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.

Syntax

For function calls:

```
myFunction(...iterableObj);
```

For array literals or strings:

For array literals or strings:

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

For object literals (new in ECMAScript; stage 3 draft):

```
let objClone = { ...obj };
```

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) { }
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.

```
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.prot
   }
   return partial;
}


function myConstructor () {
   console.log("arguments.length: " + arguments.length);
```

```
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(myConstruct

console.log(new myConstructorWithArguments);
// (internal log of myConstructor):           arguments.
// (internal log of myConstructor):           ["hi", "ho
// (log of "new myConstructorWithArguments"): {prop1: "v
```

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:

```
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

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];
```

With spread syntax this becomes:

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

`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 nov
```

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
```

Spread in object literals

The Rest/Spread Properties for ECMAScript proposal (stage 3) adds spread properties to object literals. It copies own enumerable properties from a provided object onto a new object.

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 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.

Only for iterables

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

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

Spread with many values

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

Rest syntax (parameters)

Rest syntax looks exactly like spread syntax, but is used for destructuring arrays and objects. In a way, rest syntax is the opposite of spread syntax: spread 'expands' an array into its elements, while rest collects multiple elements and 'condenses' them into a single element. See rest parameters.

Rest syntax looks exactly like spread syntax, but is used for destructuring arrays and objects. In a way, rest syntax is the opposite of spread syntax: spread 'expands' an array into its elements, while rest collects multiple elements and 'condenses' them into a single element. See rest parameters.
