The Utility of Rest & Spread

Learn the rest and spread parameters and how they allow us to seamlessly work with data contained in multiple objects and arrays. We'll go over their uses and in the end, we'll put them together with advanced destructuring and default function parameters to show how these new tools aggregate to give us increased functionality in an small, elegant package.

Array & Object Rest/Spread

`...` is called the rest operator and the spread operator, depending on its usage. It can *spread* items in an array out into another array. It can *spread* properties of an object into another object.

Array rest/spread was introduced in ES2015. Object rest/spread was introduced in 2017.

This lesson has been placed here because it ties in with the discussion of `apply`, `call`, and `bind`. Let's dive in.

Spread

```
const arr = [1, 2, 3, 4, 5];
const arr2 = ['abc', 'def', ...arr];
console.log(arr2); // -> [ 'abc', 'def', 1, 2, 3, 4, 5 ]

const obj = { item1: 'Hello', item2: 'there' };
const obj2 = { ...obj, item3: '!' };
console.log(obj2); // -> { item1: 'Hello', item2: 'there', item3: '!' }
```

`...` is *spreading out* an array into another array and an object into another object. This is useful for a number of things.

Copying Items

Copying an array or object is now as simple as using `...`.

```
const arr = [1, 2, 3];
const arrCopy = [...arr];

const obj = { item: 'value' };
const objCopy = { ...obj };

console.log(arrCopy); // -> [ 1, 2, 3 ]
console.log(objCopy); // -> { item: 'value' }
```

- This makes a shallow copy - only references are copied. They are not recursively copied.
- Using `...` with an object is essentially the same as using `Object.assign`, except for the fact that we're creating a new object.

It can also spread array values into a function.

```
function add(num1, num2) {
    console.log(num1 + num2);
}

add(...[1, 2]); // -> 3
```

Strings

Strings can be spread into arrays and objects.

```
const str = 'abcde';

const strArr = [...str];
console.log(strArr);
// -> [ 'a', 'b', 'c', 'd', 'e' ]

const strObj = { ...str };
console.log(strObj);
// -> { '0': 'a', '1': 'b', '2': 'c', '3': 'd', '4': 'e' }
```

`arguments`

`...` provides a much easier way of turning `arguments` into a real array than using `Array.slice` and `function.call`.

```
function add() {
    const args = [...arguments];
    console.log(args);
}

add(1, 2); // -> [ 1, 2 ]
```

This shows that `...` works on array-like objects, as long as they're iterable, meaning that they have indexed properties.

There's an even easier way.

Rest

`...` can collect the *rest* of the items of an array or array-like object into a new array.

```
function add(...args) {
    console.log(args);
}

add(1, 2); // -> [ 1, 2 ]
```

It's collecting the parameters in the function definition and turning them into an array. `arguments` is now obsolete.

It's called the rest operator because it doesn't necessarily collect all of the items, just the rest of them, after the ones we strip off first are gone.

```
function add(num1, ...args) {
    console.log(num1);
    console.log(args);
}
```

```
add(1, 2, 3);
// -> 1
// -> [2, 3]
```

Advanced Destructuring

The rest/spread operator adds some functionality to destructuring.

```
const arr = [1, 2, 3, 4, 5];
const [one, two, ...rest] = arr;
console.log(rest); // -> [3, 4, 5]

const obj = {
    key1: 'val1',
    key2: 'val2',
    key3: 'val3'
};

const { key1, ...others } = obj;
console.log(others); // -> { key2: 'val2', key3: 'val3' }
```

We can decide that we want to pluck off the first few items of an array, then turn the rest of them into a new array. Same for an object. It'll always collect the rest of the items after we've plucked off the ones we want.

The rest element must always be the last element when destructuring. We can't pull out the middle of an array. This goes for objects as well.

```
const arr = [1, 2, 3, 4, 5];
const [one, two, ...rest, five] = arr;
// -> Uncaught SyntaxError: Rest element must be last element
```

Function Arguments, Destructuring & `...`

We can use `...` when destructuring in function parameters.

```
function fn({ name, age, ...otherDetails }) {
    console.log(name);
    console.log(age);
    console.log(otherDetails);
}

const obj = {
    name: 'Alex Jones',
    age: 30,
    profession: 'developer',
    salary: 80000,
    address: 'CA, USA'
};

fn(obj);
// -> Alex Jones
// -> 30
// -> { profession: 'developer', salary: 80000, address: 'CA, USA' }
```

Putting it All together

Let’s see what we can do with destructuring, default destructuring values, default parameters, and `...`.

```
function fn({ name = 'John Doe', age = 25, ...otherDetails } = {}) {
    console.log(name);
    console.log(age);
    console.log(otherDetails);
}

fn();
// -> John Doe
// -> 25
// -> {}
```

Let’s walk through this one.

`fn` is expecting an object given as an argument. We’re calling it with no arguments. We use default function parameters to default to the empty object at the end of the argument declaration, `function fn({...} = {})`. It’s as if we passed in an empty object.

The function will attempt to pick off `name`, `age`, and all other properties from this empty object. Of course, this empty object doesn’t have any of those items, so the function will use default destructuring to give them values.

We've made it so a user can enter in an object and the function will pluck off the `name` and `age` from the object and throw the rest of the properties into `otherDetails`.

`name` and `age`, if not present on the object passed in, will default to some value. `otherDetails` will be empty if nothing else is present.

If the user doesn't pass in anything, the function still runs without error. All of this is done before we even get to the function body. That's pretty powerful.

Phew. That's it.

Test Yourself

These exercises are difficult. Feel free to attempt them. They cover multiple lessons that we've been through so far.

```
// This is a repeat of a problem in a previous
// section. The goal this time around is to take
// the solution and shorten it using our new
// ES2015 functionality.

const obj = {
    groceries: {
        cost: 33.22,
        amount: 1
    },
    rent: {
        cost: 899.00,
        amount: 1,
    },
    paycheck: {
        cost: -2000,
        amount: 2
    },
    restaurantBills: {
        cost: 20,
        amount: 4
    }
};

// Shorten the long method chain in this function.
// You might want to use arrow function and
// new Object functions.

// The goal is to:
// Filter out all items with a negative cost.
// Multiply the cost of each remaining item times
// the amount purchased and add it all up.
```

```
const sum = Object.keys(obj)
    .map(function(key) {
      return obj[key];

    }).filter(function(item) {
      return item.cost > 0;
    }).map(function(item) {
      return item.cost * item.amount;
    }).reduce(function(sum, next) {
      return sum + next;
    });
```

```
// Write the shortest function you possibly can
// that takes in an array and returns a new array
// that is identical to the original array,
// but a different reference. In other words,
// create an array shallow-copy function.

shallowCopy=// Finish this
```

```
// Write one line of code using the reduce
// function that will turn this array of 0s
// into an array of 0s and 1s.
// NO USING THE CHARACTERS { OR }

function fn() {
    const arr1 = [0, 0, 0];
    // We want to return [0, 1, 0, 1, 0, 1]

    return arr1.reduce(/* Finish this */);
}
```