var ishan = 324;

console.log(ishan);

console.log("hello world");

let array = [4, 3, 2, 1];

console.log(array.sort((a, b) => a - b));

console.log(array.reduce((total, i) => total + i));

console.log(array.filter(e => e >= 3));

console.log(array.map(e => e \* 2));

console.log(

  [

    { name: "ishan", age: 23 },

    { name: "kishan", age: 21 },

    { name: "nishan", age: 26 }

  ].find(student => student.age === 23)

);

array.forEach(element => {

  console.log(element);

});

console.log(array.some(e => e == 1));

console.log(array.some(e => e == 10));

console.log(array.every(e => e > 0));

console.log(array.every(e => e > 10));

console.log(array.includes(4));

function compare(a, b) {

if (a is less than b by some ordering criterion) {

return -1;

}

if (a is greater than b by the ordering criterion) {

return 1;

}

// a must be equal to b

return 0;

}

Array.includes vs Array.some

Jan 8, 2018 • flavio\_almeida • reading forecast: 4 minutes

Array in JavaScript is a powerful data structure. Functions like Array.map, Array.filterand Array.reducesolve a lot of everyday problems. However, there are two similar but essentially different functions that are part of your arsenal. In this article we will look at the functions Array.includesand Array.someeven catch and when to apply one or the other.

The problem

We have a simple array of numbers. We need to check if the number 15already exists in the list:

**const** numbers **=** [10, 11, 15, 20];

**const** searchNumber **=** 15;

**let** found **=** **false**;

**for**(number **of** numbers) {

**if**(number **===** 15) {

found **=** **true**;

**break**;

}

}

**if**(found) console.log(`${searchNumber} already exists!`);

Nothing exceptional here, except for the use of [for… of the](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/for...of) ES2015 (ES6) that can be replaced by a loop foror while. However, we can greatly simplify the code by using a feature introduced in ES2017 (ES7).

The Array.includes Function

In ES2017 (ES7) the function has been added Array.includes. Let's look at the previous code refactored to use the new function:

**const** numbers **=** [10, 11, 15, 20];

**const** searchNumber **=** 15;

**if**(numbers.includes(searchNumber))

console.log(`${searchNumber} already exists!`);

*Refactoring is altering the structure of code without changing its final behavior.*

The function Array.includesreturns trueonly if the element exists. Much leaner, no? However, this simplicity can lead the developer to error.

Unexpected Result

Let's repeat the search example with Array.includes, but this time searching a list of **objects** representing books:

**const** books **=** [

{

name: 'Cangaceiro JavaScript',

author: 'Flávio Almeida',

isbn: '9788594188014'

},

{

name: 'MEAN',

author: 'Flávio Almeida',

isbn: '9788555190476'

}

];

**const** searchBook **=** {

name: 'Cangaceiro JavaScript',

author: 'Flávio Almeida',

isbn: '9788594188014'

};

**if**(books.includes(searchBook))

console.log(`${JSON.stringify(searchBook)} already exists!`);

As much as the function Array.includesworked in the previous example, it seems unable to find our book. What is the reason for this erratic behavior?

Comparisons between types

Internally, it Array.includesmakes a comparison through ===. However, comparisons with ===or ==only work as expected (compare value) with the types:

* Boolean
* Null
* Undefined
* Number
* String
* Symbol (ES2015)

When we compare variables that reference the type object, the interpreter will test whether they point to the same object in memory. Let's look at this maxim in a smaller scope example:

**const** object1 **=** { name: 'Flávio'};

**const** object2 **=** { name: 'Flávio'};

*// false*

console.log(object1 **==** object2);

*// false*

console.log(object1 **===** object2);

However, the result will be true if we do:

**const** object1 **=** { name: 'Flávio'};

*// a variável de referência object2*

*// passou a apontar para o mesmo objeto*

*// referenciado por object1*

**const** object2 **=** object1;

*// true*

console.log(object1 **==** object2);

*// true*

console.log(object1 **===** object2);

In this context, we can Array.includesreturn trueas follows:

**const** books **=** [

{

name: 'Cangaceiro JavaScript',

author: 'Flávio Almeida',

isbn: '9788594188014'

},

{

name: 'MEAN',

author: 'Flávio Almeida',

isbn: '9788555190476'

}

];

*// A variável searchBook agora*

*// aponta para o mesmo objeto em books[0]*

**const** searchBook **=** books[0];

**if**(books.includes(searchBook))

console.log(`${JSON.stringify(searchBook)} already exists!`);

As much as this solution works, it makes no sense to use it when searching for an object that we do not know its position, let alone if it exists. In this situation we can turn to the good old Array.someES5 veteran.

Using the humble Array.some

Let's look at the previous code using the function Array.some:

**const** books **=** [

{

name: 'Cangaceiro JavaScript',

author: 'Flávio Almeida',

isbn: '9788594188014'

},

{

name: 'MEAN',

author: 'Flávio Almeida',

isbn: '9788555190476'

}

];

**const** searchBook **=** {

name: 'Cangaceiro JavaScript',

author: 'Flávio Almeida',

isbn: '9788594188014'

};

**if**(books.some(book **=>** book.isbn **==** searchBook.isbn))

console.log(`${JSON.stringify(searchBook)} already exists!`);

The function Array.someiterates through each element of the array by applying comparison logic. It will abort the iteration immediately as soon as it finds the first item that returns truein the comparison. Your return will be trueif there is any element that fits the criteria used.

The criterion used was to compare the ownership isbnof objects. Because the property stores one string, comparisons with ==or ===use the primitive value and not the reference.

We can still compare all properties of the object with the following trick:

*// código anterior omitido*

**const** searchBook **=** {

name: 'Cangaceiro JavaScript',

author: 'Flávio Almeida',

isbn: '9788594188014'

};

**const** bookAsJSON **=** JSON.stringify(searchBook);

**if**(books.some(book **=>** JSON.stringify(book) **==** bookAsJSON))

console.log(`${bookAsJSON} already exists!`);

Through JSON.stringifyconverting a JavaScript object into a JSON that is nothing more than the textual representation of an object. Being a representation through one string, we can use ===or ==that the comparison will use its value.

One of the great advantages of Array.somebeing able to define a more refined comparison logic.

Conclusion

Modern features do not always solve everyday problems. It is important to know the problem and also the resource used to solve it so we have no surprises.

Is that you? Have you ever been surprised by Array.includes? Have you ever used it Array.someto solve a problem? Leave your opinion.

Practise2

 <script>

      //console features-start

      /////////////////////////////////////////////////

      console.log("%c can add css to console", "color:red;font-weight:bold;");

      // view data clear in object array

      console.table([

        { name: "ishan", age: 12 },

        { name: "nishan", age: 12 },

        { name: "hishan", age: 12 }

      ]);

      //track time

      console.time("loop1");

      let i = 0;

      while (i < 100000) {

        i++;

      }

      console.timeEnd("loop1");

      //trace

      function trace() {

        console.trace("trace method called");

      }

      trace();

      trace();

      trace();

 /////////////////////////////////////////////////

      //console features -end

      //template literals-start

      /////////////////////////////////////////////////

      let age = 23;

      let height = 5.7;

      console.log("ishan age is " + age + " height is " + height);

      console.log(`ishan age is  ${age} height is ${height}`);

      console.log();

      function templateMethod(strArray, age) {

        console.log(strArray);

        return `${age} is ${strArray[0]} users age job is ${strArray[1]}`;

      }

      const str = templateMethod`ishan ${12} engineer`;

      console.log(str);

      /////////////////////////////////////////////////

      //template literals-end

      //spread syntax start ...

      /////////////////////////////////////////////////

      //objects

      const koththu = { name: "cheese koththu", shop: "koththu labs" };

      const description = { price: 500, meat: "chicken" };

      const koththuFullInfo = { ...koththu, ...description };

      console.log(koththuFullInfo);

      //arrays

      const lang1 = ["C++", "C", "B"];

      const lang2 = ["Java", "Javascript", "Google go"];

      const langs = [...lang2, ...lang1];

      console.log(langs);

      /////////////////////////////////////////////////

      //spread syntax end ...

//destructuring-start

      /////////////////////////////////////////////////

      let user = { name: "ishan", age: 23, occupation: "software engineer" };

      ishan(user);

      console.log("");

      ishanDestructure(user);

      console.log();

      ishanDestructureInBody(user);

      function ishan(userObject) {

        //without destructuring

        console.log(userObject.name);

        console.log(userObject.age);

        console.log(userObject.occupation);

      }

      function ishanDestructure({ name, age, occupation }) {

        //destructure inside method argument

        console.log(name);

        console.log(age);

        console.log(occupation);

      }

      function ishanDestructureInBody(userObject) {

        //with destructure inside body

        const { name, age, occupation } = userObject;

        console.log(name);

        console.log(age);

        console.log(occupation);

      }

      /////////////////////////////////////////////////

      //destructuring-end

 //async await-start

      /////////////////////////////////////////////////

      const random = () => {

        return Promise.resolve(Math.random());

      };

      //problem

      const sumRandomNumbersBad = () => {

        let firstNo;

        let secondNo;

        let thrirdNo;

        return random()

          .then(result1 => {

            firstNo = result1;

            return random();

          })

          .then(result2 => {

            secondNo = result2;

            return random();

          })

          .then(result3 => {

            thrirdNo = result3;

            return firstNo + secondNo + thrirdNo;

          });

      };

      sumRandomNumbersBad().then(e => {

        console.log(e);

      });

      //solution

      const sumRandomNumbers = async () => {

        let firstNo = await random();

        let secondNo = await random();

        let thrirdNo = await random();

        return firstNo + secondNo + thrirdNo;

      };

      sumRandomNumbers().then(e => {

        console.log(e);

      });

      /////////////////////////////////////////////////

      //async await-end

  //repeat method

      console.log("\*".repeat(23));

      //Array making methods

      console.log(Array.from(Array(5).keys()));

      console.log([...Array(5).keys()]);

      //Array clone method

      let array1 = [1, 2, 3];

      let array2 = array1;

      let array3 = [...array1]; //cloned

      console.log(array1 === array2);

      console.log(array1 === array3);

Parse strings in javascript

# Converting strings to numbers with vanilla JavaScript

In JavaScript, you can represent a number is an actual number (ex. 42), or as a string (ex. '42').

If you were to use a strict comparison to compare the two, it would fail because they’re two different types of objects.

var num1 = 42;

var num2 = '42';

if (num1 === num2) {

console.log(true);

} else {

console.log(false);

}

// Will log `false`

Today, let’s look at three different ways to convert a string into a number.

## parseInt()

The parseInt() method converts a string into an integer (a whole number).

It accepts two arguments. The first argument is the string to convert. The second argument is called the radix. This is the base number used in mathematical systems. For our use, it should always be 10.

var text = '42px';

var integer = parseInt(text, 10);

// returns 42

## parseFloat()

The parseFloat() method converts a string into a point number (a number with decimal points). You can even pass in strings with random text in them.

var text = '3.14someRandomStuff';

var pointNum = parseFloat(text);

// returns 3.14

## Number()

The Number() method converts a string to a number.

Sometimes it’s an integer. Other times it’s a point number. And if you pass in a string with random text in it, you’ll get NaN, an acronym for “Not a Number.”

As a result of this inconsistency, it’s a less safe choice than parseInt() and parseFloat(). If you know the format of the number you’d like, use those instead. If you want the string to fail with NaN if it has other characters in it, Number() may actually be a better choice.

// Convert strings

Number('123'); // returns 123

Number('12.3'); // returns 12.3

Number('3.14someRandomStuff'); // returns NaN

Number('42px'); // returns NaN

Self Invocation methods

(function ishan() {

  console.log("hi i am ishan");

})();

Object values- get all values in object as

a array

//GET ALL VALUES IN A OBJECT

const object = { name: "ishan", age: 1221, job: "softeare Engineer" };

console.log(Object.values(object));

convert Array in to a single String

//GET VALUES IN A ARRAY AS A STRING

const array = ["banana", "cake", "pineapple"];

console.log(array.join());

console.log(array.join(""));

console.log(array.join(" and "));

console.log(Object.values({ name: "ishan", age: 123 }).join("+"));

***output***

banana,cake,pineapple

bananacakepineapple

banana and cake and pineapple

ishan+123

Get object entries as a array (as a key and value pair)

//get object entries as a array (as a key and value pair)

const arrayE = { name: "ishan", age: 112, sex: "male" };

console.log(Object.entries(arrayE));

//output

//[ [ 'name', 'ishan' ], [ 'age', 112 ], [ 'sex', 'male' ] ]

Slice Trouble

## Description

slice() extracts the text from one string and returns a new string. Changes to the text in one string do not affect the other string.

slice() extracts up to but not including endIndex. str.slice(1, 4) extracts the second character through the fourth character (characters indexed 1, 2, and 3).

As an example, str.slice(2, -1) extracts the third character through the second to last character in the string.

0 1 2 3 4 5

I s h a n

I s h a n

-5 -4 -3 -2 -1

Single argument nam

console.log("ishan".slice(3)); => “an”

console.log("ishan".slice(-3)); => ”han”

### **Using slice() to create a new string**

The following example uses slice() to create a new string.

var str1 = 'The morning is upon us.', // the length of str1 is 23.

str2 = str1.slice(1, 8),

str3 = str1.slice(4, -2),

str4 = str1.slice(12),

str5 = str1.slice(30);

console.log(str2); // OUTPUT: he morn

console.log(str3); // OUTPUT: morning is upon u

console.log(str4); // OUTPUT: is upon us.

console.log(str5); // OUTPUT: ""

### **Using slice() with negative indexes**

The following example uses slice() with negative indexes.

var str = 'The morning is upon us.';

str.slice(-3); // returns 'us.'

str.slice(-3, -1); // returns 'us'

str.slice(0, -1); // returns 'The morning is upon us'

This example counts backwards from the end of the string by 11 to find the start index and forwards from the start of the string by 16 to find the end index.

console.log(str.slice(-11, 16)) // => "is u";

Here it counts forwards from the start by 11 to find the start index and backwards from the end by 7 to find the end index.

console.log(str.slice(11, -7)) // => " is u";

These arguments count backwards from the end by 5 to find the start index and backwards from the end by 1 to find the end index.

console.log(str.slice(-5, -1)) // => "n us";

* use anfn in vs code shortcut

**This in javascript**

## How *this* behaves inside arrow functions

One of the most relevant, if not the most relevant difference between classic functions and arrow functions is how the this works. This difference is the main reason why in some cases we cannot use arrow functions, as we will see soon. Before highlighting the differences, **let's recap how this works when it is used in standard functions**. The first thing to remember is that, the value of this is determined by how the function itself is called, let's see some examples.

### The default: *this* is a reference to the global scope

When this is used inside a standalone function, and we are not working in strict mode, it is references the global scope, which is the window object on a browser environment, or the global object in Node.js. In the same situation, but in strict mode, this will be undefined and we will receive an error:

var i = 20; // Here we used var instead of let because the latter doesn't create a property on the global scope.

function foo() {

console.log(this.i);

}

// Non-strict mode

foo()

20

// Strict mode

foo()

TypeError: Cannot read property 'i' of undefined

### Implicit binding

When a standard function is referenced inside an object, and that function is called with that object as a context, using the dot notation, this becomes a reference to that object. This is what we call implicit binding:

function foo() {

console.log(this.i);

}

let object = {

i: 20,

foo: foo // The foo property is a reference to the foo function

}

object.foo() // this is a reference to object, so this.i is object.i

20

### Explicit binding

We say that we are using an explicit binding when we are explicitly declaring what this should reference. It can be accomplished by using the call, apply or bind methods of a function (which in Javascript is itself a first-class object. Remember the first case we mentioned above, when the default binding applies:

var i = 20;

function foo() {

console.log(this.i);

}

const object = {

i: 100

}

foo() // This will output 20 or generate a TypeError in strict mode.

// If we explicitly set this to be a reference to object the things changes.

// call and apply execute the function immediately with the new context:

foo.call(object) // Output is 100

foo.apply(object) // Output is 100

// bind instead, returns a new function with the specified context.

let boundFoo = foo.bind(object)

boundFoo() // Output is 100

The are some differences between call, apply and bind: the relevant is that the latter returns a new function bound to the specified context, while with the other two, the function, bound to the specified context, is executed immediately. There are other differences, but we will not see them here. The important thing is to understand how explicitly binding works.

### How arrow functions are different in *this* regard?

In all the cases and examples above, we saw how, when using standard functions, the value of this depends on how the function is called. Arrow functions, instead, use the lexical this: they don't have their own this, but always use the this from their enclosing scope. A typical example where this could produce unexpected effects is on event listeners. Suppose we have a button with id "button1", and we want to change its text when it is clicked:

<button id="button1" type="button">Click me!</button>

// The event listener with a standard function as a callback

document.getElementById('button1').addEventListener('click', function() {

this.innerText = "Clicked!";

})

The code works perfectly, and once the button is clicked, its text changes as expected. What if we use an arrow function in this case? Suppose we write it like this:

document.getElementById('button1').addEventListener('click', () => this.innerText = "Clicked!"; )

The code above doesn't work, why? Easy: because, as we said before, while in the first example, this inside the standard callback function references the object on which the event occurs (the button), when we use the arrow function this is inherited from the parent scope, which in this case is the window object. For the sake of completeness, we should say that the example above could be easily fixed to work with an arrow function:

document.getElementById('button1').addEventListener('click', event => event.target.innerText = "Clicked!"; )

This time the code works because we didn't use this to reference the button, but we let our function accept one argument, which is event. In the function body we used event.target to reference the object which dispatched the event.

For the same reason we mentioned above, arrow functions cannot be used as object methods or prototype methods:

// Arrow functions don't work as object methods...

const object1 = {

i: 1000,

foo: () => console.log(`the value of i is ${this.i}`)

}

object1.foo()

the value of i is undefined

// ...and they don't not work as prototype methods.

const Person = function(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.introduce = () => console.log(`My name is ${this.name} and I am ${this.age} years old`);

const jack = new Person('Jack', 100);

jack.name

'Jack'

jack.age

100

jack.introduce()

My name is undefined and I am undefined years old

**Stringify replacer**

function replacer(key, value) {

  // Filtering out properties

  if (typeof value === 'string') {

    return undefined;

  }

  return value;

}

var foo = {foundation: 'Mozilla', model: 'box', week: 45, transport: 'car', month: 7};

JSON.stringify(foo, replacer);

// '{"week":45,"month":7}'

//or

JSON.stringify(foo, ['week', 'month']);

// '{"week":45,"month":7}', only keep "week" and "month" properties

**hasOwnProperty**

const object1 = new Object();

object1.property1 = 42;

console.log(object1.hasOwnProperty('property1'));

// expected output: true

console.log(object1.hasOwnProperty('toString'));

// expected output: false

console.log(object1.hasOwnProperty('hasOwnProperty'));

// expected output: false

**Get Unique Values of an Array**

Getting an [array of unique values](https://davidwalsh.name/array-unique) is probably easier than you think:

var j = [...new Set([1, 2, 3, 3])]

>> [1, 2, 3]

I love the mixture of rest expression and Set!

**Array and Boolean**

Ever need to [filter falsy values](https://davidwalsh.name/array-boolean) (0, undefined, null, false, etc.) out of an array? You may not have known this trick:

myArray

.map(item => {

// ...

})

// Get rid of bad values

.filter(Boolean);

Just pass Boolean and all those falsy value go away!

**Create Empty Objects**

Sure you can create an object that seems empty with {}, but that object still has a \_\_proto\_\_ and the usual hasOwnProperty and other object methods. There is a way, however, to [create a pure "dictionary" object](https://davidwalsh.name/object-create-null):

let dict = Object.create(null);

// dict.\_\_proto\_\_ === "undefined"

// No object properties exist until you add them

There are absolutely no keys or methods on that object that you don't put there!

**Merge Objects**

The need to [merge multiple objects](https://davidwalsh.name/merge-objects) in JavaScript has been around forever, especially as we started creating classes and widgets with options:

const person = { name: 'David Walsh', gender: 'Male' };

const tools = { computer: 'Mac', editor: 'Atom' };

const attributes = { handsomeness: 'Extreme', hair: 'Brown', eyes: 'Blue' };

const summary = {...person, ...tools, ...attributes};

/\*

Object {

"computer": "Mac",

"editor": "Atom",

"eyes": "Blue",

"gender": "Male",

"hair": "Brown",

"handsomeness": "Extreme",

"name": "David Walsh",

}

\*/

Those three dots made the task so much easier!

**Require Function Parameters**

Being able to set default values for function arguments was an awesome addition to JavaScript, but check out this trick for [requiring values](https://davidwalsh.name/javascript-function-parameters) be passed for a given argument:

const isRequired = () => { throw new Error('param is required'); };

const hello = (name = isRequired()) => { console.log(`hello ${name}`) };

// This will throw an error because no name is provided

hello();

// This will also throw an error

hello(undefined);

// These are good!

hello(null);

hello('David');

That's some next level validation and JavaScript usage!

**Destructuring Aliases**

[Destructuring](https://davidwalsh.name/destructuring-alias) is a very welcomed addition to JavaScript but sometimes we'd prefer to refer to those properties by another name, so we can take advantage of aliases:

const obj = { x: 1 };

// Grabs obj.x as { x }

const { x } = obj;

// Grabs obj.x as { otherName }

const { x: otherName } = obj;

Useful for avoiding naming conflicts with existing variables!

**Get Query String Parameters**

For years we wrote gross regular expressions to get query string values but those days are gone -- enter the amazing [URLSearchParams](https://davidwalsh.name/query-string-javascript) API:

// Assuming "?post=1234&action=edit"

var urlParams = new URLSearchParams(window.location.search);

console.log(urlParams.has('post')); // true

console.log(urlParams.get('action')); // "edit"

console.log(urlParams.getAll('action')); // ["edit"]

console.log(urlParams.toString()); // "?post=1234&action=edit"

console.log(urlParams.append('active', '1')); // "?post=1234&action=edit&active=1"

Much easier than we used to fight with!

**Destructuring**

**Swap**



In the following example, **swapping**has done using another variable called "temp". Therefore the code got lengthier.

[Live Demo](http://tpcg.io/j8Qyg6)

<html>

<body>

<script>

var a = "Sachin";

var b = "Tendulkar";

document.write("Before swapping-"+ " "+ a + " " +b);

var tmp = a;

a = b;

b = tmp;

document.write("</br>");

document.write("After swapping-"+ " " + a + " " +b);

</script>

</body>

</html>

### Output

Before swapping- Sachin Tendulkar

After swapping- Tendulkar Sachin

The task of **swapping**has become easier because of **destructuring**. Here we don't need to use another variable and even the code is not lengthy.

### Example-2

In the following example, no **third variable** is used and the **swapping**has done with **destructuring**. Here the code is much smaller than the above code.

[Live Demo](http://tpcg.io/Fte96T)

<html>

<body>

<script>

  var a = "Sachin";

var b = "Tendulkar";

document.write("Before swapping-"+ " "+ a + " " +b);

[a,b] = [b,a];

document.write("</br>");

document.write("After swapping-"+ " " + a + " " +b);

</script>

</body>

</html>

### Output

Before swapping- Sachin Tendulkar

After swapping- Tendulkar Sachin

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

**Syntax**

let a, b, rest;

[a, b] = [10, 20];

console.log(a); // 10

console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(a); // 10

console.log(b); // 20

console.log(rest); // [30, 40, 50]

({ a, b } = { a: 10, b: 20 });

console.log(a); // 10

console.log(b); // 20

// Stage 4(finished) proposal

({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40});

console.log(a); // 10

console.log(b); // 20

console.log(rest); // {c: 30, d: 40}

**Description**

The object and array literal expressions provide an easy way to create *ad hoc* packages of data.

const x = [1, 2, 3, 4, 5];

The destructuring assignment uses similar syntax, but on the left-hand side of the assignment to define what values to unpack from the sourced variable.

const x = [1, 2, 3, 4, 5];

const [y, z] = x;

console.log(y); // 1

console.log(z); // 2

This capability is similar to features present in languages such as Perl and Python.

**Array destructuring**

Basic variable assignment

const foo = ['one', 'two', 'three'];

const [red, yellow, green] = foo;

console.log(red); // "one"

console.log(yellow); // "two"

console.log(green); // "three"

Assignment separate from declaration

A variable can be assigned its value via destructuring separate from the variable's declaration.

let a, b;

[a, b] = [1, 2];

console.log(a); // 1

console.log(b); // 2

Default values

A variable can be assigned a default, in the case that the value unpacked from the array is undefined.

let a, b;

[a=5, b=7] = [1];

console.log(a); // 1

console.log(b); // 7

Swapping variables

Two variables values can be swapped in one destructuring expression.

Without destructuring assignment, swapping two values requires a temporary variable (or, in some low-level languages, the [XOR-swap trick](https://en.wikipedia.org/wiki/XOR_swap_algorithm)).

let a = 1;

let b = 3;

[a, b] = [b, a];

console.log(a); // 3

console.log(b); // 1

const arr = [1,2,3];

[arr[2], arr[1]] = [arr[1], arr[2]];

console.log(arr); // [1,3,2]

Parsing an array returned from a function

It's always been possible to return an array from a function. Destructuring can make working with an array return value more concise.

In this example, f() returns the values [1, 2] as its output, which can be parsed in a single line with destructuring.

function f() {

return [1, 2];

}

let a, b;

[a, b] = f();

console.log(a); // 1

console.log(b); // 2

Ignoring some returned values

You can ignore return values that you're not interested in:

function f() {

return [1, 2, 3];

}

const [a, , b] = f();

console.log(a); // 1

console.log(b); // 3

You can also ignore all returned values:

[,,] = f();

Assigning the rest of an array to a variable

When destructuring an array, you can unpack and assign the remaining part of it to a variable using the rest pattern:

const [a, ...b] = [1, 2, 3];

console.log(a); // 1

console.log(b); // [2, 3]

Be aware that a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError" \o "The SyntaxError object represents an error when trying to interpret syntactically invalid code.) will be thrown if a trailing comma is used on the left-hand side with a rest element:

const [a, ...b,] = [1, 2, 3];

// SyntaxError: rest element may not have a trailing comma

Unpacking values from a regular expression match

When the regular expression [exec()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/exec) method finds a match, it returns an array containing first the entire matched portion of the string and then the portions of the string that matched each parenthesized group in the regular expression. Destructuring assignment allows you to unpack the parts out of this array easily, ignoring the full match if it is not needed.

function parseProtocol(url) {

const parsedURL = /^(\w+)\:\/\/([^\/]+)\/(.\*)$/.exec(url);

if (!parsedURL) {

return false;

}

console.log(parsedURL); // ["https://developer.mozilla.org/en-US/Web/JavaScript", "https", "developer.mozilla.org", "en-US/Web/JavaScript"]

const [, protocol, fullhost, fullpath] = parsedURL;

return protocol;

}

console.log(parseProtocol('https://developer.mozilla.org/en-US/Web/JavaScript')); // "https"

**Object destructuring**

Basic assignment

const o = {p: 42, q: true};

const {p, q} = o;

console.log(p); // 42

console.log(q); // true

Assignment without declaration

A variable can be assigned its value with destructuring separate from its declaration.

let a, b;

({a, b} = {a: 1, b: 2});

**Notes**: The parentheses ( ... ) around the assignment statement are required when using object literal destructuring assignment without a declaration.

{a, b} = {a: 1, b: 2} is not valid stand-alone syntax, as the {a, b} on the left-hand side is considered a block and not an object literal.

However, ({a, b} = {a: 1, b: 2}) is valid, as is const {a, b} = {a: 1, b: 2}

Your ( ... ) expression needs to be preceded by a semicolon or it may be used to execute a function on the previous line.

Assigning to new variable names

A property can be unpacked from an object and assigned to a variable with a different name than the object property.

const o = {p: 42, q: true};

const {p: foo, q: bar} = o;

console.log(foo); // 42

console.log(bar); // true

Here, for example, const {p: foo} = o takes from the object o the property named p and assigns it to a local variable named foo.

Default values

A variable can be assigned a default, in the case that the value unpacked from the object is undefined.

const {a = 10, b = 5} = {a: 3};

console.log(a); // 3

console.log(b); // 5

Assigning to new variables names and providing default values

A property can be both 1) unpacked from an object and assigned to a variable with a different name and 2) assigned a default value in case the unpacked value is undefined.

const {a: aa = 10, b: bb = 5} = {a: 3};

console.log(aa); // 3

console.log(bb); // 5

Unpacking fields from objects passed as function parameter

const user = {

id: 42,

displayName: 'jdoe',

fullName: {

firstName: 'John',

lastName: 'Doe'

}

};

function userId({id}) {

return id;

}

function whois({displayName, fullName: {firstName: name}}) {

return `${displayName} is ${name}`;

}

console.log(userId(user)); // 42

console.log(whois(user)); // "jdoe is John"

This unpacks the id, displayName and firstName from the user object and prints them.

Setting a function parameter's default value

function drawChart({size = 'big', coords = {x: 0, y: 0}, radius = 25} = {}) {

console.log(size, coords, radius);

// do some chart drawing

}

drawChart({

coords: {x: 18, y: 30},

radius: 30

});

In the function signature for **drawChart** above, the destructured left-hand side is assigned to an empty object literal on the right-hand side: {size = 'big', coords = {x: 0, y: 0}, radius = 25} = {}. You could have also written the function without the right-hand side assignment. However, if you leave out the right-hand side assignment, the function will look for at least one argument to be supplied when invoked, whereas in its current form, you can simply call **drawChart()** without supplying any parameters. The current design is useful if you want to be able to call the function without supplying any parameters, the other can be useful when you want to ensure an object is passed to the function.

Nested object and array destructuring

const metadata = {

title: 'Scratchpad',

translations: [

{

locale: 'de',

localization\_tags: [],

last\_edit: '2014-04-14T08:43:37',

url: '/de/docs/Tools/Scratchpad',

title: 'JavaScript-Umgebung'

}

],

url: '/en-US/docs/Tools/Scratchpad'

};

let {

title: englishTitle, // rename

translations: [

{

title: localeTitle, // rename

},

],

} = metadata;

console.log(englishTitle); // "Scratchpad"

console.log(localeTitle); // "JavaScript-Umgebung"

For of iteration and destructuring

const people = [

{

name: 'Mike Smith',

family: {

mother: 'Jane Smith',

father: 'Harry Smith',

sister: 'Samantha Smith'

},

age: 35

},

{

name: 'Tom Jones',

family: {

mother: 'Norah Jones',

father: 'Richard Jones',

brother: 'Howard Jones'

},

age: 25

}

];

for (const {name: n, family: {father: f}} of people) {

console.log('Name: ' + n + ', Father: ' + f);

}

// "Name: Mike Smith, Father: Harry Smith"

// "Name: Tom Jones, Father: Richard Jones"

Computed object property names and destructuring

Computed property names, like on [object literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#Computed_property_names), can be used with destructuring.

let key = 'z';

let {[key]: foo} = {z: 'bar'};

console.log(foo); // "bar"

Rest in Object Destructuring

The [Rest/Spread Properties for ECMAScript](https://github.com/tc39/proposal-object-rest-spread) proposal (stage 4) adds the [rest](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) syntax to destructuring. Rest properties collect the remaining own enumerable property keys that are not already picked off by the destructuring pattern.

let {a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40}

a; // 10

b; // 20

rest; // { c: 30, d: 40 }

Invalid JavaScript identifier as a property name

Destructuring can be used with property names that are not valid JavaScript [identifiers](https://developer.mozilla.org/en-US/docs/Glossary/Identifier) by providing an alternative identifier that is valid.

const foo = { 'fizz-buzz': true };

const { 'fizz-buzz': fizzBuzz } = foo;

console.log(fizzBuzz); // "true"

Combined Array and Object Destructuring

Array and Object destructuring can be combined. Say you want the third element in the array props below, and then you want the name property in the object, you can do the following:

const props = [

{ id: 1, name: 'Fizz'},

{ id: 2, name: 'Buzz'},

{ id: 3, name: 'FizzBuzz'}

];

const [,, { name }] = props;

console.log(name); // "FizzBuzz"

The prototype chain is looked up when the object is deconstructed

When deconstructing an object, if a property is not accessed in itself, it will continue to look up along the prototype chain.

let obj = {self: '123'};

obj.\_\_proto\_\_.prot = '456';

const {self, prot} = obj;

// self "123"

// prot "456"（Access to the prototype chain）

**Clone Objects**

const food = { beef: '🥩', bacon: '🥓' }

// "Spread"

{ ...food }

// "Object.assign"

Object.assign({}, food)

// "JSON"

JSON.parse(JSON.stringify(food))

// RESULT:

// { beef: '🥩', bacon: '🥓' }

**Array clone**

# How to Deep Clone an Array

There are 2 types of array cloning: shallow & deep. Shallow copies only cover the 1st level of the array and the rest are referenced. If you want a true copy of nested arrays, you’ll need a deep clone. For deep clones, go with the JSON way OR better yet use Lodash 👍

const numbers = [1, [2], [3, [4]], 5];

// Using JavaScript

JSON.parse(JSON.stringify(numbers));

// Using Lodash

\_.cloneDeep(numbers);

## Arrays are Reference Types

In order to understand why there are two types of cloning. Let's dig into the fundamentals and explains what are reference types.

Unlike your primitive types (ie. number or string), arrays are reference types. Which means when you assign an array to a variable, you're assigning a memory address and not the actual array itself. WTH 😱. I know this is a bit confusing. So let's explain with an example.

### Copying a Value type

So no biggie here. We're creating a copy of value. And if we change the valueCopy, it doesn't affect the original value. Makes sense - when we change the copy it shouldn't affect the original at all. All good here 👍

let value = 3;

let valueCopy = value; // create copy

console.log(valueCopy); // 3

// Change valueCopy

valueCopy = 100

console.log(valueCopy); // 100

// ✅ Original NOT affected

console.log(value); // 3

### Copying a Reference type

Okay, things are about to get weird now. Let's copy our array using the same method as we did to copy a value type.

let array = [1,2,3];

let arrayCopy = array; // create copy

console.log(arrayCopy); // [1,2,3];

// Change 1st element of the array

arrayCopy[0] = '👻';

console.log(arrayCopy); // [ '👻', 2, 3 ]

// ❌Original got affected

console.log(array); // [ '👻', 2, 3 ]

Why did the original array also got affected? That's because what you copied over is not the array itself but the pointer to the memory space the array occupies. Reference types don't hold values, they are a pointer to the value in memory.

**Solution to Copying Reference Types**

So the solution is to copy over the value NOT the pointer. Like this:

let array = [1,2,3];

let arrayCopy = [...array]; // create TRUE copy

console.log(arrayCopy); // [1,2,3];

// Change 1st element of the array

arrayCopy[0] = '👻';

console.log(arrayCopy); // [ '👻', 2, 3 ]

// ✅ Original NOT affected

console.log(array); // [ 1, 2, 3 ]

### Shallow vs Deep Clone

When I used spread ... to copy an array, I'm only creating a shallow copy. If the array is nested or multi-dimensional, it won't work. Let's take a look:

let nestedArray = [1, [2], 3];

let arrayCopy = [...nestedArray];

// Make some changes

arrayCopy[0] = '👻'; // change shallow element

arrayCopy[1][0] = '💩'; // change nested element

console.log(arrayCopy); // [ '👻', [ '💩' ], 3 ]

// ❌ Nested array got affected

console.log(nestedArray); // [ 1, [ '💩' ], 3 ]

As you can see, the shallow or first layer is fine. However, once we change the nested element, the original array also got affected. So the solution is to do a deep clone:

let nestedArray = [1, [2], 3];

let arrayCopy = JSON.parse(JSON.stringify(nestedArray));

// Make some changes

arrayCopy[0] = '👻'; // change shallow element

arrayCopy[1][0] = '💩'; // change nested element

console.log(arrayCopy); // [ '👻', [ '💩' ], 3 ]

// ✅ Nested array NOT affected

console.log(nestedArray); // 1, [ 2 ], 3 ]

## Community Input

### Values Not Compatible with JSON

[*Anton Istomin*](https://dev.to/tailcall/comment/96nc): One has to be really careful with JSON solution! It doesn't work with values not compatible with JSON. Consider using a library function if you have to work with such data.

function nestedCopy(array) {

return JSON.parse(JSON.stringify(array));

}

// undefineds are converted to nulls

nestedCopy([1, undefined, 2]) // -> [1, null, 2]

// DOM nodes are converted to empty objects

nestedCopy([document.body, document.querySelector('p')]) // -> [{}, {}]

// JS dates are converted to strings

nestedCopy([new Date()]) // -> ["2019-03-04T10:09:00.419Z"]

### deepClone vs JSON

[*Alfredo Salzillo*](https://dev.to/alfredosalzillo/comment/96ne): I'd like you to note that there are some differences between deepClone and JSON.stringify/parse.

* **JSON.stringify/parse** only work with Number and String and Object literal without function or Symbol properties.
* **deepClone** work with all types, function and Symbol are copied by reference.

Here's an example:

const lodashClonedeep = require("lodash.clonedeep");

const arrOfFunction = [() => 2, {

test: () => 3,

}, Symbol('4')];

// deepClone copy by refence function and Symbol

console.log(lodashClonedeep(arrOfFunction));

// JSON replace function with null and function in object with undefined

console.log(JSON.parse(JSON.stringify(arrOfFunction)));

// function and symbol are copied by reference in deepClone

console.log(lodashClonedeep(arrOfFunction)[0] === lodashClonedeep(arrOfFunction)[0]);

console.log(lodashClonedeep(arrOfFunction)[2] === lodashClonedeep(arrOfFunction)[2]);

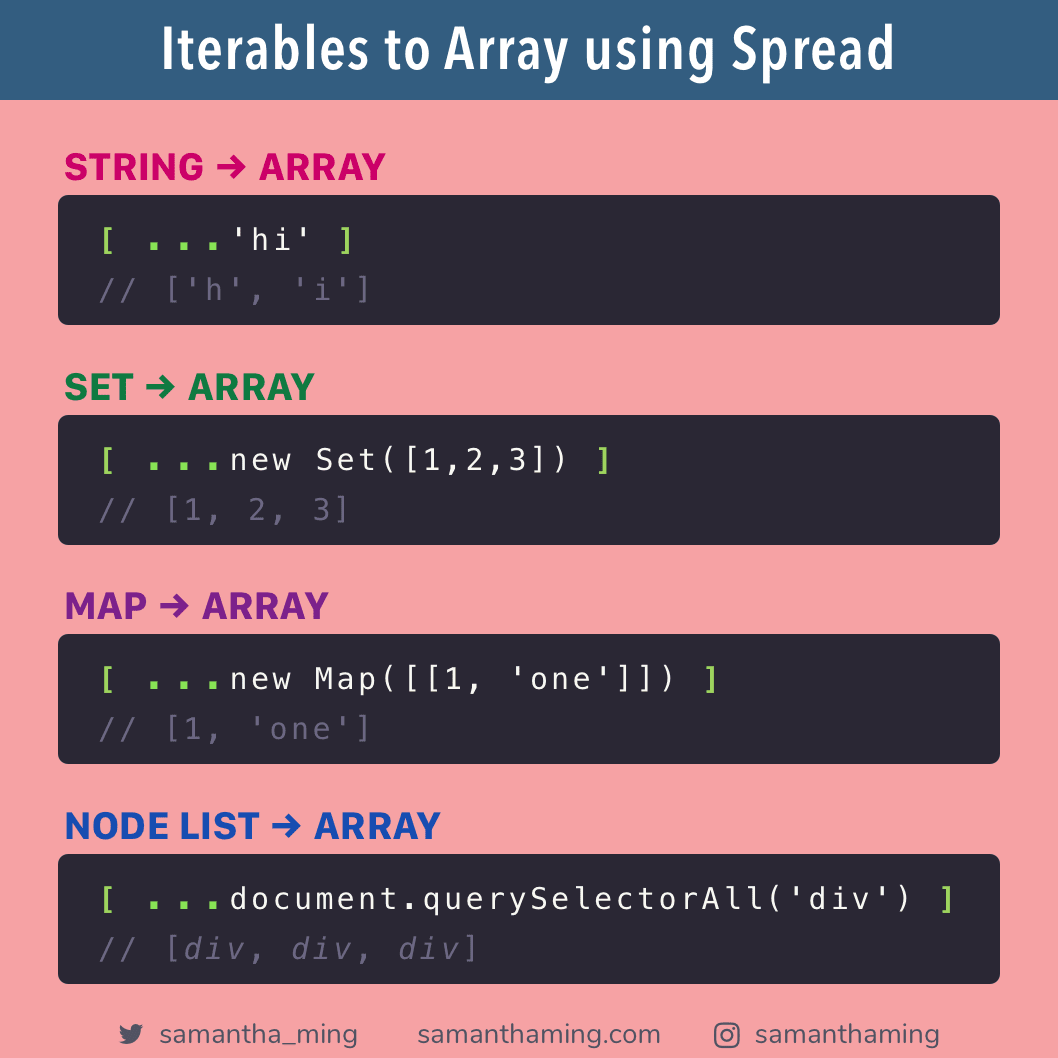
### Using Recursion

[*Tareq Al-Zubaidi*](https://medium.com/@zubaidi/there-is-other-simple-and-more-performant-solution-for-this-problem-8c8bda77d042): There is another simple and more performant solution to this problem. I would use recursion to solve this.

const clone = (items) => items.map(item => Array.isArray(item) ? clone(item) : item);

See comparison test [here](http://jsben.ch/q2ez1)

**Spread**



**Higher order functions**

**Eg1:-**

function greaterT(e) {

  return m => m > e;

}

let greaterT10 = greaterT(10);

console.log(greaterT10(12));

**Eg2:-**

function zero(...e) {

  const no = 0;

  return e.length === 0 ? no : e[0](no);

}

function one(...e) {

  const no = 1;

  return e.length === 0 ? no : e[0](no);

}

function two(...e) {

  const no = 2;

  return e.length === 0 ? no : e[0](no);

}

function three(...e) {

  const no = 3;

  return e.length === 0 ? no : e[0](no);

}

function four(...e) {

  const no = 4;

  return e.length === 0 ? no : e[0](no);

}

function five(...e) {

  const no = 5;

  return e.length === 0 ? no : e[0](no);

}

function six(...e) {

  const no = 6;

  return e.length === 0 ? no : e[0](no);

}

function seven(...e) {

  const no = 7;

  return e.length === 0 ? no : e[0](no);

}

function eight(...e) {

  const no = 8;

  return e.length === 0 ? no : e[0](no);

}

function nine(...e) {

  const no = 9;

  return e.length === 0 ? no : e[0](no);

}

function plus(f) {

  return e => parseInt(e + f);

}

function minus(f) {

  return e => parseInt(e - f);

}

function times(f) {

  return e => parseInt(e \* f);

}

function dividedBy(f) {

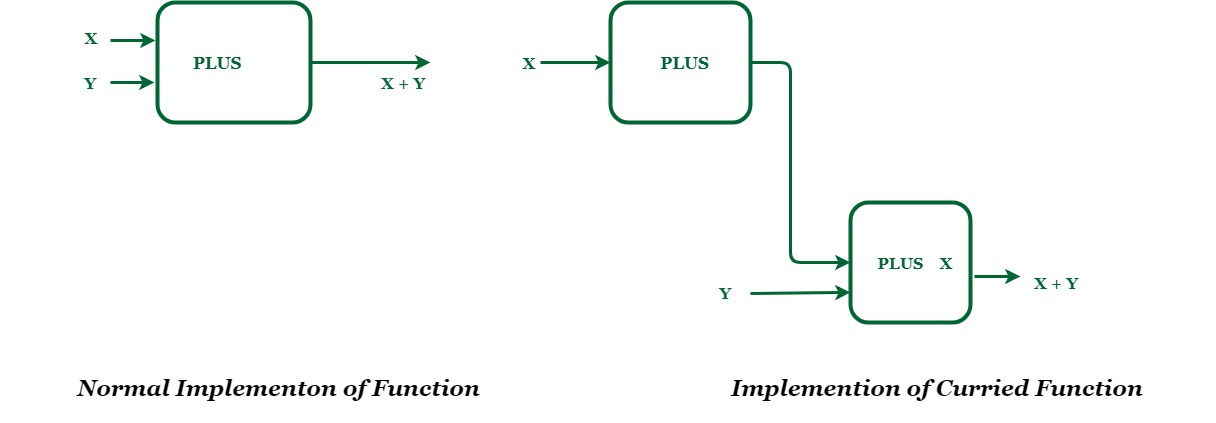
  return e => parseInt(e / f);

}

console.log(nine(dividedBy(seven())));

**A Simple Example of Currying:**  
Let’s take an example, PLUS is a function which adds two number

1. We wish to add two numbers X and Y. X will be the input to PLUS function which returns a function named PLUS X.
2. PLUS X function takes one number and add X to it. Now input to this function will be Y. Final output will be X + Y.



**Deep clone**

function add(x, y) {

  if (y == 0) return x;

  else {

    return add(x ^ y, (x & y) << 1);

  }

}

const array1 = [1, [122, [222]], { id: 123, obj: { name: "ish" } }];

const array2 = array1;

const deepClone = params => {

  return params.map(params => {

    return params instanceof Array

      ? deepClone(params)

      : params instanceof Object

      ? { ...params }

      : params;

  });

};

const array3 = deepClone(array1);

array1[1][1] = 3333;

array1[2].id = 444;

array1[2].obj.name = "sirisena";

console.log(array1);

console.log(array2);

console.log(array3);

**Object cool methods**

… Params is a object…

console.log(Object.keys(params));

  console.log(Object.values(params));

  console.log(Object.entries(params));