CADT IDT

Next-Gen Engagement Program

JAVASCRIPT

Week 3

Spread and Rest Operator + Array Methods

Ory Chanraksa & Sao Visal

REST OPERATOR

- Use to pack array elements when you don’t know how many elements are there

- Can be used in desctructuring when you don’t know how many elements are there or take only a few of the elements and use rest for the rest of the object or array

USE REST TO PACK UP ELEMENTS

- Using rest operator here is to pack all of the elements when you don't know how much element it will be as shown in this example

- You can put how many elements there and it will pack up all of it

```
const rest=(...array)=>{
    console.log(array);
}

rest(1,2,3,4,5);
// Result: [1,2,3,4,5]

rest(1,2,3,4,5,6,7);
// Result: [1,2,3,4,5,6,7]
```

USE REST TO PACK UP THE REST OF THE ELEMENTS

- Use rest operator combine with normal parameters to get the desired parameters and use rest to get the rest of the parameters
- You can get as many elements from the start and can use rest to get the rest of it

```
const getFirst = (first, ...rest) => {
    console.log(`first: ${first}`);
    console.log(`rest: ${rest}`);
};

getFirst(1, 2, 3, 4, 5);
// Result: first: 1
//         rest: [2, 3, 4, 5]

const getFirstAndSecond = (first, second, ...rest) => {
    console.log(`first: ${first}`);
    console.log(`second: ${second}`);
    console.log(`rest: ${rest}`);
};

getFirstAndSecond(1, 2, 3, 4, 5);
// Result: first: 1
//         second: 2
//         rest: [3, 4, 5]
```

USING REST IN DESTRUCTURING ARRAY

- Using rest enable to have a variable to contain the rest of the element after the normal variable declared before it

```
let array=[1,2,3,4,5,6]

let [first, second, ...rest] = array;
console.log(first);
console.log(second);
console.log(rest);
// Result: first: 1,
//         second: 2,
//         rest: [3, 4, 5, 6]
```

USING REST IN DESTRUCTURING OBJECT

- Slightly different to destructuring array, in object it doesn't mater which order you declare it, it will behave normal and using rest will get the element that is not declared earlier

```
let object={
    name: 'John',
    age: 30,
    city: 'New York',
    country: 'USA'
};

let {name,city,...rest} = object;
console.log(name);
console.log(city);
console.log(rest);
// Result: name: John,
//         city: New York,
//         rest: {age: 30, country: 'USA'}
```

SPREAD OPERATOR

- Spread is opposite to rest, instead of packing the elements it unpack the a big package and returns all the elements

- It is most used when wanting to copy, merge, or add extra elements to an existing one

USE SPREAD TO COPY AN ARRAY

- When copying an array using spread is recommended since it takes the elements itself instead of making a reference to the array is copied

- Using spread will create a new array and when the array that was copied previously, it won't be affected which contrast to just using “=”

```
let array = [1, 2, 3, 4, 5];

let normalCopy = array;
let restCopy = [...array];

array.push(6);

console.log(normalCopy);
console.log(restCopy);
// Result: normalCopy: [1, 2, 3, 4, 5, 6]
//         restCopy: [1, 2, 3, 4, 5]
```

USE SPREAD TO COPY AN OBJECT

- Same thing applied for object since it takes the elements instead of making a reference to the object so when the original object get changed the object that used spread won't be affected

```
let object = {name: "John", age: 30};

let normalCopy = object;
let restCopy = {...object};

object.name = "Jane";

console.log(normalCopy.name);
console.log(restCopy.name);
// Result: normalCopy: {name: "Jane"}
//         restCopy: {name: "John"}
```

USE SPREAD TO MERGE ARRAY

- Using spread will spread out those elements so you get all the elements itself

- The difference between the first example and the second is the first one you spread all the elements to get one array and the second one you get 2 arrays inside an array instead

```
const array1 = [1, 2, 3, 4, 5, 6];
const array2 = [7, 8, 9, 10, 11, 12];

const combinedWithSpread = [...array1, ...array2];
console.log(combinedWithSpread);
// Result:
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

const combinedWithoutSpread = [array1, array2];
console.log(combinedWithoutSpread);
// Result:
// [[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]]
```

USE SPREAD TO MERGE OBJECT

- Using spread will spread out those elements so you get all the elements itself

- The difference between the first example and the second is the first one you spread all the elements to get one object and the second one you get 2 object inside an object iinstead

```
const object1 = {name: 'Kim', age: 30};
const object2 = {name: 'Kong', age: 28};

const combinedWithSpread = {...object1, ...object2};
console.log(combinedWithSpread);
// Result:
// { name: 'Kong', age: 28 }

const combinedWithoutSpread = {object1, object2};
console.log(combinedWithoutSpread);
// Result:
// {
//  object1: { name: 'Kim', age: 30 },
//  object2: { name: 'Kong', age: 28 }
// }
```

USE SPREAD TO ADD A SET OF ARRAY TO AN ARRAY

- Spread can be used to unpack or spread all of its elements to anywhere inside an array

```
const array = [1, 2, 3, 4, 5, 6];

const spreadAtTheStart = [...array, 7];
console.log(spreadAtTheStart);
// Result:
// [1, 2, 3, 4, 5, 6, 7]

const spreadAtTheMiddle = [0, ...array, 7];
console.log(spreadAtTheMiddle);
// Result:
// [0, 1, 2, 3, 4, 5, 6, 7]

const spreadAtTheEnd = [0, ...array];
console.log(spreadAtTheEnd);
// Result:
// [0, 1, 2, 3, 4, 5, 6]
```

USE SPREAD TO ADD A SET OF OBJECT TO AN OBJECT

- Spread can be used to unpack or spread all of its elements to anywhere inside an object

```
const user = { name: "Alice", age: 30, city: "New York" };

const spreadAtTheStart = { ...user, country: "USA" };
console.log(spreadAtTheStart);
// Result:
// { name: 'Alice', age: 30, city: 'New York', country: 'USA' }

const spreadAtTheMiddle = { id: 101, ...user, country: "USA" };
console.log(spreadAtTheMiddle);
// Result:
// { id: 101, name: 'Alice', age: 30, city: 'New York', country: 'USA' }

const spreadAtTheEnd = { id: 101, ...user };
console.log(spreadAtTheEnd);
// Result:
// { id: 101, name: 'Alice', age: 30, city: 'New York', country: 'USA' }
```

ARRAY METHODS

Arrays in JavaScript come with many built-in methods that make it easy to work with, change, or analyze the data inside them.

These methods let you:

- Add or remove elements
- Search or find values
- Transform or process arrays
- Combine, split, or rearrange arrays
- Loop through elements efficiently
- Many other...

PUSH, POP, SHIFT, UNSHIFT

- **push()** – Add element(s) to the end of an array.
- **pop()** – Remove element from the end of an array.
- **shift()** – Remove element from the start of an array.
- **unshift()** – Add element(s) to the start of an array.

```
const push = [1, 2, 3];
console.log(push.push(4));
// Results: 1, 2, 3, 4

const pop = [1, 2, 3];
console.log(pop.pop());
// Results: 1, 2

const shift = [1, 2, 3];
console.log(shift.shift());
// Results: 2, 3

const unshift = [1, 2, 3];
console.log(unshift.unshift(0));
// Results: 0, 1, 2, 3
```

SLICE

- Used to extract a portion of an array into a new array without modifying the original array.

- Very useful when you need a subset of data or want to copy part of an array.

- **Syntax:**
  - **array.slice(start, end):** Extracts elements from index start up to but not including end
  - **array.slice(start):** Extracts elements from index start to the end of the array.

- **Negative number:** Normally you write positive number like 0 mean index 0, but using negative number it counts from the end to the start.

SLICE

```
const numbers=[1,2,3,4,5]

console.log(numbers.slice(1,4));
// Result: [2, 3, 4]

console.log(numbers.slice(2));
// Result: [3, 4, 5]

console.log(numbers.slice(-3));
// Result: [3, 4, 5]

console.log(numbers.slice(-2, -1));
// Result: [4]
```

INCLUDES, INDEXOF, FIND

- **includes() –** Check if an array contains a value.
- **indexOf() –** Find the index of a value.
- **find() –** Return the first element matching a condition.

```
const includes = [1, 2, 3];
console.log(includes.includes(2));
// Results: true
console.log(includes.includes(4));
// Results: false

const indexOf = [1, 2, 3];
console.log(indexOf.indexOf(2));
// Results: 1 (index)
console.log(indexOf.indexOf(4));
// Results: -1 (not found)

const find = [1, 2, 3];
console.log(find.find(value => value === 2));
// Results: 2
console.log(find.find(value => value > 2));
// Results: 3
```

FOREACH, FILTER, MAP

forEach():

- Executes a function on each element of an array.
- Does not return a new array — just performs an action.
- Useful for looping through arrays when you don't need a new array or transformation.

FOREACH, FILTER, MAP

filter():

- Executes a function on each element of an array.
- Returns a new array containing only elements that pass a test.
- Original array remains unchanged.
- Useful when you want to select certain elements based on a condition.

FOREACH, FILTER, MAP

map():

- Executes a function on each element of an array.
- Returns a new array with the transformed values.
- Original array remains unchanged.
- Useful when you want to transform each element and get a new array of results.

CALLBACK ARGUMENTS IN FOREACH, MAP, FILTER

When using forEach, map, or filter, the callback function can receive up to three arguments:

1. **element –** The current item being processed **(most commonly used)**.

2. **index –** The position of the element in the array **(used often when index matters)**.

3. **array –** Return the whole array **(rarely used, only use when needing the whole array).**

- You don't have to use all three — include only what you need.

- You can name them whatever you want, it will be assign to its position.
  Position: 1 = element, 2 = index, and 3 = array.

- If an argument is not used, you can replace it with '_' to indicate it's intentionally ignored. **Ex:** (_, index) means you only need the index.

FOREACH()

Simple

```
const numbers = [1, 2, 3, 4, 5];

numbers.forEach((num) => console.log(num * 2));
// Results: 2 4 6 8 10

numbers.forEach((num, index) => {
    const squareNum = num * num;
    console.log(`Index: ${index}, Square: ${squareNum}`);
});
// Index: 0, Square: 1
// Index: 1, Square: 4
// Index: 2, Square: 9
// Index: 3, Square: 16
// Index: 4, Square: 25

numbers.forEach((_, index) => {
    console.log(`Index: ${index}`);
});
// Index: 0
// Index: 1
// Index: 2
// Index: 3
// Index: 4
```

Real-World use case

```
const ids = [1, 2, 3, 4, 5];

const writeToDatabase = (data) => {
    // Function to write data to the database
    console.log("Writing data:", data);
};

ids.forEach((id) => {
    writeToDatabase({ id });
});
```

FILTER()

```
const numbers = [1, 2, 3, 4, 5];

numbers.filter((number) => number > 2);
// Results: 3, 4, 5

const students = [
    { name: "Alice", major: "Computer Science" },
    { name: "Bob", major: "Computer Science" },
    { name: "Charlie", major: "Physics" },
    { name: "David", major: "Computer Science" },
    { name: "Eve", major: "Biology" },
];

students.filter((student) => student.major === "Computer Science");
//Results: Alice, Bob, David
```

MAP()

```
const numbers = [1, 2, 3, 4, 5];

numbers.map((number) => number * 2);
// Results: 2, 4, 6, 8, 10

const students = [
    { name: "Alice", major: "Computer Science" },
    { name: "Bob", major: "Computer Science" },
];

const studentWithYear = students.map((student) => {
    return({
        ...student,
        year: 2
    })
});
console.log(studentWithYear);
// Results: [
//   { name: "Alice", major: "Computer Science", year: 2 },
//   { name: "Bob", major: "Computer Science", year: 2 },
// ]
```

MORE METHODS

https://www.w3schools.com/js/js_array_methods.asp

https://youtu.be/RVxuGCWZ_8E?si=fBFYiaKSHuG3RFEw

THANK YOU