Arrays and Iterables,

WEB
6 - Client Server applications

Document summary

TBD

Table of content

1. Educational objectives

- Read and use the Array object’s methods.
- Resolve problems using the Array object’s methods.
- Read and use the functional methods of the Array object.
- Resolve problems using the functional methods of the Array object.
- Read and use the Iterator and Generator objects.
- Read and use the Map and Set objects.
- Describe the anatomy of a web application.
- Describe alternatives to ExpressJS.

2. Arrays

2.1. Array methods

- `concat()` concatenates two or more arrays and returns a new array.
- `join()` joins all elements of an array into a string.
- `pop()` removes the last element from an array and returns that element.
- `push()` adds one or more elements to the end of an array and returns the new length of the array.
- `reverse()` reverses the order of the elements of an array.
- `shift()` removes the first element from an array and returns that element.
- `slice()` selects a part of an array, and returns it as a new array.
- `sort()` sorts the elements of an array.
- `includes()` determines whether an array contains a specified element.
- `flat()` flattens an array up to the specified depth.

2.1.1. concat()

```
const a = ['a', 'b', 'c'];
const b = ['d', 'e', 'f'];
const c = a.concat(b);

console.log(c); // ['a', 'b', 'c', 'd', 'e', 'f']
console.log(a); // ['a', 'b', 'c']
console.log(b); // ['d', 'e', 'f']
```

2.1.2. join()

```
const a = ['a', 'b', 'c'];
const b = a.join(' - ');

console.log(b); // 'a - b - c'

const c = [{ name: 'John' }, { name: 'Jane' }];
const d = c.join(', ');

console.log(d); // '[object Object], [object Object]'
```

2.1.3. pop()

```
const a = ['a', 'b', 'c'];
const b = a.pop();

console.log(b); // 'c'
console.log(a); // ['a', 'b']
```

2.1.4. push()

```
const a = ['a', 'b', 'c'];
const b = a.push('d');

console.log(b); // 4
console.log(a); // ['a', 'b', 'c', 'd']
```

2.1.5. reverse()

```
const a = ['a', 'b', 'c'];
const b = a.reverse();

console.log(b); // ['c', 'b', 'a']
console.log(a); // ['c', 'b', 'a']
```

2.1.6. shift()

```
const a = ['a', 'b', 'c'];
const b = a.shift();
```

```
console.log(b); // 'a'
console.log(a); // ['b', 'c']
```

2.1.7. slice()

```
const a = ['a', 'b', 'c'];
const b = a.slice(1, 2);

console.log(b); // ['b']
console.log(a); // ['a', 'b', 'c']
```

2.1.8. sort()

```
const a = ['c', 'a', 'b'];
const b = a.sort();

console.log(b); // ['a', 'b', 'c']
console.log(a); // ['a', 'b', 'c']

const c = [{ name: 'John' }, { name: 'Jane' }];
const d = c.sort((a, b) => a.name.localeCompare(b.name)); // yes, the 'a' and 'b' are the
parameters of the function and not the arrays
// const d = c.sort((x, y) => x.name.localeCompare(y.name)); // equivalent
console.log(d); // [{ name: 'Jane' }, { name: 'John' }]
```

2.1.9. includes()

```
const a = ['a', 'b', 'c'];
const b = a.includes('b');

console.log(b); // true

const c = [1, 2, 3];
const d = c.includes('2');

console.log(d); // false
console.log(c['2']); // 3
console.log(c[2]); // 3
```

2.1.10. flat()

```
const a = [1, 2, [3, 4, [5, 6]]];
const b = a.flat();

console.log(b); // [1, 2, 3, 4, [5, 6]]
console.log(a); // [1, 2, [3, 4, [5, 6]]]
```

2.2. Functional methods

2.2.1. forEach

```
const a = ['a', 'b', 'c'];
const b = a.forEach(element => console.log(element));

console.log(b); // undefined
console.log(a); // ['a', 'b', 'c']
```

2.2.2. map

```
var a = ["apple", "banana", "pear"];
const b = a.map(a => a.length)

console.log(b); // [5, 6, 4]
console.log(a); // ["apple", "banana", "pear"]
```

2.2.3. flatMap

```
var a = ['Yverdon is', 'a', 'beautiful city'];
a.flatMap(s => s.split(" "));
// First executes map: [['Yverdon', 'is'], 'a', ['beautiful', 'city']]
// Then flattens: ['Yverdon', 'is', 'a', 'beautiful', 'city'].
```

2.2.4. filter

```
const words = ['Yverdon', 'is', 'a', 'beautiful', 'city'];
const result = words.filter(word => word.length > 6);

console.log(words); // ['Yverdon', 'is', 'a', 'beautiful', 'city']
console.log(result); // ['Yverdon', 'beautiful']
```

2.2.5. reduce

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce(
  (accumulator, currentValue) => accumulator + currentValue,
  initialValue,
);

console.log(sumWithInitial);
// Expected output: 10
```

2.2.6. every & some

```
var a = [1, 2, 3];
a.every(a => a > 0) // true
a.every(a => a > 1) // false
a.some(a => a > 1) // true
```

2.2.7. find & findIndex

```
var a = [1, 2, 3];
console.log(a.find(a => a > 2)) // 3
console.log(a.findIndex(a => a > 2)) // 2
console.log(a.find(a => a > 3)) // undefined
```

3. Iterators and Generators

3.1. Iterator

Iterators are objects that provide a `next()` method which returns an object with two properties:

- value: the next value in the iterator, and
- done: whether the last value has already been provided.

Iterables are objects that have a `Symbol.iterator` method that returns an iterator over them.

```
let idGenerator = {};

idGenerator[Symbol.iterator] = function() {
    return {
        nextId: 0,
        next() {
            if (this.nextId < 10) {
                return { value: this.nextId++, done: false };
            } else {
                return { done : true };
            }
        }
    }
}

for (let id of idGenerator) {
    console.log(id); // 0 /n 1 /n 2 /n ... 9
}
```

3.2. Generator

Generators are functions that can be paused and resumed. They are created using the `function*` syntax and yield values using the `yield` keyword.

```
let idGenerator = {};
idGenerator[Symbol.iterator] = function() {
    return {
        nextId: 0,
        next() {
            if (this.nextId < 10) {
                return { value: this.nextId++, done: false };
            } else {
                return { done : true };
            }
        }
    }
}
idGenerator[Symbol.iterator] = function* () {
    let nextId = 0;
    while (nextId < 10) {
        yield nextId++;
    }
};
for (let id of idGenerator) {
    console.log(id); // 0 /n 1 /n 2 /n ... 9
}
```

3.3. Built-in iterables

3.3.1. Map

```
let map = new Map();
map.set("key", "value");
map.get("key");
```

```
map.delete("key");
map.has("key");
map.forEach((key, value) => console.log(key, value));
```

3.3.2. Set

```
let set = new Set();
set.add("value");
set.has("value");
set.delete("value");
```

3.4. Exemple

3.4.1. Flattening generator

```
const arr = [1, [2, [3, 4], 5], 6];
const flattened = [...flatten(arr)];
console.log(flattened); // [1, 2, 3, 4, 5, 6]

function* flatten(arr) {
    for (const elem of arr) {
        if (Array.isArray(elem)) {
            yield* flatten(elem);
        } else {
            yield elem;
        }
    }
}
```

3.4.2. Sum of squares

```
const arr = [1, 2, 3, 4, 5];
const sum = sumOfSquares(arr);
console.log(sum); // 55 (1^2 + 2^2 + 3^2 + 4^2 + 5^2)

function sumOfSquares(arr) {
    return arr
        .map(num => num ** 2)
        .reduce((acc, num) => acc + num, 0);
}
```

3.4.3. Counting words

```
const countWords = () => {
    return text.split(" ")
        .map(word => word.toLowerCase())
        .filter(word => /^[a-z]+$/.test(word))
        .reduce((count, word) => {
            count[word] = (count[word] || 0) + 1;
            return count;
        }, {});
}
const text = "the quick brown fox jumps over the lazy dog";
const wordCount = countWords(text);
console.log(wordCount);
// Output:
// {
// "the": 2,
// "quick": 1,
// "brown": 1,
// "fox": 1,
// "jumps": 1,
// "over": 1,
// "lazy": 1,
```

```
// "dog": 1
// }
```