**JavaScript ES6— The Spread Syntax (…)**

**Spread syntax** allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

function sum(x, y, z) {

return x + y + z;

}

const numbers = [1, 2, 3];

console.log(sum(...numbers));

// expected output: 6

console.log(sum.apply(null, numbers));

// expected output: 6

## Syntax Section

For function calls:

myFunction(...iterableObj);

For array literals or strings:

[...iterableObj, '4', 'five', 6];

For object literals (new in ECMAScript 2018):

let objClone = { ...obj };

#### Replace apply

It is common to use Function.prototype.apply in cases where you want to use the elements of an array as arguments to a function.

function myFunction(x, y, z) { }

var args = [0, 1, 2];

myFunction.apply(null, args);

With spread syntax the above can be written as:

function myFunction(x, y, z) { }

var args = [0, 1, 2];

myFunction(...args);

Any argument in the argument list can use spread syntax and it can be used multiple times.

function myFunction(v, w, x, y, z) { }

var args = [0, 1];

myFunction(-1, ...args, 2, ...[3]);

#### Apply for new

When calling a constructor with new, it's not possible to **directly** use an array and apply (apply does a [[Call]] and not a [[Construct]]). However, an array can be easily used with new thanks to spread syntax:

var dateFields = [1970, 0, 1]; // 1 Jan 1970

var d = new Date(...dateFields);

To use new with an array of parameters without spread syntax, you would have to do it **indirectly** through partial application:

function applyAndNew(constructor, args) {

function partial () {

return constructor.apply(this, args);

};

if (typeof constructor.prototype === "object") {

partial.prototype = Object.create(constructor.prototype);

}

return partial;

}

function myConstructor () {

console.log("arguments.length: " + arguments.length);

console.log(arguments);

this.prop1="val1";

this.prop2="val2";

};

var myArguments = ["hi", "how", "are", "you", "mr", null];

var myConstructorWithArguments = applyAndNew(myConstructor, myArguments);

console.log(new myConstructorWithArguments);

// (internal log of myConstructor): arguments.length: 6

// (internal log of myConstructor): ["hi", "how", "are", "you", "mr", null]

// (log of "new myConstructorWithArguments"): {prop1: "val1", prop2: "val2"}

#### A more powerful array literal

Without spread syntax, to create a new array using an existing array as one part of it, the array literal syntax is no longer sufficient and imperative code must be used instead using a combination of push, splice, concat, etc. With spread syntax this becomes much more succinct:

var parts = ['shoulders', 'knees'];

var lyrics = ['head', ...parts, 'and', 'toes'];

// ["head", "shoulders", "knees", "and", "toes"]

Just like spread for argument lists, ... can be used anywhere in the array literal and it can be used multiple times.

#### Copy an array

var arr = [1, 2, 3];

var arr2 = [...arr]; // like arr.slice()

arr2.push(4);

// arr2 becomes [1, 2, 3, 4]

// arr remains unaffected

**Note:** Spread syntax effectively goes one level deep while copying an array. Therefore, it may be unsuitable for copying multidimensional arrays as the following example shows (it's the same with Object.assign() and spread syntax).

var a = [[1], [2], [3]];

var b = [...a];

b.shift().shift(); // 1

// Now array a is affected as well: [[], [2], [3]]

#### A better way to concatenate arrays

Array.concat is often used to concatenate an array to the end of an existing array. Without spread syntax this is done as:

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

// Append all items from arr2 onto arr1

arr1 = arr1.concat(arr2);

With spread syntax this becomes:

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

arr1 = [...arr1, ...arr2]; // arr1 is now [0, 1, 2, 3, 4, 5]

Array.unshift is often used to insert an array of values at the start of an existing array.  Without spread syntax this is done as:

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

// Prepend all items from arr2 onto arr1

Array.prototype.unshift.apply(arr1, arr2) // arr1 is now [3, 4, 5, 0, 1, 2]

With spread syntax, this becomes [Note, however, that this creates a new arr1 array. Unlike Array.unshift, it does not modify the original arr1 array in-place]:

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

arr1 = [...arr2, ...arr1]; // arr1 is now [3, 4, 5, 0, 1, 2]

Shallow-cloning (excluding prototype) or merging of objects is now possible using a shorter syntax than Object.assign().

var obj1 = { foo: 'bar', x: 42 };

var obj2 = { foo: 'baz', y: 13 };

var clonedObj = { ...obj1 };

// Object { foo: "bar", x: 42 }

var mergedObj = { ...obj1, ...obj2 };

// Object { foo: "baz", x: 42, y: 13 }

Note that Object.assign() triggers setters whereas spread syntax doesn't.

Note that you cannot replace nor mimic the Object.assign() function:

var obj1 = { foo: 'bar', x: 42 };

var obj2 = { foo: 'baz', y: 13 };

const merge = ( ...objects ) => ( { ...objects } );

var mergedObj = merge ( obj1, obj2);

// Object { 0: { foo: 'bar', x: 42 }, 1: { foo: 'baz', y: 13 } }

var mergedObj = merge ( {}, obj1, obj2);

// Object { 0: {}, 1: { foo: 'bar', x: 42 }, 2: { foo: 'baz', y: 13 } }

In the above example, the spread syntax does not work as one might expect: it spreads an array of arguments into the object literal, due to the rest parameter.

Spread syntax (other than in the case of spread properties) can be applied only to iterableobjects:

var obj = {'key1': 'value1'};

var array = [...obj]; // TypeError: obj is not iterable

#### The Spread Syntax

* The spread syntax is simply three dots: ...
* **It allows an iterable to expand in places where 0+ arguments are expected.**

Definitions are tough without context. Lets explore some different use cases to help understand what this means.

#### Example #1 — Inserting Arrays

Take a look at the code below. In this code, we **don’t** use the spread syntax:

Above, we’ve created an array named mid. We then create a second array which contains our mid array. Finally, we log out the result. What do you expect arr to pring? Click **run** above to see what happens. Here is the output:

[1, 2, [3, 4], 5, 6]

Is that the result you expected?

By inserting the mid array into the arr array, we’ve ended up with an array within an array. That’s fine if that was the goal, but what if want only a single array with the values of 1 through 6? To accomplish this, we can use the spread syntax! Remember, the spread syntax allows the elements of our array to **expand**.

Lets look at the code below. Everything is the same — except we’re now using the spread syntax to insert the mid array into the arr array:

And when you hit the run button, here’s the result:

[1, 2, 3, 4, 5, 6]

Remember the spread syntax definition you just read above? Here’s where it comes into play. As you can see, when we create the arr array and use the spread operator on the mid array, instead of just being inserted, the midarray expands. This expansion means that each and every element in the midarray is inserted into the arr array. Instead of nested arrays, the result is a single array of numbers ranging from 1 to 6.

#### Example #2 — Math

JavaScript has a built in math object that allows us to do some fun math calculations. In this example we’ll be looking at Math.max(). If you’re unfamiliar, Math.max() returns the largest of zero or more numbers. Here are a few examples:

Math.max();  
// -Infinity

Math.max(1, 2, 3);  
// 3

Math.max(100, 3, 4);  
// 100

As you can see, if you want to find the maximum value of multiple numbers, Math.max() requires multiple parameters. You unfortunately can’t simply use a single array as input. Before the spread syntax, the easiest way to use Math.max() on an array is to use .apply()

Instead of having to create a function and utilize the apply method to return the result of Math.max() , we only need two lines of code! The spread syntax expands our array elements and inputs each element in our array individually into the Math.max() method!

#### Example #3 — Copy an Array

In JavaScript, you can’t just copy an array by setting a new variable equal to already existing array. Consider the following code example:

When you press **run**, you’ll get the following output:

['a', 'b', 'c']

Now, at first glance, it looks like it worked — it looks like we’ve copied the values of arr into arr2. But that’s not what has happened. You see, when working with objects in javascript (arrays are a type of object) we assign by reference, not by value. This means that arr2 has been assigned to the same reference as arr. In other words, anything we do to arr2 will also affect the original arr array (and vice versa). Take a look below:

Above, we’ve pushed a new element d into arr2. Yet, when we log out the value of arr, you’ll see that the d value was also added to that array:

['a', 'b', 'c', 'd']

No need to fear though! We can use the spread operator!

Consider the code below. It’s almost the same as above. Instead though, we’ve used the spread operator within a pair of square brackets:

Hit run, and you’ll see the expected output:

['a', 'b', 'c']

Above, the array values in arr expanded to become individual elements which were then assigned to arr2. We can now change the arr2 array as much as we’d like with no consequences on the original arr array:

Again, the reason this works is because the value of arr is expanded to fill the brackets of our arr2 array definition. Thus, we are setting arr2 to equal the individual values of arr instead of the reference to arr like we did in the first example.

#### Bonus Example — String to Array

As a fun final example, you can use the spread syntax to convert a string into an array. Simply use the spread syntax within a pair of square brackets: