Spread syntax (...)

The **spread (`...`)** syntax allows an iterable, such as an array or string, to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected. In an object literal, the spread syntax enumerates the properties of an object and adds the key-value pairs to the object being created.

Spread syntax looks exactly like rest syntax. In a way, spread syntax is the opposite of rest syntax. Spread syntax "expands" an array into its elements, while rest syntax collects multiple elements and "condenses" them into a single element. See rest parameters and rest property.

Try it

JavaScript Demo: Expressions - Spread syntax

```
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
```

Run ›

Reset

Syntax

```
myFunction(a, ...iterableObj, b)
[1, ...iterableObj, '4', 'five', 6]
{ ...obj, key: 'value' }
```

Description

Spread syntax can be used when all elements from an object or array need to be included in a new array or object, or should be applied one-by-one in a function call's arguments list. There are three distinct places that accept the spread syntax:

- Function arguments list (`myFunction(a, ...iterableObj, b)`)
- Array literals (`[1, ...iterableObj, '4', 'five', 6]`)
- Object literals (`{ ...obj, key: 'value' }`)

Although the syntax looks the same, they come with slightly different semantics.

Only iterable objects, like `Array`, can be spread in array and function parameters. Many objects are not iterable, including all plain objects that lack a `Symbol.iterator` method:

```
const obj = { key1: "value1" };
const array = [...obj]; // TypeError: obj is not iterable
```

On the other hand, spreading in object literals enumerates the own properties of the object. For typical arrays, all indices are enumerable own properties, so arrays can be spread into objects.

```
const array = [1, 2, 3];
const obj = { ...array }; // { 0: 1, 1: 2, 2: 3 }
```

When using spread syntax for function calls, be aware of the possibility of exceeding the JavaScript engine's argument length limit. See `Function.prototype.apply()` for more details.

Examples

Spread in function calls

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) {}
const args = [0, 1, 2];
myFunction.apply(null, args);
```

With spread syntax the above can be written as:

```
function myFunction(x, y, z) {}
const args = [0, 1, 2];
myFunction(...args);
```

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

```
function myFunction(v, w, x, y, z) {}
const args = [0, 1];
myFunction(-1, ...args, 2, ...[3]);
```

Apply for new operator

When calling a constructor with `new`, it's not possible to **directly** use an array and `apply()`, because `apply()` *calls* the target function instead of *constructing* it, which means, among other things, that `new.target` will be `undefined`. However, an array can be easily used with `new` thanks to spread syntax:

```
const dateFields = [1970, 0, 1]; // 1 Jan 1970
const d = new Date(...dateFields);
```

Spread in array literals

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:

```
const parts = ["shoulders", "knees"];
const 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 may be used more than once.

Copy an array

```
const arr = [1, 2, 3];
const 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. The same is true with `Object.assign()` — no native operation in JavaScript does a deep clone. The web API method `structuredClone()` allows deep copying values of certain supported types.
>
> ```
> const a = [[1], [2], [3]];
> const b = [...a];
>
> b.shift().shift();
> // 1
>
> // Oh no! Now array 'a' is affected as well:
> console.log(a);
> // [[], [2], [3]]
> ```

A better way to concatenate arrays

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

```
let arr1 = [0, 1, 2];
const arr2 = [3, 4, 5];

// Append all items from arr2 onto arr1
arr1 = arr1.concat(arr2);
```

With spread syntax this becomes:

```
let arr1 = [0, 1, 2];
const arr2 = [3, 4, 5];

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

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

```
const arr1 = [0, 1, 2];
const arr2 = [3, 4, 5];

//  Prepend all items from arr2 onto arr1
Array.prototype.unshift.apply(arr1, arr2);
console.log(arr1); // [3, 4, 5, 0, 1, 2]
```

With spread syntax, this becomes:

```
let arr1 = [0, 1, 2];
const arr2 = [3, 4, 5];

arr1 = [...arr2, ...arr1];
console.log(arr1); // [3, 4, 5, 0, 1, 2]
```

> **Note:** Unlike `unshift()`, this creates a new `arr1`, instead of modifying the original `arr1` array in-place.

Spread in object literals

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

```
const obj1 = { foo: "bar", x: 42 };
const obj2 = { foo: "baz", y: 13 };

const clonedObj = { ...obj1 };
// { foo: "bar", x: 42 }

const mergedObj = { ...obj1, ...obj2 };
// { foo: "baz", x: 42, y: 13 }
```

Note that `Object.assign()` can be used to mutate an object, whereas spread syntax can't.

```
const obj1 = { foo: "bar", x: 42 };
Object.assign(obj1, { x: 1337 });
console.log(obj1); // { foo: "bar", x: 1337 }
```

In addition, `Object.assign()` triggers setters on the target object, whereas spread syntax does not.

```
const objectAssign = Object.assign(
  {
    set foo(val) {
      console.log(val);
    },
  },
  { foo: 1 },
);
// Logs "1"; objectAssign.foo is still the original setter

const spread = {
  set foo(val) {
    console.log(val);
  },
  ...{ foo: 1 },
};
// Nothing is logged; spread.foo is 1
```

You cannot naively re-implement the `Object.assign()` function through a single spreading:

```
const obj1 = { foo: "bar", x: 42 };
const obj2 = { foo: "baz", y: 13 };
const merge = (...objects) => ({ ...objects });

const mergedObj1 = merge(obj1, obj2);
// { 0: { foo: 'bar', x: 42 }, 1: { foo: 'baz', y: 13 } }

const mergedObj2 = merge({}, obj1, obj2);
// { 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. Here is an implementation of `merge` using the spread syntax, whose behavior is similar to `Object.assign()`, except that it doesn't trigger setters, nor mutates any object:

```
const obj1 = { foo: "bar", x: 42 };
const obj2 = { foo: "baz", y: 13 };
const merge = (...objects) =>
  objects.reduce((acc, cur) => ({ ...acc, ...cur }));

const mergedObj1 = merge(obj1, obj2);
// { foo: 'baz', x: 42, y: 13 }
```

Specifications

Specification
ECMAScript Language Specification # prod-SpreadElement
ECMAScript Language Specification # prod-ArgumentList
ECMAScript Language Specification # prod-PropertyDefinition

Browser compatibility

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS
Spread syntax (...)	✓ Chrome 46	✓ Edge 12	✓ Firefox 16	✓ Opera 37	✓ Safari 8	✓ Chrome 46 Android	✓ Firefox 16 for Android	✓ Opera 37 Android	✓ Safa on i
Spread in array	✓ Chrome 46	✓ Edge 12	✓ Firefox 16	✓ Opera 37	✓ Safari 8	✓ Chrome 46 Android	✓ Firefox 16 for	✓ Opera 37 Android	✓ Safa on i

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS
literals									
Spread in function calls	✓ Chrome 46	✓ Edge 12	✓ Firefox 27	✓ Opera 37	✓ Safari 8	✓ Chrome 46 Android	✓ Firefox 27 for Android	✓ Opera 37 Android	✓ Safa on iO
Spread in object literals	✓ Chrome 60	✓ Edge 79	✓ Firefox 55	✓ Opera 47	✓ Safari 11.1	✓ Chrome 60 Android	✓ Firefox 55 for Android	✓ Opera 44 Android	✓ Safa on iOS

Tip: you can click/tap on a cell for more information.

✓ Full support ✗ No support ⚑ User must explicitly enable this feature. ··· Has more compatibility info.

See also

- Rest parameters
- Rest property
- `Function.prototype.apply()`

This page was last modified on Feb 21, 2023 by MDN contributors.

mdn web docs