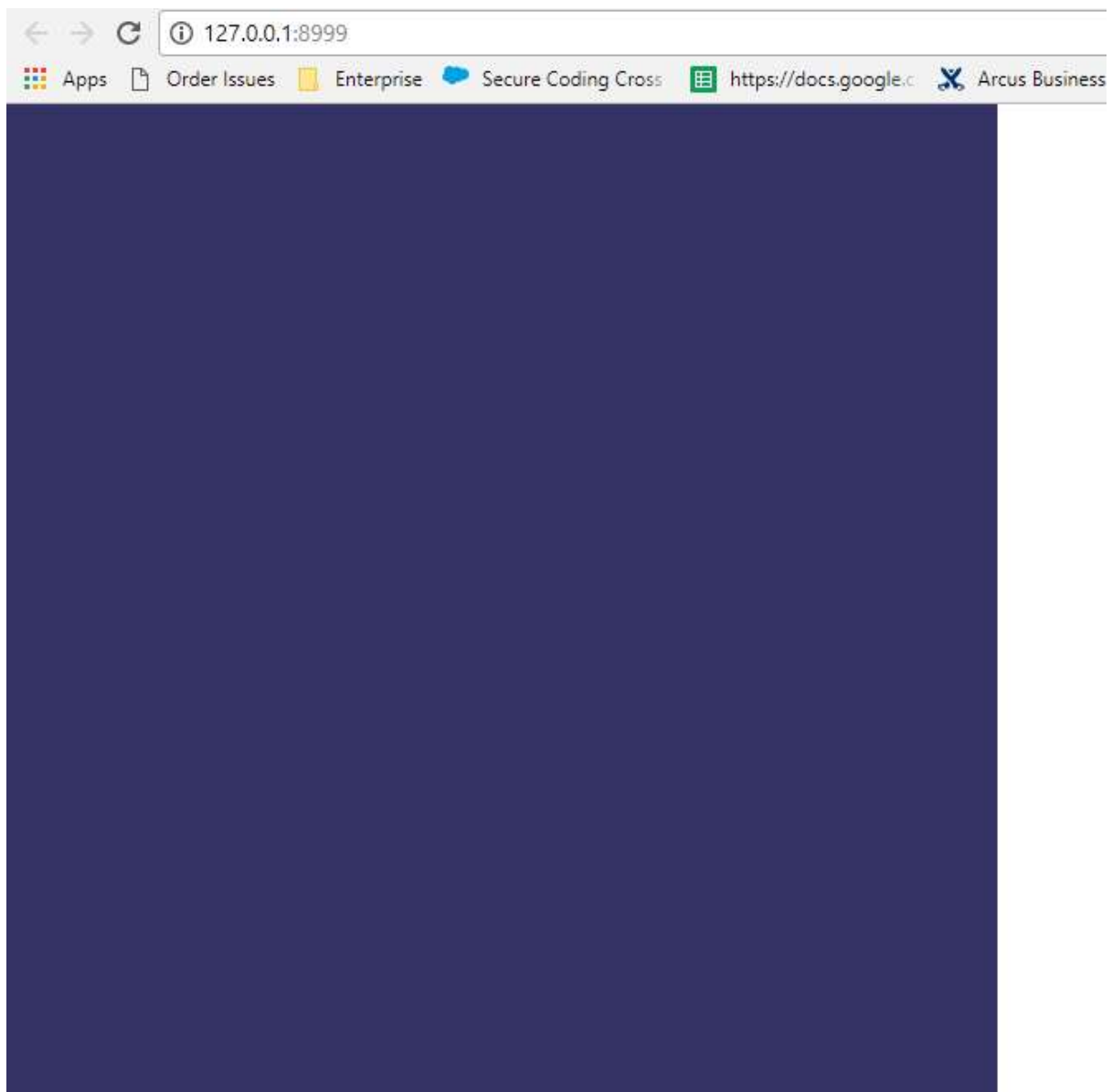
In hexadecimal we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...

This is how each number compares:

Decimal	Binary	Hexadecimal
0	00000000	0
1	00000001	1
2	00000010	2
3	00000011	3
4	00000100	4
5	00000101	5
6	00000110	6
7	00000111	7
8	00001000	8
9	00001001	9
10	00001010	A
11	00001011	B
12	00001100	C

13	00001101	D
14	00001110	E
15	00001111	F
16	00010000	10
17	00010001	11
18	00010010	12
19	00010011	13
20	00010100	14

Because it can also be confusing to say, “fourteen”, when we mean 20, many times we convert these hexadecimal numbers to decimal. This then is why we use such odd numbers when we describe colours in p5. Above, **background(50, 50, 100);** means we’re telling p5 to use a colour which uses 50/255 red, 50/255 green and 100/255 blue, like this:



There's probably lots more to say about web colours, this link might help:

https://en.wikipedia.org/wiki/Web_colors It's worth remembering that in the USA colour is spelt differently, just make sure you don't use the *color* spelling at school.

Another thing to bear in mind is that p5 can use proper hexadecimal values if we make them a string and ensure that we use the correct notation. This means that instead of writing **background(50, 50, 100)**; as we did, we could use **background("#323264")**; instead. There's lots more to know about colours in p5 and this link: <https://p5js.org/reference/#/p5/background> will tell you about the other ways of describing colours.

Challenge

Can you convert these numbers into different number bases (at least one number base has been given to you – simply work out what should go in the gaps)?

Decimal	Binary	Hexadecimal
	01101000	68
57		39
30	00011110	
113		71
	00011001	19
110		
		A2
	10101000	
149		
		BB

Step 3: Starting our snake

1. Add this code after your draw function:

```
8
9 class Snake {
10   constructor(){
11     this.x = 0;
12     this.y = 0;
13     this.xspeed = 1;
14     this.yspeed = 0;
15   }
16
17   update() {
18     this.x = this.x + this.xspeed;
19     this.y = this.y + this.yspeed;
20   }
21
22   show() {
23     fill(255);
24     rect(this.x, this.y, 10, 10);
25   }
26 }
27
```

Here we're creating a class which will represent our snake. It has an **x** and **y** coordinate as well as a horizontal and vertical speed (**xspeed** and **yspeed**). We've given it two methods (**update** and **show**) which will move our snake as well as showing it. Let's make a snake and

make it visible.

2. Making a snake involves creating a global variable for our program, which we'll call **s**. Then we need to make sure our global variable points to an instance of our snake class. To do this add this code to the top of your sketch code and alter the setup function as shown:

```
1 let s;  
2  
3 function setup() {  
4   createCanvas(600, 600);  
5   s = new Snake();  
6 }  
7
```

We're using **let** to create our variable as we want to assign our snake instance to it within the setup function.

3. Once our program knows about the snake we need to show it, add this code to your draw function:

```
8 function draw() {  
9   background("#323264");  
10  s.update();  
11  s.show()  
12 }  
13
```

This tells the snake called **s** to update and then show itself. If you press play now then you should see your snake starting on the top right of the square and moving to the right.

Step 4: Moving our snake

1. Add this code after your **draw** function:

```
17  
18 function keyPressed() {  
19   if (keyCode === UP_ARROW) {  
20     s.dir(0, -1);  
21   } else if (keyCode === DOWN_ARROW) {  
22     s.dir(0, 1);  
23   } else if (keyCode === RIGHT_ARROW) {  
24     s.dir(1, 0);  
25   } else if (keyCode === LEFT_ARROW) {  
26     s.dir(-1, 0);  
27   }  
28 }  
29  
30
```

2. Add this code to your Snake class:

```
24
25
26
27
28
29

dir(x, y) {
  this.xspeed = x;
  this.yspeed = y;
}
```

Explanation

Here we're listening to keys pressed on the keyboard and, if they are arrow keys, we're invoking a method of our Snake (called **dir**) which alters its speed along either the **x** or **y** axis. The first block of code says that if the up arrow is pressed then the speed along the **x** axis should be zero and the speed along the **y** axis should be -1 (which will make the snake move upwards). You should be able to tell what the other arrow keys do to the direction.

Step 5: Snake enhancements

Our snake can move, but it moves a little too smoothly! We need it to move within a grid, and we also need it to stop moving off the canvas. Because we know how large the canvas will be (**600** pixels wide by **600** pixels high) we can send these numbers to the snake class. Our snake is also quite small so we'll make it bigger, it might be best if we tell the snake how big it should be when we create it. We'll add all these numbers (the size of the snake, which we'll call **scale**; the **width** of the canvas and the **height** of the canvas) as variables which can be changed if we decide to alter the mechanics of the game later.

We'll also make the frame rate smaller. By default, it's 60 frames a second, we'll make it 10. This is because if it jumped 20 pixels each second it would move far too fast.

Our snake class is also getting a little too big, so we'll move it into its own file.

1. Change the main file so it looks like this:

```
1  let s;
2  const scale = 20;
3  const width = 600;
4  const height = 600;
5
6  function setup() {
7    createCanvas(width, height);
8    frameRate(10);
9    s = new Snake(scale, width, height);
10 }
11
```

Here we're setting our main variables at the top of the file, and then using them to create the canvas as well as the snake.

2. Create a new file called **Snake.js** and copy the snake class into it, then make the alterations shown here (we're changing its constructor as well as its **update** and **show** methods to make use of its new properties: **scale**, **canvasWidth** and **canvasHeight**):


```

1  class Snake {
2      constructor(s, w, h) {
3          this.x = 0;
4          this.y = 0;
5          this.xspeed = 1;
6          this.yspeed = 0;
7          this.scale = s;
8          this.canvasWidth = w;
9          this.canvasHeight = h;
10     }
11
12     update() {
13         this.x = this.x + this.xspeed * this.scale;
14         this.y = this.y + this.yspeed * this.scale;
15
16         this.x = constrain(this.x, 0, this.canvasWidth - this.scale);
17         this.y = constrain(this.y, 0, this.canvasHeight - this.scale);
18     }
19
20     show() {
21         fill(255);
22         rect(this.x, this.y, this.scale, this.scale);
23     }
24
25     dir(x, y) {
26         this.xspeed = x;
27         this.yspeed = y;
28     }
29
30 }
31
32

```

Explanation

We've made a few changes here. We've updated the constructor to take in three new values: the size of the snake, the width of the canvas and the height of the canvas.

Because we've got the size of the snake (which we're calling scale here) we can use that to make the snake jump from point to point rather than move slowly – we do this by multiplying the speed by the scale and adding it to the coordinate of the snake.

Because the snake now knows about the canvas we can also constrain its movements so that it won't go any further than the edge of the canvas.

If you press play now you should see the snake moving on the canvas in a jerky way, almost like it's moving along rows and columns.

Step 6: Making food

Our snake needs to eat, let's give it some food.

1. Add this code under the setup function:

```
13
14  const pickLocation = () => {
15      const cols = floor(width/scale);
16      const rows = floor(height/scale);
17      food = createVector(floor(random(cols)), floor(random(rows)));
18      food.mult(scale);
19  };
20
```

2. Add a **food** variable and alter the **setup** function to call our new **pickLocation** function.

```
1  let s;
2  let food;
3  const scale = 20;
4  const width = 600;
5  const height = 600;
6
7  function setup() {
8      createCanvas(width, height);
9      frameRate(10);
10     s = new Snake(scale, width, height);
11     pickLocation();
12 }
13
```

3. Finally, we need to draw the food. Change the **draw** function:

```
20
21  function draw() {
22      background(50, 50, 100);
23      s.update();
24      s.show();
25      fill(255, 0, 100);
26      rect(food.x, food.y, scale, scale);
27  }
28
```

Explanation

We're using a vector to hold the location of our food. Vectors can have an **x**, **y** and **z** property but we're only really interested in the **x** and **y**. If our snake moved in three dimensions we'd need a three-dimensional vector using **x**, **y** and **z** but it's limited to a two-dimensional board so we'll just have **x** and **y**.

You might remember from the last step that we noted that the snake moves as though moving across rows and columns, we'll make use of this now when we create our vector. The **cols** and **rows** constants in the **pickLocation** function are created by dividing the width or height by the scale of the snake. In our case $600/20 = 30$ so we have 30 rows and 30 columns. We'll assign a random number between 0 and 30 to the vectors **x** and **y**. Because our game board is 600 pixels on-a-side we'll then multiply the vector by the **scale** so that the **x** and **y** coordinates are a multiple of the **scale**, ensuring that the food is on the same coordinate system as the snake.

Finally, we need to **setup** a location for our food and then **draw** it.

Step 7: Eating food

Now we have food we need to let the snake eat it.

1. Add this method to our Snake class:

```
29
30     eat(pos) {
31         const d = dist(this.x, this.y, pos.x, pos.y);
32         return d === 0;
33     }
34
```

2. Change our **draw** function:

```
21     function draw() {
22         background(50, 50, 100);
23         if (s.eat(food)) {
24             pickLocation();
25         }
26         s.update();
27         s.show();
28         fill(255, 0, 100);
29         rect(food.x, food.y, scale, scale);
30     }
```

Explanation

The **eat** method on our Snake class now returns true or false depending upon whether the snake is at the same location of the food. It does this by calculating the distance between the snake head's **x** and **y** coordinates and the **x** and **y** coordinates of the food. If the distance is 0 it returns true, else it returns false.

If the food has been eaten by the snake, we need to add another random piece of food. We do this in the draw function.

Step 8: Digesting food

Now the snake can eat the food we need it to grow.

1. Update the Snake constructor:

```
1     class Snake {
2         constructor(s, w, h) {
3             this.x = 0;
4             this.y = 0;
5             this.xspeed = 1;
6             this.yspeed = 0;
7             this.scale = s;
8             this.canvasWidth = w;
9             this.canvasHeight = h;
10            this.total = 0;
11            this.tail = [];
12        }
13    }
```

Here we're giving our Snake a total number of segments in total and an empty array to hold

the positions of those segments.

2. We need to change the eat function to increase the total we've just created:

```
46     eat(pos) {
47         if(dist(this.x, this.y, pos.x, pos.y) === 0){
48             this.total++;
49             return true;
50         }else{
51             return false;
52         }
53     }
54
```

Instead of just returning true or false we increment the **total** by one if the snake has eaten.

3. Update the Snake **update** method:

```
13
14     update() {
15         for (let i = 0; i < this.tail.length - 1; i++) {
16             this.tail[i] = this.tail[i + 1];
17         }
18         if (this.total > 0) {
19             this.tail[this.total - 1] = createVector(this.x, this.y);
20         }
21
22         this.x = this.x + this.xspeed * this.scale;
23         this.y = this.y + this.yspeed * this.scale;
24
25         this.x = constrain(this.x, 0, this.canvasWidth - this.scale);
26         this.y = constrain(this.y, 0, this.canvasHeight - this.scale);
27     }
28
```

This ensures that the segments of the snake get updated properly when the snake moves. It also puts the current position of the snake at the end of the array. We'll go into this more in the explanation.

4. Now we have segments we need to **show** them:

```
29     show() {
30         fill(255);
31         this.tail.forEach(segment => {
32             rect(segment.x, segment.y, this.scale, this.scale);
33         });
34         rect(this.x, this.y, this.scale, this.scale);
35     }
36
```

Explanation

That's an awful lot of array manipulation going on, isn't it?

What we're doing is keeping a reference to the length of the snake and creating an array of each vector that the tail has. When we update the snake, we shift each element of the array to the left while keeping the array the same length. For instance, if we had an array like this `[1, 2, 3, 4]` then the for loop would result in an array like this: `[2, 3, 4, 4]`. We could have written it differently:

```

15 // for (let i = 0; i < this.tail.length - 1; i++) {
16 //     this.tail[i] = this.tail[i + 1];
17 // }
18 if(this.tail.length){
19     this.tail.splice(0, 1).push(this.tail[this.tail.length - 2]);
20 }

```

This does the same as we remove the first element of the array and then duplicate the last element at the end of the array.

We're also making use of our **total** and if the total is greater than zero we replace the end element in the array with the current position of the snake. Neat eh?

In the **show** function, we make sure we show all the segments of our snake.

Step 9: Killing our snake

In the original game, the snake lost its tail if it crossed over itself. We don't know why this is the rule of the game, it just is. We sort of know how to do this quite easily thanks to looking at the **eat** method. We check the distance between the current position of the snakes head and each segment, if they are the same then it's game over... Or, at least, the tail needs to go.

1. Add this method to the Snake:

```

55 death() {
56     this.tail.forEach(segment => {
57         if(dist(this.x, this.y, segment.x, segment.y) === 0){
58             this.total = 0;
59             this.tail = [];
60         }
61     });
62 }

```

2. Add its invocation to the **draw** function:

```

21 function draw() {
22     background(50, 50, 100);
23     if (s.eat(food)) {
24         pickLocation();
25     }
26     s.death();
27     s.update();
28     s.show();
29     fill(255, 0, 100);
30     rect(food.x, food.y, scale, scale);
31 }

```

Explanation

We need to call the **death** method on each **draw**. In it we iterate over each segment of the **tail** array as a segment; if the distance between the segment and the head of the snake is 0 then we truncate the snake by setting the **total** to 0 and reinitialising the **tail** array to an empty array.

Step 10: Final Code

This is the final code for our snake game. Hosted here: <https://github.com/annoyingmouse/p5Snake>

Snake.js

```
class Snake {
  constructor(s, w, h) {
    this.x = 0;
    this.y = 0;
    this.xspeed = 1;
    this.yspeed = 0;
    this.scale = s;
    this.canvasWidth = w;
    this.canvasHeight = h;
    this.total = 0;
    this.tail = [];
  }

  update() {
    if(this.tail.length){
      this.tail.splice(0, 1).push(this.tail[this.tail.length - 2]);
    }
    if (this.total > 0) {
      this.tail[this.total - 1] = createVector(this.x, this.y);
    }
    this.x = this.x + this.xspeed * this.scale;
    this.y = this.y + this.yspeed * this.scale;
    this.x = constrain(this.x, 0, this.canvasWidth - this.scale);
    this.y = constrain(this.y, 0, this.canvasHeight - this.scale);
  }

  show() {
    fill(255);
    this.tail.forEach(segment => {
      rect(segment.x, segment.y, this.scale, this.scale);
    });
    rect(this.x, this.y, this.scale, this.scale);
  }

  dir(x, y) {
    this.xspeed = x;
    this.yspeed = y;
  }

  eat(pos) {
    if(dist(this.x, this.y, pos.x, pos.y) === 0){
      this.total++;
      return true;
    }else{
      return false;
    }
  }

  death() {
    this.tail.forEach(segment => {
      if(dist(this.x, this.y, segment.x, segment.y) === 0){
        this.total = 0;
        this.tail = [];
      }
    });
  }
}
```

sketch.js

```
let s;
let food;
const scale = 20;
const width = 600;
const height = 600;

function setup() {
  createCanvas(width, height);
  frameRate(10);
  s = new Snake(scale, width, height);
  pickLocation();
}

const pickLocation = () => {
  const cols = floor(width/scale);
  const rows = floor(height/scale);
  food = createVector(floor(random(cols)), floor(random(rows)));
  food.mult(scale);
};

function draw() {
  background(50, 50, 100);
  if (s.eat(food)) {
    pickLocation();
  }
  s.death();
  s.update();
  s.show();
  fill(255, 0, 100);
  rect(food.x, food.y, scale, scale);
}

function keyPressed() {
  if (keyCode === UP_ARROW) {
    s.dir(0, -1);
  } else if (keyCode === DOWN_ARROW) {
    s.dir(0, 1);
  } else if (keyCode === RIGHT_ARROW) {
    s.dir(1, 0);
  } else if (keyCode === LEFT_ARROW) {
    s.dir(-1, 0);
  }
}
```

Step 11: Removing width and height

When we create our snake we pass through the width and height of the canvas, this is not required as width and height are global variables (i.e. available throughout the game). We can remove them from this **new Snake()** call and use them directly within the snake without having to add them via the **constructor**:

1. Change sketch so it looks like this:

```
1  let s;  
2  let food;  
3  const scale = 20;  
4  
5  function setup() {  
6    createCanvas(600, 600);  
7    frameRate(10);  
8    s = new Snake(scale);  
9    pickLocation();  
10 }  
11
```

2. Change Snake so its constructor looks like this:

```
1  class Snake {  
2    constructor(s) {  
3      this.x = 0;  
4      this.y = 0;  
5      this.xspeed = 1;  
6      this.yspeed = 0;  
7      this.scale = s;  
8      this.total = 0;  
9      this.tail = [];  
10 }  
11
```

And its update looks like this:

```
12  update() {  
13    for (let i = 0; i < this.tail.length - 1; i++) {  
14      this.tail[i] = this.tail[i + 1];  
15    }  
16  
17    if (this.total > 0) {  
18      this.tail[this.total - 1] = createVector(this.x, this.y);  
19    }  
20  
21    this.x = this.x + this.xspeed * this.scale;  
22    this.y = this.y + this.yspeed * this.scale;  
23  
24    this.x = constrain(this.x, 0, width - this.scale);  
25    this.y = constrain(this.y, 0, height - this.scale);  
26  }  
27
```

That's it, we're done!