Variable scope, closure

JavaScript is a very function-oriented language. It gives us a lot of freedom. A function can be created at any moment, passed as an argument to another function, and then called from a totally different place of code later.
We already know that a function can access variables outside of it ("outer" variables).
But what happens if outer variables change since a function is created? Will the function get newer values or the old ones?
And what if a function is passed along as an argument and called from another place of code, will it get access to outer variables at the new place?
Let's expand our knowledge to understand these scenarios and more complex ones.
We'll talk about `let/const` variables here
In JavaScript, there are 3 ways to declare a variable: `let`, `const` (the modern ones), and `var` (the remnant of the past).
In this article we'll use `let` variables in examples.
Variables, declared with `const`, behave the same, so this article is about `const` too.
The old `var` has some notable differences, they will be covered in the article The old "var".

Code blocks

If a variable is declared inside a code block `{...}`, it's only visible inside that block.
For example:

```
{
  // do some job with local variables that should not be seen outside

  let message = "Hello"; // only visible in this block

  alert(message); // Hello
}

alert(message); // Error: message is not defined
```

We can use this to isolate a piece of code that does its own task, with variables that only belong to it:

```
{
  // show message
  let message = "Hello";
  alert(message);
}

{
  // show another message
  let message = "Goodbye";
  alert(message);
}
```

There'd be an error without blocks
Please note, without separate blocks there would be an error, if we use `let` with the existing variable name:

```
// show message
let message = "Hello";
```

```
alert(message);

// show another message
let message = "Goodbye"; // Error: variable already declared
alert(message);
```
For `if`, `for`, `while` and so on, variables declared in `{...}` are also only visible inside:
```
if (true) {
  let phrase = "Hello!";

  alert(phrase); // Hello!
}

alert(phrase); // Error, no such variable!
```
Here, after `if` finishes, the `alert` below won't see the `phrase`, hence the error.
That's great, as it allows us to create block-local variables, specific to an `if` branch.
The similar thing holds true for `for` and `while` loops:
```
for (let i = 0; i < 3; i++) {
  // the variable i is only visible inside this for
  alert(i); // 0, then 1, then 2
}

alert(i); // Error, no such variable
```
Visually, `let i` is outside of `{...}`. But the `for` construct is special here: the variable, declared inside it, is considered a part of the block.

Nested functions

A function is called "nested" when it is created inside another function.
It is easily possible to do this with JavaScript.
We can use it to organize our code, like this:
```
function sayHiBye(firstName, lastName) {

  // helper nested function to use below
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```
Here the *nested* function `getFullName()` is made for convenience. It can access the outer variables and so can return the full name. Nested functions are quite common in JavaScript.
What's much more interesting, a nested function can be returned: either as a property of a new object or as a result by itself. It can then be used somewhere else. No matter where, it still has access to the same outer variables.
Below, `makeCounter` creates the "counter" function that returns the next number on each invocation:
```
function makeCounter() {
  let count = 0;
```

```
  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Despite being simple, slightly modified variants of that code have practical uses, for instance, as a random number generator to generate random values for automated tests.

How does this work? If we create multiple counters, will they be independent? What's going on with the variables here?

Understanding such things is great for the overall knowledge of JavaScript and beneficial for more complex scenarios. So let's go a bit in-depth.

Lexical Environment

Here be dragons!

The in-depth technical explanation lies ahead.

As far as I'd like to avoid low-level language details, any understanding without them would be lacking and incomplete, so get ready.

For clarity, the explanation is split into multiple steps.

Step 1. Variables

In JavaScript, every running function, code block `{...}`, and the script as a whole have an internal (hidden) associated object known as the *Lexical Environment*.

The Lexical Environment object consists of two parts:

Environment Record – an object that stores all local variables as its properties (and some other information like the value of `this`).

A reference to the *outer lexical environment*, the one associated with the outer code.

A "variable" is just a property of the special internal object, `Environment Record`. "To get or change a variable" means "to get or change a property of that object".

In this simple code without functions, there is only one Lexical Environment:

This is the so-called *global* Lexical Environment, associated with the whole script.

On the picture above, the rectangle means Environment Record (variable store) and the arrow means the outer reference. The global Lexical Environment has no outer reference, that's why the arrow points to `null`.

As the code starts executing and goes on, the Lexical Environment changes.

Here's a little bit longer code:

Rectangles on the right-hand side demonstrate how the global Lexical Environment changes during the execution:

When the script starts, the Lexical Environment is pre-populated with all declared variables.

Initially, they are in the "Uninitialized" state. That's a special internal state, it means that the engine knows about the variable, but it cannot be referenced until it has been declared with `let`. It's almost the same as if the variable didn't exist.

Then `let phrase` definition appears. There's no assignment yet, so its value is `undefined`. We can use the variable from this point forward.

`phrase` is assigned a value.
`phrase` changes the value.
Everything looks simple for now, right?
A variable is a property of a special internal object, associated with the currently executing block/function/script.
Working with variables is actually working with the properties of that object.
Lexical Environment is a specification object
"Lexical Environment" is a specification object: it only exists "theoretically" in the language specification to describe how things work. We can't get this object in our code and manipulate it directly.
JavaScript engines also may optimize it, discard variables that are unused to save memory and perform other internal tricks, as long as the visible behavior remains as described.

Step 2. Function Declarations

A function is also a value, like a variable.
The difference is that a Function Declaration is instantly fully initialized.
When a Lexical Environment is created, a Function Declaration immediately becomes a ready-to-use function (unlike `let`, that is unusable till the declaration).
That's why we can use a function, declared as Function Declaration, even before the declaration itself.
For example, here's the initial state of the global Lexical Environment when we add a function:
Naturally, this behavior only applies to Function Declarations, not Function Expressions where we assign a function to a variable, such as `let say = function(name)...`.

Step 3. Inner and outer Lexical Environment

When a function runs, at the beginning of the call, a new Lexical Environment is created automatically to store local variables and parameters of the call.
For instance, for `say("John")`, it looks like this (the execution is at the line, labelled with an arrow):
During the function call we have two Lexical Environments: the inner one (for the function call) and the outer one (global):
The inner Lexical Environment corresponds to the current execution of `say`. It has a single property: `name`, the function argument. We called `say("John")`, so the value of the `name` is `"John"`.
The outer Lexical Environment is the global Lexical Environment. It has the `phrase` variable and the function itself.
The inner Lexical Environment has a reference to the `outer` one.
When the code wants to access a variable – the inner Lexical Environment is searched first, then the outer one, then the more outer one and so on until the global one.
If a variable is not found anywhere, that's an error in strict mode (without `use strict`, an assignment to a non-existing variable creates a new global variable, for compatibility with old code).
In this example the search proceeds as follows:
For the `name` variable, the `alert` inside `say` finds it immediately in the inner Lexical Environment.
When it wants to access `phrase`, then there is no `phrase` locally, so it follows the reference to the outer Lexical Environment and finds it there.

Step 4. Returning a function

Let's return to the `makeCounter` example.

```
function makeCounter() {
```

```
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
```

At the beginning of each `makeCounter()` call, a new Lexical Environment object is created, to store variables for this `makeCounter` run.
So we have two nested Lexical Environments, just like in the example above:
What's different is that, during the execution of `makeCounter()`, a tiny nested function is created of only one line: `return count++`. We don't run it yet, only create.
All functions remember the Lexical Environment in which they were made. Technically, there's no magic here: all functions have the hidden property named `[[Environment]]`, that keeps the reference to the Lexical Environment where the function was created:
So, `counter.[[Environment]]` has the reference to `{count: 0}` Lexical Environment. That's how the function remembers where it was created, no matter where it's called.
The `[[Environment]]` reference is set once and forever at function creation time.
Later, when `counter()` is called, a new Lexical Environment is created for the call, and its outer Lexical Environment reference is taken from `counter.[[Environment]]`:
Now when the code inside `counter()` looks for `count` variable, it first searches its own Lexical Environment (empty, as there are no local variables there), then the Lexical Environment of the outer `makeCounter()` call, where it finds and changes it.
A variable is updated in the Lexical Environment where it lives.
Here's the state after the execution:
If we call `counter()` multiple times, the `count` variable will be increased to `2`, `3` and so on, at the same place.
Closure
There is a general programming term "closure", that developers generally should know.
A closure is a function that remembers its outer variables and can access them. In some languages, that's not possible, or a function should be written in a special way to make it happen. But as explained above, in JavaScript, all functions are naturally closures (there is only one exception, to be covered in The "new Function" syntax).
That is: they automatically remember where they were created using a hidden `[[Environment]]` property, and then their code can access outer variables.
When on an interview, a frontend developer gets a question about "what's a closure?", a valid answer would be a definition of the closure and an explanation that all functions in JavaScript are closures, and maybe a few more words about technical details: the `[[Environment]]` property and how Lexical Environments work.

Garbage collection

Usually, a Lexical Environment is removed from memory with all the variables after the function call finishes. That's because there are no references to it. As any JavaScript object, it's only kept in memory while it's reachable.
However, if there's a nested function that is still reachable after the end of a function, then it has `[[Environment]]` property that references the lexical environment.
In that case the Lexical Environment is still reachable even after the completion of the function, so it stays alive.
For example:

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // g.[[Environment]] stores a reference to the Lexical
Environment
// of the corresponding f() call
```

Please note that if `f()` is called many times, and resulting functions are saved, then all corresponding Lexical Environment objects will also be retained in memory. In the code below, all 3 of them:

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}

// 3 functions in array, every one of them links to Lexical Environment
// from the corresponding f() run
let arr = [f(), f(), f()];
```

A Lexical Environment object dies when it becomes unreachable (just like any other object). In other words, it exists only while there's at least one nested function referencing it.

In the code below, after the nested function is removed, its enclosing Lexical Environment (and hence the `value`) is cleaned from memory:

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // while g function exists, the value stays in memory

g = null; // ...and now the memory is cleaned up
```

Real-life optimizations

As we've seen, in theory while a function is alive, all outer variables are also retained.

But in practice, JavaScript engines try to optimize that. They analyze variable usage and if it's obvious from the code that an outer variable is not used – it is removed.

An important side effect in V8 (Chrome, Edge, Opera) is that such variable will become unavailable in debugging.

Try running the example below in Chrome with the Developer Tools open.

When it pauses, in the console type `alert(value)`.

```
function f() {
  let value = Math.random();

  function g() {
    debugger; // in console: type alert(value); No such variable!
```

```
  }

  return g;
}

let g = f();
g();
```

As you could see – there is no such variable! In theory, it should be accessible, but the engine optimized it out.

That may lead to funny (if not such time-consuming) debugging issues. One of them – we can see a same-named outer variable instead of the expected one:

```
let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // in console: type alert(value); Surprise!
  }

  return g;
}

let g = f();
g();
```

This feature of V8 is good to know. If you are debugging with Chrome/Edge/Opera, sooner or later you will meet it.

That is not a bug in the debugger, but rather a special feature of V8. Perhaps it will be changed sometime. You can always check for it by running the examples on this page.

Currying

Currying is an advanced technique of working with functions. It's used not only in JavaScript, but in other languages as well.

Currying is a transformation of functions that translates a function from callable as `f(a, b, c)` into callable as `f(a)(b)(c)`.

Currying doesn't call a function. It just transforms it.

Let's see an example first, to better understand what we're talking about, and then practical applications.

We'll create a helper function `curry(f)` that performs currying for a two-argument `f`. In other words, `curry(f)` for two-argument `f(a, b)` translates it into a function that runs as `f(a)(b)`:

```
function curry(f) { // curry(f) does the currying transform
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
```

```
}

// usage
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3
```

As you can see, the implementation is straightforward: it's just two wrappers.
The result of `curry(func)` is a wrapper `function(a)`.
When it is called like `curriedSum(1)`, the argument is saved in the Lexical Environment, and a new wrapper is returned `function(b)`.
Then this wrapper is called with `2` as an argument, and it passes the call to the original `sum`.
More advanced implementations of currying, such as _.curry from lodash library, return a wrapper that allows a function to be called both normally and partially:

```
function sum(a, b) {
  return a + b;
}

let curriedSum = _.curry(sum); // using _.curry from lodash library

alert( curriedSum(1, 2) ); // 3, still callable normally
alert( curriedSum(1)(2) ); // 3, called partially
```

Currying? What for?

To understand the benefits we need a worthy real-life example.
For instance, we have the logging function `log(date, importance, message)` that formats and outputs the information. In real projects such functions have many useful features like sending logs over the network, here we'll just use `alert`:

```
function log(date, importance, message) {
  alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}]
${message}`);
}
```

Let's curry it!

```
log = _.curry(log);
```

After that `log` works normally:

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

...But also works in the curried form:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Now we can easily make a convenience function for current logs:

```
// logNow will be the partial of log with fixed first argument
let logNow = log(new Date());

// use it
logNow("INFO", "message"); // [HH:mm] INFO message
```

Now `logNow` is `log` with fixed first argument, in other words "partially applied function" or "partial" for short.

We can go further and make a convenience function for current debug logs:

```
let debugNow = logNow("DEBUG");

debugNow("message"); // [HH:mm] DEBUG message
```

So:

We didn't lose anything after currying: `log` is still callable normally.

We can easily generate partial functions such as for today's logs.

Advanced curry implementation

In case you'd like to get in to the details, here's the "advanced" curry implementation for multi-argument functions that we could use above.

It's pretty short:

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  };

}
```

Usage examples:

```
function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, still callable normally
alert( curriedSum(1)(2,3) ); // 6, currying of 1st arg
alert( curriedSum(1)(2)(3) ); // 6, full currying
```

The new `curry` may look complicated, but it's actually easy to understand.

The result of `curry(func)` call is the wrapper `curried` that looks like this:

```
// func is the function to transform
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};
```

When we run it, there are two `if` execution branches:

If passed `args` count is the same or more than the original function has in its definition (`func.length`) , then just pass the call to it using `func.apply`.
Otherwise, get a partial: we don't call `func` just yet. Instead, another wrapper is returned, that will re-apply `curried` providing previous arguments together with the new ones.
Then, if we call it, again, we'll get either a new partial (if not enough arguments) or, finally, the result.

Fixed-length functions only

The currying requires the function to have a fixed number of arguments.
A function that uses rest parameters, such as `f(...args)`, can't be curried this way.

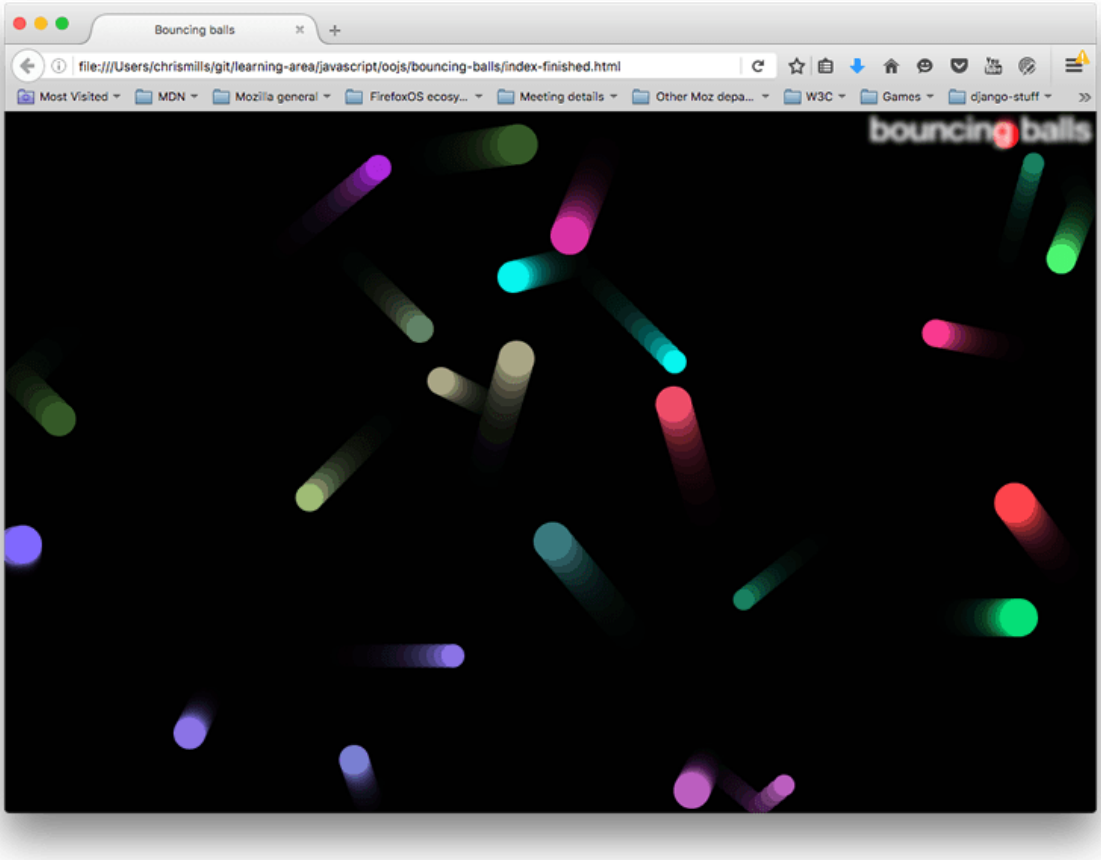
A little more than currying

By definition, currying should convert `sum(a, b, c)` into `sum(a)(b)(c)`.
But most implementations of currying in JavaScript are advanced, as described: they also keep the function callable in the multi-argument variant.

Object building practice

Let's bounce some balls

In this article we will write a classic "bouncing balls" demo, to show you how useful objects can be in JavaScript. Our little balls will bounce around on the screen, and change color when they touch each other. The finished example will look a little something like this:



This example will make use of the Canvas API for drawing the balls to the screen, and the requestAnimationFrame API for animating the whole display — you don't need to have any previous knowledge of these APIs, and we hope that by the time you've finished this article you'll be interested in exploring them more. Along the way, we'll make use of some nifty objects, and show you a couple of nice techniques like bouncing balls off walls, and checking whether they have hit each other (otherwise known as *collision detection*).

Getting started

To begin with, make local copies of our index.html, style.css, and main.js files. These contain the following, respectively:

1. A very simple HTML document featuring an h1 element, a <canvas> element to draw our balls on, and elements to apply our CSS and JavaScript to our HTML.
2. Some very simple styles, which mainly serve to style and position the <h1>, and get rid of any scrollbars or margin around the edge of the page (so that it looks nice and neat).
3. Some JavaScript that serves to set up the <canvas> element and provide a general function that we're going to use.

The first part of the script looks like so:

```
const canvas = document.querySelector("canvas");
const ctx = canvas.getContext("2d");
```

```
const width = (canvas.width = window.innerWidth);
const height = (canvas.height = window.innerHeight);
```
Copy to Clipboard

This script gets a reference to the <canvas> element, then calls the getContext() method on it to give us a context on which we can start to draw. The resulting constant (ctx) is the object that directly represents the drawing area of the canvas and allows us to draw 2D shapes on it.

Next, we set constants called width and height, and the width and height of the canvas element (represented by the canvas.width and canvas.height properties) to equal the width and height of the browser viewport (the area which the webpage appears on — this can be gotten from the Window.innerWidth and Window.innerHeight properties).

Note that we are chaining multiple assignments together, to get the variables all set quicker — this is perfectly OK.

Then we have two helper functions:

```
function random(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function randomRGB() {
  return `rgb(${random(0, 255)},${random(0, 255)},${random(0, 255)})`;
}
```
Copy to Clipboard

The random() function takes two numbers as arguments, and returns a random number in the range between the two. The randomRGB() function generates a random color represented as an rgb() string.

Modeling a ball in our program

Our program will feature lots of balls bouncing around the screen. Since these balls will all behave in the same way, it makes sense to represent them with an object. Let's start by adding the following class definition to the bottom of our code.

```
class Ball {
  constructor(x, y, velX, velY, color, size) {
    this.x = x;
    this.y = y;
    this.velX = velX;
    this.velY = velY;
    this.color = color;
    this.size = size;
  }
}
```
Copy to Clipboard

So far this class only contains a constructor, in which we can initialize the properties each ball needs in order to function in our program:

- x and y coordinates — the horizontal and vertical coordinates where the ball starts on the screen. This can range between 0 (top left hand corner) to the width and height of the browser viewport (bottom right-hand corner).
- horizontal and vertical velocity (velX and velY) — each ball is given a horizontal and vertical velocity; in real terms these values are regularly added to the x/y coordinate values when we animate the balls, to move them by this much on each frame.
- color — each ball gets a color.
- size — each ball gets a size — this is its radius, in pixels.

This handles the properties, but what about the methods? We want to get our balls to actually do something in our program.

Drawing the ball

First add the following draw() method to the Ball class:

```
draw() {
 ctx.beginPath();
 ctx.fillStyle = this.color;
 ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
 ctx.fill();
}
```
Copy to Clipboard

Using this function, we can tell the ball to draw itself onto the screen, by calling a series of members of the 2D canvas context we defined earlier (ctx). The context is like the paper, and now we want to command our pen to draw something on it:

- First, we use beginPath() to state that we want to draw a shape on the paper.
- Next, we use fillStyle to define what color we want the shape to be — we set it to our ball's color property.
- Next, we use the arc() method to trace an arc shape on the paper. Its parameters are:
    - The x and y position of the arc's center — we are specifying the ball's x and y properties.
    - The radius of the arc — in this case, the ball's size property.
    - The last two parameters specify the start and end number of degrees around the circle that the arc is drawn between. Here we specify 0 degrees, and 2 * PI, which is the equivalent of 360 degrees in radians (annoyingly, you have to specify this in radians). That gives us a complete circle. If you had specified only 1 * PI, you'd get a semi-circle (180 degrees).
- Last of all, we use the fill() method, which basically states "finish drawing the path we started with beginPath(), and fill the area it takes up with the color we specified earlier in fillStyle."

You can start testing your object out already.

1. Save the code so far, and load the HTML file in a browser.
2. Open the browser's JavaScript console, and then refresh the page so that the canvas size changes to the smaller visible viewport that remains when the console opens.
3. Type in the following to create a new ball instance:
4. const testBall = new Ball(50, 100, 4, 4, "blue", 10);

Copy to Clipboard

5. Try calling its members:
6. testBall.x;
7. testBall.size;
8. testBall.color;
9. testBall.draw();

Copy to Clipboard

10. When you enter the last line, you should see the ball draw itself somewhere on the canvas.

Updating the ball's data

We can draw the ball in position, but to actually move the ball, we need an update function of some kind. Add the following code inside the class definition for Ball:

```
update() {
 if ((this.x + this.size) >= width) {
  this.velX = -(this.velX);
 }

 if ((this.x - this.size) <= 0) {
  this.velX = -(this.velX);
 }

 if ((this.y + this.size) >= height) {
  this.velY = -(this.velY);
 }

 if ((this.y - this.size) <= 0) {
  this.velY = -(this.velY);
 }

 this.x += this.velX;
 this.y += this.velY;
}
```

Copy to Clipboard

The first four parts of the function check whether the ball has reached the edge of the canvas. If it has, we reverse the polarity of the relevant velocity to make the ball travel in the opposite direction. So for example, if the ball was traveling upwards (negative velY), then the vertical velocity is changed so that it starts to travel downwards instead (positive velY).

In the four cases, we are checking to see:

- if the x coordinate is greater than the width of the canvas (the ball is going off the right edge).
- if the x coordinate is smaller than 0 (the ball is going off the left edge).
- if the y coordinate is greater than the height of the canvas (the ball is going off the bottom edge).
- if the y coordinate is smaller than 0 (the ball is going off the top edge).

In each case, we include the size of the ball in the calculation because the x/y coordinates are in the center of the ball, but we want the edge of the ball to bounce off the perimeter — we don't want the ball to go halfway off the screen before it starts to bounce back.

The last two lines add the velX value to the x coordinate, and the velY value to the y coordinate — the ball is in effect moved each time this method is called.

This will do for now; let's get on with some animation!

Animating the ball

Now let's make this fun. We are now going to start adding balls to the canvas, and animating them.

First, we need to create somewhere to store all our balls and then populate it. The following will do this job — add it to the bottom of your code now:

```
const balls = [];

while (balls.length < 25) {
  const size = random(10, 20);
  const ball = new Ball(
    // ball position always drawn at least one ball width
    // away from the edge of the canvas, to avoid drawing errors
    random(0 + size, width - size),
    random(0 + size, height - size),
    random(-7, 7),
    random(-7, 7),
    randomRGB(),
    size
  );

  balls.push(ball);
}
```

Copy to Clipboard

The while loop creates a new instance of our Ball() using random values generated with our random() and randomRGB() functions, then push()es it onto the end of our balls array, but only while the number of balls in the array is less than 25. So when we have 25 balls in the array, no more balls will be pushed. You can try varying the number in balls.length < 25 to get more or fewer balls in the array. Depending on how much processing power your computer/browser has, specifying several thousand balls might slow down the animation rather a lot!

Next, add the following to the bottom of your code:

```
function loop() {
  ctx.fillStyle = "rgba(0, 0, 0, 0.25)";
  ctx.fillRect(0, 0, width, height);

  for (const ball of balls) {
    ball.draw();
    ball.update();
```

```
 }

 requestAnimationFrame(loop);
}
```
Copy to Clipboard

All programs that animate things generally involve an animation loop, which serves to update the information in the program and then render the resulting view on each frame of the animation; this is the basis for most games and other such programs. Our `loop()` function does the following:

- Sets the canvas fill color to semi-transparent black, then draws a rectangle of the color across the whole width and height of the canvas, using `fillRect()` (the four parameters provide a start coordinate, and a width and height for the rectangle drawn). This serves to cover up the previous frame's drawing before the next one is drawn. If you don't do this, you'll just see long snakes worming their way around the canvas instead of balls moving! The color of the fill is set to semi-transparent, `rgba(0,0,0,0.25)`, to allow the previous few frames to shine through slightly, producing the little trails behind the balls as they move. If you changed 0.25 to 1, you won't see them at all any more. Try varying this number to see the effect it has.
- Loops through all the balls in the `balls` array, and runs each ball's `draw()` and `update()` function to draw each one on the screen, then do the necessary updates to position and velocity in time for the next frame.
- Runs the function again using the `requestAnimationFrame()` method — when this method is repeatedly run and passed the same function name, it runs that function a set number of times per second to create a smooth animation. This is generally done recursively — which means that the function is calling itself every time it runs, so it runs over and over again.

Finally, add the following line to the bottom of your code — we need to call the function once to get the animation started.

```
loop();
```
Copy to Clipboard

That's it for the basics — try saving and refreshing to test your bouncing balls out!

Adding collision detection

Now for a bit of fun, let's add some collision detection to our program, so our balls know when they have hit another ball.

First, add the following method definition to your `Ball` class.

```
collisionDetect() {
  for (const ball of balls) {
    if (this !== ball) {
      const dx = this.x - ball.x;
      const dy = this.y - ball.y;
      const distance = Math.sqrt(dx * dx + dy * dy);

      if (distance < this.size + ball.size) {
```

```
      ball.color = this.color = randomRGB();
    }
   }
  }
}
```
Copy to Clipboard

This method is a little complex, so don't worry if you don't understand exactly how it works for now. An explanation follows:

- For each ball, we need to check every other ball to see if it has collided with the current ball. To do this, we start another `for...of` loop to loop through all the balls in the `balls[]` array.
- Immediately inside the for loop, we use an `if` statement to check whether the current ball being looped through is the same ball as the one we are currently checking. We don't want to check whether a ball has collided with itself! To do this, we check whether the current ball (i.e., the ball whose collisionDetect method is being invoked) is the same as the loop ball (i.e., the ball that is being referred to by the current iteration of the for loop in the collisionDetect method). We then use `!` to negate the check, so that the code inside the `if` statement only runs if they are **not** the same.
- We then use a common algorithm to check the collision of two circles. We are basically checking whether any of the two circle's areas overlap. This is explained further in 2D collision detection.
- If a collision is detected, the code inside the inner `if` statement is run. In this case, we only set the `color` property of both the circles to a new random color. We could have done something far more complex, like get the balls to bounce off each other realistically, but that would have been far more complex to implement. For such physics simulations, developers tend to use a games or physics libraries such as PhysicsJS, matter.js, Phaser, etc.

You also need to call this method in each frame of the animation. Update your `loop()` function to call `ball.collisionDetect()` after `ball.update()`:

```
function loop() {
 ctx.fillStyle = "rgba(0, 0, 0, 0.25)";
 ctx.fillRect(0, 0, width, height);

 for (const ball of balls) {
  ball.draw();
  ball.update();
  ball.collisionDetect();
 }

 requestAnimationFrame(loop);
}
```
Copy to Clipboard

Save and refresh the demo again, and you'll see your balls change color when they collide!

Note: If you have trouble getting this example to work, try comparing your JavaScript code against our finished version (also see it running live).

LEARN MORE

Reading

1. Read Getting Started with JSON Web Tokens (JWT)
2. Watch Develop, Debug, Learn? (Youtube, 19min)