27th December 2019

Rest parameters and spread syntax

Many JavaScript built-in functions support an arbitrary number of arguments.

For instance:

- `Math.max(arg1, arg2, ..., argN)` – returns the greatest of the arguments.
- `Object.assign(dest, src1, ..., srcN)` – copies properties from `src1..N` into `dest`.
- …and so on.

In this chapter we'll learn how to do the same. And also, how to pass arrays to such functions as parameters.

Rest parameters `...`

A function can be called with any number of arguments, no matter how it is defined.

Like here:

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

alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of "excessive" arguments. But of course in the result only the first two will be counted.

The rest of the parameters can be included in the function definition by using three dots `...` followed by the name of the array that will contain them. The dots literally mean "gather the remaining parameters into an array".

For instance, to gather all arguments into array `args`:

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

We can choose to get the first parameters as variables, and gather only the rest.

Here the first two arguments go into variables and the rest go into `titles` array:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar

  // the rest go into titles array
  // i.e. titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

The rest parameters must be at the end

The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

```
function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
  // error
}
```

The `...rest` must always be last.

The “arguments” variable

There is also a special array-like object named `arguments` that contains all arguments by their index.

For instance:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // it's iterable
  // for(let arg of arguments) alert(arg);
}

// shows: 2, Julius, Caesar
showName("Julius", "Caesar");

// shows: 1, Ilya, undefined (no second argument)
showName("Ilya");
```

In old times, rest parameters did not exist in the language, and using `arguments` was the only way to get all arguments of the function. And it still works, we can find it in the old code.

But the downside is that although `arguments` is both array-like and iterable, it's not an array. It does not support array methods, so we can't call `arguments.map(...)` for example.

Also, it always contains all arguments. We can't capture them partially, like we did with rest parameters.

So when we need these features, then rest parameters are preferred.

Arrow functions do not have `"arguments"`

If we access the `arguments` object from an arrow function, it takes them from the outer "normal" function.

Here's an example:

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

As we remember, arrow functions don't have their own `this`. Now we know they don't have the special `arguments` object either.

Spread syntax

We've just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

For instance, there's a built-in function Math.max that returns the greatest number from a list:

```
alert( Math.max(3, 5, 1) ); // 5
```

Now let's say we have an array `[3, 5, 1]`. How do we call `Math.max` with it?

Passing it "as is" won't work, because `Math.max` expects a list of numeric arguments, not a single array:

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

And surely we can't manually list items in the code `Math.max(arr[0], arr[1], arr[2])`, because we may be unsure how many there are. As our script executes, there could be a lot, or there could be none. And that would get ugly.

Spread syntax to the rescue! It looks similar to rest parameters, also using `...`, but does quite the opposite.

When `...arr` is used in the function call, it "expands" an iterable object `arr` into the list of arguments.

For `Math.max`:

```javascript
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)
```

We also can pass multiple iterables this way:

```javascript
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8
```

We can even combine the spread syntax with normal values:

```javascript
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

Also, the spread syntax can be used to merge arrays:

```javascript
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

In the examples above we used an array to demonstrate the spread syntax, but any iterable will do.

For instance, here we use the spread syntax to turn the string into array of characters:

```javascript
let str = "Hello";

alert( [...str] ); // H,e,l,l,o
```

The spread syntax internally uses iterators to gather elements, the same way as `for..of` does.

So, for a string, `for..of` returns characters and `...str` becomes `"H","e","l","l","o"` . The list of characters is passed to array initializer `[...str]` .

For this particular task we could also use `Array.from` , because it converts an iterable (like a string) into an array:

```javascript
let str = "Hello";

// Array.from converts an iterable into an array
alert( Array.from(str) ); // H,e,l,l,o
```

The result is the same as `[...str]`.

But there's a subtle difference between `Array.from(obj)` and `[...obj]`:

- `Array.from` operates on both array-likes and iterables.
- The spread syntax works only with iterables.

So, for the task of turning something into an array, `Array.from` tends to be more universal.

Summary

When we see `"..."` in the code, it is either rest parameters or the spread syntax.

There's an easy way to distinguish between them:

- When `...` is at the end of function parameters, it's "rest parameters" and gathers the rest of the list of arguments into an array.
- When `...` occurs in a function call or alike, it's called a "spread syntax" and expands an array into a list.

Use patterns:

- Rest parameters are used to create functions that accept any number of arguments.
- The spread syntax is used to pass an array to functions that normally require a list of many arguments.

Together they help to travel between a list and an array of parameters with ease.

All arguments of a function call are also available in "old-style" `arguments`: array-like iterable object.

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)