Spread Operator and Destructuring (Arrays)

Desctructuring

Destructuring is the process of breaking down values in a collection (objects or arrays) into separate variables. This is useful when we want to quickly use values individually without having the need to access the collection itself.

Here's an example of how you can destructure an array.

```
const [a, b] = [10, 20]

console.log(a) // 10
console.log(b) // 20
```

In the example above, the variables `a` and `b` now hold values `10` and `20` respectively because we have destructured them. As you can see, the syntax itself is quite easy, we only need to use brackets (`[]`) to wrap our variable names. Then, we can assign it to an array. The way it works is the value which will then be stored in the variables, correspond according to the order of the values in the array itself.

For example, when the values of an array from start to end are `10`, `20`, and `30`. When we destructure the array, the variables' values will also start from `10`, `20`, and then `30`.

```
const [a, b, c] = [10, 20, 30]
//     10 20 30
```

Note that destructuring also works for arrays stored in a variable, like so.

```
const arr = [10, 20]
const [a, b] = arr

console.log(a) // 10
console.log(b) // 20
```

But sometimes, we don't need to destructure all of the values stored within an array. No problem, we can simply stop destructuring the array whenever we want, assuming that the number of variables do not exceed the number of values in the array.

```
const arr = [10, 20, 30]
const [a, b] = arr

console.log(a) // 10
console.log(b) // 20
```

In the example above we only destructured values `10` and `20` into `a` and `b` respectively. `30` however, has been ignored and not destructured which isn't a problem. To access `30` we can either destructure it by further adding a variable, or just use pointer indexes like usual (`arr[2]`).

Now, one thing that confuses a lot of people is that destructuring doesn't modify the array at all. That means if we've destructured an array, but still want to use pointer indexes, then it is completely fine.

```
const arr = [10, 20, 30]
const [a, b] = arr

console.log(arr[0]) // 10
console.log(arr[1]) // 20
console.log(arr[2]) // 30
```

Spread Operator

The spread operator (`...`) is an operator where you can practically "open" the brackets of objects or arrays. It is usually used to "duplicate" objects and arrays or modify its contents. Here's how you can do it.

```
const arr = [10, 20, 30, 40]
const newArr = [ ...arr ] // [ 10, 20, 30, 40 ]
```

In the example above, we spread `arr` into another array called `newArr`, and now `newArr` has the exact same value as `arr`. What happened is after applying `...` to `arr`, we "opened" `arr`'s brackets, revealing only the values of the array separated by commas. Another important thing to keep in mind is that `arr` and `newArr` are to separate arrays. They **DO NOT** refer to the same arrays since `newArr` duplicates the contents of `arr`, **not refering** to it.

📝 So, in a way when we spread `arr`'s content, this happens:

`[10, 20, 30, 40]` → `10, 20, 30, 40`

Notice how in the first example, after spreading `arr`, we still needed to wrap it with brackets (`[]`). This relates to this statement.

> The spread operator (`...`) is an operator where you can practically "open" the brackets of objects or arrays.

Because the spread operator takes out our brackets, that means when we assign the already spread array into a variable, we need to replace the brackets with new ones.

```
const arr = [10, 20, 30, 40]

const arrNew1 = ...arr // Wrong
// arrNew1 = 10, 20, 30, 40
```

This example is a wrong example of assigning an array that has been spread, into a variable. Once again, it is invalid because `...arr` is the same as writing `10, 20, 30, 40` **not** `[10, 20, 30, 40]`.

We can also use the spread operator to "copy" and add more items to an array. Here's an example.

```
const arr = [10, 20, 30, 40]
const newArr = [ -10, 0, ...arr, 50, 60 ]

console.log(newArr) // [ -10, 0, 10, 20, 30, 40, 50, 60 ]
```

In the example above, we've copied `arr`'s value into `newArr` but we've also added some more values into the array. Remember, because spreading an array is basically just removing its brackets, that means we can add items before and after the spread array.