Spread Operator

The *spread operator* unpacks *iterables* into individual elements. Let’s look into different scenarios when this is useful.

Arrays

The spread operator unpacks an array into separate arguments:

<>

const array = [1, 2, 3];

console.log(…array);

// -> 1 2 3</span>

This comes in handy when a function expects a list of arguments and all you have is an array:

<>

const array = [4, 2, 9, 5];

Math.max(…array); // -> 9</span>

To **copy** an array we put the spread values into another array:

<>

const arrayA = [1, 2, 3];

const arrayB = [...arrayA];

console.log(arrayB);

// -> [1, 2, 3]

This is a great way of cloning arrays. If we do changes to either of the arrays they will not affect the other.

The spread syntax can also be used to compose several values into one:

<>

const arrayA = [1, 2, 3];

const arrayB = [0, ...arrayA, 4];

console.log(arrayB);

// -> [0, 1, 2, 3, 4]

This is useful when we want to add elements into an existing array. We can even **merge** two arrays:

<>

const first = [1, 2];

const other = [3, 4];

const combo = [...first, ...other];

// [1, 2, 3, 4]

These operations are not only available for arrays but also other iterables like strings:

<>

const word = 'test';

console.log([...word]);

// -> ["t", "e", "s", "t"]

Objects

The spread operator (…) with objects is used to create copies of existing objects with new or updated values or to make a copy of an object with more properties. Let’s take an example of how to use the spread operator on an object.

Here we are spreading the user object. All key-value pairs are **copied** into the clonedUser object.

<>

const user = {

name: 'Max',

age: 42

};

const clonedUser = { ...user };

The spread syntax is useful for **merging** the properties and methods on objects into a new object:

<>

const x = { x: 1 };

const y = { y: 2 };

const coord = {...x, ...y};

console.log(coord);

// {x: 1, y: 2}

**Note:** Spread syntax only does a **shallow copy** meaning nested arrays or objects will not copy properly. The deeper data is still linked to the original.

Rest Parameter

Where the *spread operator* unpacks the contents of an iterable into single elements, the *rest parameter* instead collects all the **remaining** elements into an **array**.

In JavaScript, it’s possible to call a function with any number of arguments.  
We can use the *rest parameter* when we don’t know how many arguments will be used or just want to collect some of them into an array.

Let’s try an example where we add together all the values sent into our function:

<>

function sum(...args) {

let result = 0;

for (let arg of args) {

result += arg;

}

return result

}

sum(4, 2) // -> 6

sum(3, 4, 5, 6) // -> 18

So, no matter how many numbers we send into the sum function they will be added together.

Let’s do another example:

<>

function family(spouse, …children) {

console.log(`Spouse: ${spouse}`);

for(const child of children) {

console.log(`Child: ${child}`);

}

}

family('Veronica', 'Max', 'Jack');

// -> Spouse: Veronica

// -> Child: Max

// -> Child: Jack

As shown in the above example, the three dots collected all the elements after the spouse into the children array.

We can also use the rest syntax with objects together with **destructuring**:

<>

const player = {

name: 'Max Best',

age: 42,

game: 'Football'

}

const { name, ...rest } = player;

console.log(name); // -> Max Best

console.log(rest);

// -> {age: 42, game: 'Football'}

**Note:** Rest parameters have to be the last argument. This is because it collects all remaining arguments into an array.

Conclusion

* **Rest Parameter** collects all remaining elements into an array.
* **Spread Operator** expands collected elements such as arrays into single elements.
* The spread syntax only does a **shallow copy** one level deep.
* Only the **last parameter** can be a *rest parameter*.