Arrays in javascript

An array in JavaScript is a data structure used to store multiple values in a single variable. , typically of the same type, but JavaScript arrays can hold elements of different types as well. Arrays are zero-indexed, meaning the first element is at index 0.

Array literal syntax:

let fruits = ['apple', 'banana', 'orange'];

Using the Array constructor:

let fruits = new Array('apple', 'banana', 'orange');

Empty array:

let emptyArray = [];

Accessing Array Elements

You access array elements using their index, starting from 0:

const fruits = ['apple', 'banana', 'orange'];

console.log(fruits[0]); // Output: 'apple'

console.log(fruits[1]); // Output: 'banana'

Modifying Array Elements

You can modify an element by directly assigning a new value to a specific index:

fruits[0] = 'grape'; // Now the first element is 'grape' (even though we declare arrays with const , we can modify values as the array is a non-primitive data type. We can't change values of primitive data types.

console.log(fruits); // Output: ['grape', 'banana', 'orange']

Methods in array

1) Adding elements into array

Elements can be added to the array using methods like push() and unshift().

- The push() method add the element to **the end of the array.**
- The unshift() method add the element to the **starting of the array.**

const a = ["HTML", "CSS", "JS", 2, 3, 4, true, undefined, null];

//array methods

//1.push : it will add an element to the end of the array.

a.push("ram");

console.log(a);

//2.The unshift() method add the element to the starting of the array.

a.unshift("dhoni");

console.log(a);

output

['dhoni', 'HTML', 'CSS', 'JS', 2, 3, 4, false, undefined, null, 'ram']

2) Removing Elements from an Array

To remove the elements from an array we have different methods like pop(), shift(), or splice().

- **The pop()** method removes an element from the **last index of the array.**
- **The shift()** method removes the element from the **first index of the array**.
- **The splice()** method **removes or replaces the element from the array**.

// The pop() method removes an element from the last index of the array.

a.pop();

console.log(a);

//The shift() method removes the element from the first index of the array.

```
a.shift();

console.log(a);
```

3) Search for an element in an array.

In JavaScript, both indexOf() and includes() are used to search for an element in an array.

1. indexOf()

The indexOf() method is used to search for a specified element in an array and returns the **first index** at which that element is found. If the element is not found, it returns -1.

```
let fruits = ['apple', 'banana', 'orange', 'banana'];

let index = fruits.indexOf('banana');

console.log(index); // Output: 1 (the first occurrence of 'banana')

let notFoundIndex = fruits.indexOf('grape');

console.log(notFoundIndex); // Output: -1 (since 'grape' is not in the array)
```

2. includes()

The includes() method is used to check if a specified element exists in an array. It returns a **boolean value** (true if the element is found, false if it is not).

```
let fruits = ['apple', 'banana', 'orange', 'banana'];

let isIncluded = fruits.includes('banana');

console.log(isIncluded); // Output: true (since 'banana' is in the array)
```

let isNotIncluded = fruits.includes('grape');

console.log(isNotIncluded); // Output: false (since 'grape' is not in the array)

4) Increase and Decrease the Array Length

let arr = ["HTML", "CSS", "JS"]

// Increase the array length to 7arr.length = 7;

console.log("After Increasing Length: ", arr);

// Decrease the array length to 2

arr.length = 2;

console.log("After Decreasing Length: ", arr);

5) Array Concatenation

Combine two or more arrays using the concat() method. It returns a new array containing joined array elements.

//array concatenation

// Creating an Array and Initializing with Values

let array1 = ["HTML", "CSS", "JS", "React"];

let array2 = ["Node.js", "Expess.js"];

// Concatenate both arrays

let concateArray = array1.concat(array2);

console.log("Concatenated Array: ", concateArray);

output

Concatenated Array:['HTML', 'CSS', 'JS', 'React', 'Node.js', 'Expess.js']

An array can store objects also.

arrays can store objects just like any other data type, such as numbers, strings, or even other arrays. Each element in an array can be an object, and these objects can contain properties and methods.

Ex:

```
const students = [
  { name: 'John', age: 18 },
  { name: 'Sara', age: 20 },
  { name: 'Mike', age: 22 },
];
console.log(students);


//2
const employee1 = {
  name: 'raju',
  salary: 23000,
};
const employee2 = {
  name: 'ram',
  salary: 24000,
};
const employee3 = {
  name: 'rani',
  salary: 24000,
};
const emplAarray = [employee1, employee1, employee3];
console.log(emplAarray);
```

Output

```
[
  { name: 'John', age: 18 },
  { name: 'Sara', age: 20 },
  { name: 'Mike', age: 22 }
]
[
  { name: 'raju', salary: 23000 },
  { name: 'ram', salary: 24000 },
  { name: 'rani', salary: 24000 }
]
```

Array can store set also

```
const set1 = new Set([1, 2, 3]);
const set2 = new Set([4, 5, 6]);
const arrayWithSets = [set1, set2]; // Array storing Set objects
console.log(arrayWithSets);
```

Output

```
[
  Set { 1, 2, 3 },

  Set { 4, 5, 6 }

]
```

Map also

```
const map1 = new Map([

  ['name', 'Alice'],

  ['age', 25]

]);

const map2 = new Map([

  ['name', 'Bob'],

  ['age', 30]

]);

const arrayWithMaps = [map1, map2]; // Array storing Map objects

console.log(arrayWithMaps);
```

Output

```
[ Map(2) { 'name' => 'Alice', 'age' => 25 }, Map(2) { 'name' => 'Bob', 'age' => 30 } ]
```

Project using Arrays

JavaScript arrays **are objects**. They are a special type of object specifically designed to hold ordered collections of values.

ex

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

console.log(typeof arr); // "object"

It has also have few more methods

i)slice(start,end)

Like the slice method in Strings, Arrays also have a slice method to get elements /data from arrays without changing the original array.

The slice(start, end) method only works if start < end. If start >= end, it will return an empty array.

```
const array = ['1', '5', 'A', 'N', 'V', 'J'];
console.log(array.slice(2)); //['A', 'N', 'V', 'J']
console.log(array.slice(1, 4)); //['5', 'A', 'N']
console.log(array.slice(1, -3)); //['5', 'A']
console.log(array.slice(-4)); //['A', 'N', 'V', 'J']
console.log(array.slice(-1, -4)); //start >= end, it will return an empty array.
console.log(array.slice()); // we can create shallow copy using slice() also.['1', '5', 'A', 'N', 'V', 'J']
console.log([...array]); //['1', '5', 'A', 'N', 'V', 'J']
```

ii)Splice

It is similar to slice() but it will modify the original array.
The splice() method in JavaScript is used to add, remove, or replace elements in an array. It modifies the original array and can return the removed elements.
Syntax:

array.splice(start, deleteCount, item1, item2, ..., itemN)

start: The index at which to start changing the array.
deleteCount: The number of elements to remove starting from the start index.
item1, item2, ..., itemN *(optional)*: The elements to add to the array starting from the start index.

Use Cases:

Remove elements: If only the start and deleteCount are specified, elements are removed.
Add elements: If deleteCount is 0 and new elements are specified, the new elements are added without removing any.

Replace elements: By specifying deleteCount and new elements, existing elements can be replaced.

```
//SPLICE
// Removing Elements

const fruits = ['Apple', 'Banana', 'Mango', 'Orange'];
const removed = fruits.splice(1, 2); // Removes 2 elements starting from index 1
console.log(fruits); // ["Apple", "Orange"]
console.log(removed); // ["Banana", "Mango"]

//adding elements
const colors = ['Red', 'Green', 'Blue'];
colors.splice(1, 0, 'Yellow', 'Pink'); // Adds at index 1, removes 0 elements
console.log(colors); // ["Red", "Yellow", "Pink", "Green", "Blue"]

// Replacing Elements
const animals = ['Dog', 'Cat', 'Rabbit'];
animals.splice(1, 1, 'Elephant', 'Tiger'); // Replaces 1 element at index 1 with two new elements
console.log(animals); // ["Dog", "Elephant", "Tiger", "Rabbit"]

// Removing All Elements from an Index
const numbers = [1, 2, 3, 4, 5];
numbers.splice(2); // Removes all elements from index 2
console.log(numbers); // [1, 2]
```

iii)reverse()

The reverse() method in JavaScript reverses the order of elements in an array. It modifies the original array and also returns the reversed array.

Syntax:

array.reverse()

Return value: The reversed array (modifies the original array).

```
//Reverse
//Reversing an Array
const number = [1, 2, 3, 4, 5];
console.log(number.reverse()); // [5, 4, 3, 2, 1]
console.log(number); // [5, 4, 3, 2, 1]

const words = ['one', 'two', 'three', 'four'];
console.log(words.reverse()); // ["four", "three", "two", "one"]
console.log(words); // ["four", "three", "two", "one"]

//Combining with join() to Reverse a String
let str = 'hello';
let reversedStr = str.split('').reverse().join('');
console.log(reversedStr); // "olleh"
```

iv) concat

The concat() method in JavaScript is used to merge two or more arrays or values into a new array. It does not modify the original arrays; instead, it returns a new array that contains the combined elements.
concat() **does not change the original arrays**; it returns a new array.

array.concat(value1, value2, ..., valueN)

```
//concat
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let result = arr1.concat(arr2);
console.log(result); // [1, 2, 3, 4, 5, 6]
console.log(arr1); // [1, 2, 3]  (original array remains unchanged)

//using ...spread opertor
const res = [...arr1, ...arr2];
console.log(res); // [1, 2, 3, 4, 5, 6]
```

v)join

The join() method in JavaScript is used to join all elements of an array into a single string. You can specify a separator to insert between the elements, or it defaults to a comma if no separator is provided.

array.join(separator)

```
//join
//Joining Array Elements with a Comma (Default Separator)
let fruits = ['Apple', 'Banana', 'Mango'];
let result1 = fruits.join();
console.log(result1); // "Apple,Banana,Mango"

//Joining with a Custom Separator
let colors = ['Red', 'Green', 'Blue'];
let result2 = colors.join(' - ');
console.log(result2); // "Red - Green - Blue"
//Joining Array Elements Without a Separator
let letters = ['a', 'b', 'c'];
let result3 = letters.join('');
console.log(result3); // "abc"
//Joining Array Elements with a Space
let words = ['Hello', 'World'];
let result4 = words.join(' ');
console.log(result4); // "Hello World"
```

vi) at()

The at() method is part of ECMAScript 2022 (ES13), so it may not be supported in older environments (e.g., some legacy browsers).

The at() method in JavaScript is used to access an element at a specific index in an array or a string, with support for **negative indices**. Negative indices count from the end of the array or string, making it easier to access elements from the end without needing to calculate the length.

array.at(index)

In **traditional indexing**, we calculate the last index using fruits.length - 1.
With **at()**, we can directly use at(-1) to get the last element, which simplifies the code.

Before at()

```
let fruits = ['Apple', 'Banana', 'Mango', 'Orange'];
console.log(fruits[0]); // Apple
// Accessing the last element using length and slice
let lastElement = fruits[fruits.length - 1];
```

```
console.log(lastElement); // "Orange"
console.log(fruits.slice(-1)); //['Orange']
```

After at()

```
let fruits = ['Apple', 'Banana', 'Mango', 'Orange'];
console.log(fruits.at(0)); // Apple
// Accessing the last element using `at()`
let lastElement = fruits.at(-1);
console.log(lastElement); // "Orange"
console.log(fruits.at(-3)); //Banana
```

Looping Arrays

For..of

Use for...of when you need more control over the iteration (e.g., breaking the loop or iterating over non-array objects).

The for...of loop is used to iterate over iterable objects such as arrays, strings, maps, sets, etc. It gives you direct access to the value of each element in the iterable.

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

for (const num of arr) {
  console.log(num); // 10, 20, 30, 40
}
```

Ex2

```
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

for (const [i, mov] of movements.entries()) {
  console.log(`index ${i + 1}: ${mov}`);
}
```

Output

```
index 1: 200
index 2: 450
index 3: -400
index 4: 3000
index 5: -650
index 6: -130
index 7: 70
index 8: 1300
```

It provides access to each element directly.
It works with any iterable (like Arrays, Strings, Maps, etc.).
It can be stopped early using break or continue.

Example with break:

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

for (const num of arr) {
  if (num === 30) break;
  console.log(num); // 10, 20
}
```

forEach

The forEach method in JavaScript is used to iterate over arrays. It takes a callback function and executes it once for each element in the array, in order.

Use forEach() when you just need to execute a function for each item in an array and don't need to modify the flow.
It is an array method and is not available on other iterable objects like Map or Set.

The function passed to forEach() receives the current element, index, and the original array as arguments.

```
array.forEach(function(currentValue, index, array) {
  // Code to execute for each element
});
```

currentValue: The current element being processed in the array.
index: The index of the current element.
array(optional):The array that the forEach method is being called on. It is not mandatory to use. You can omit it, and the loop will still function as expected.

Example 1: Basic Iteration over an Array

```
const fruits = ['Apple', 'Banana', 'Orange'];

fruits.forEach(function(fruit, index) {
  console.log(`Index: ${index}, Fruit: ${fruit}`);
});
```

Flow of Execution:

- The forEach loop starts with the first element, "Apple".
  - It calls the callback with fruit = 'Apple' and index = 0.
  - The output will be: Index: 0, Fruit: Apple
- Next, it moves to the second element, "Banana".
  - It calls the callback with fruit = 'Banana' and index = 1.
  - The output will be: Index: 1, Fruit: Banana
- Finally, it processes the third element, "Orange".
  - It calls the callback with fruit = 'Orange' and index = 2.
  - The output will be: Index: 2, Fruit: Orange

Output:

```
Index: 0, Fruit: Apple
Index: 1, Fruit: Banana
Index: 2, Fruit: Orange
```

Example 2: Using the index and array Parameters

You can use the index and array parameters to access additional information about the iteration.

```
const numbers = [10, 20, 30, 40];

numbers.forEach(function(number, index, array) {
  console.log(`Element at index ${index}: ${number}, Full array: ${array}`);
});
```

Flow of Execution:

- When iterating over 10 (at index 0), the callback will receive:
  - number = 10, index = 0, array = [10, 20, 30, 40]

    - The output will be: Element at index 0: 10, Full array: [10, 20, 30, 40]
- When iterating over 20 (at index 1), the callback will receive:
    - number = 20, index = 1, array = [10, 20, 30, 40]
    - The output will be: Element at index 1: 20, Full array: [10, 20, 30, 40]
- This process continues for each element in the array.

Output:

```
Element at index 0: 10, Full array: 10,20,30,40
Element at index 1: 20, Full array: 10,20,30,40
Element at index 2: 30, Full array: 10,20,30,40
Element at index 3: 40, Full array: 10,20,30,40
```

forEach with Maps and sets

Map

```
const currencies = new Map([
  ['USD', 'United States dollar'],
  ['EUR', 'Euro'],
  ['GBP', 'Pound sterling'],
]);

currencies.forEach(function (value, key, map) {
  console.log(`${key}: ${value}`);
});
```

Output

```
USD: United States dollar
EUR: Euro
GBP: Pound sterling
```

Set

wrong

```
const currencies1 = new Set(['USD', 'EUR', 'GBP']);

currencies1.forEach(function (value, key, set) {
  console.log(`${value},${key},${set}`);
});
```

USD,USD,[object Set]
EUR,EUR,[object Set]
GBP,GBP,[object Set]

A Set in JavaScript is a collection of unique values, but it does not store key-value pairs like a Map. Instead, it stores just **values** without any associated keys. Therefore, when using forEach on a Set, the callback function will receive only the **value** (not a key)

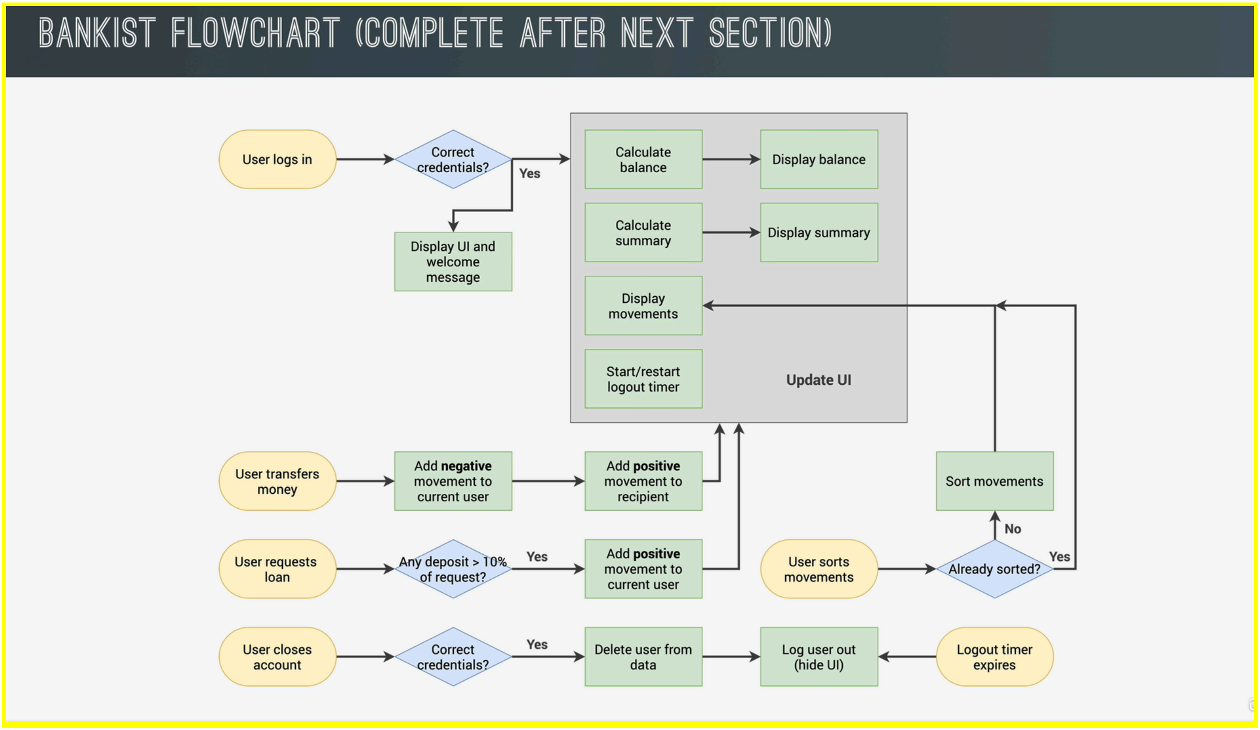
Correct

```
const currencies1 = new Set(['USD', 'EUR', 'GBP']);
currencies1.forEach(function (value) {
  console.log(`${value}`);
});
```

Output
USD
EUR
GBP

Projects Using Arrays

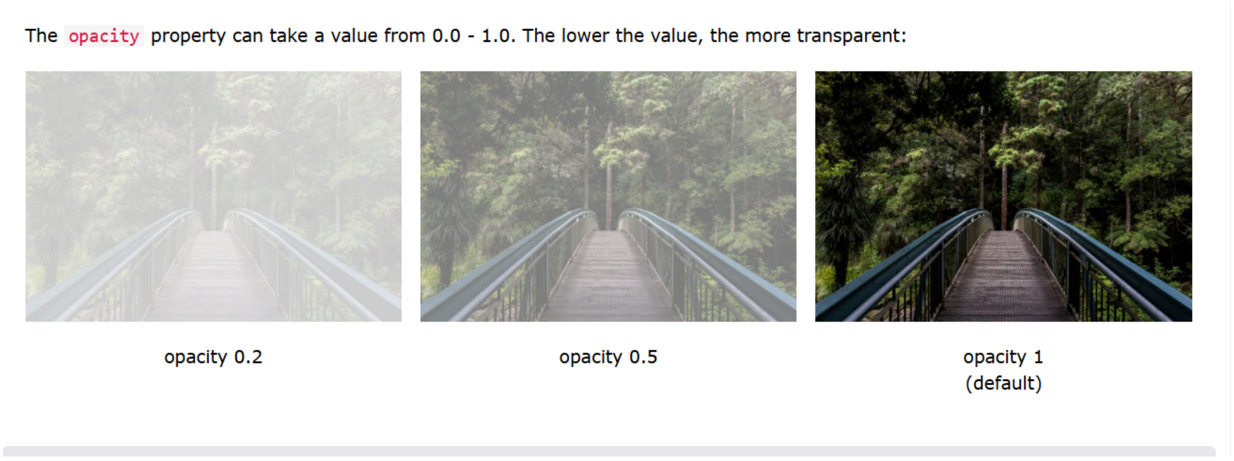
Project : Bankist App



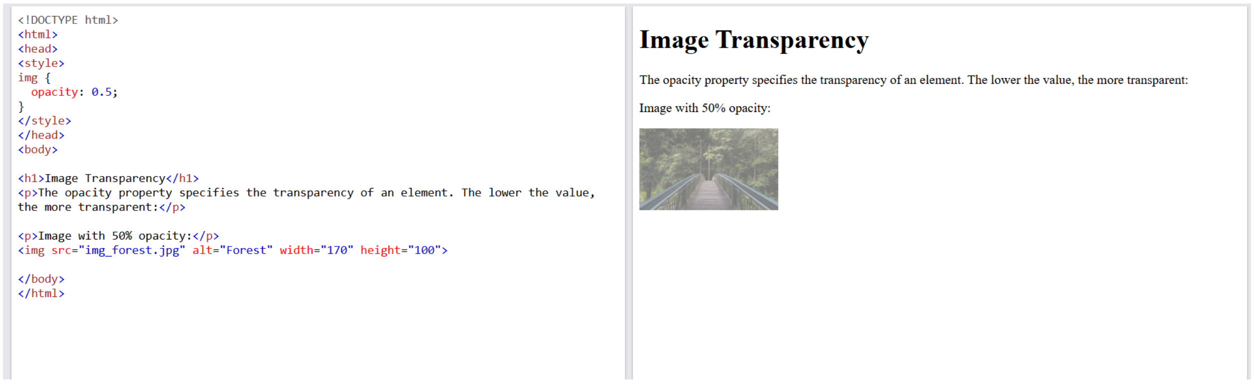
The opacity property specifies the opacity/transparency of an element.

Transparent Image

The opacity property can take a value from 0.0 - 1.0. The lower the value, the more transparent:


The `opacity` property can take a value from 0.0 - 1.0. The lower the value, the more transparent:

0.5



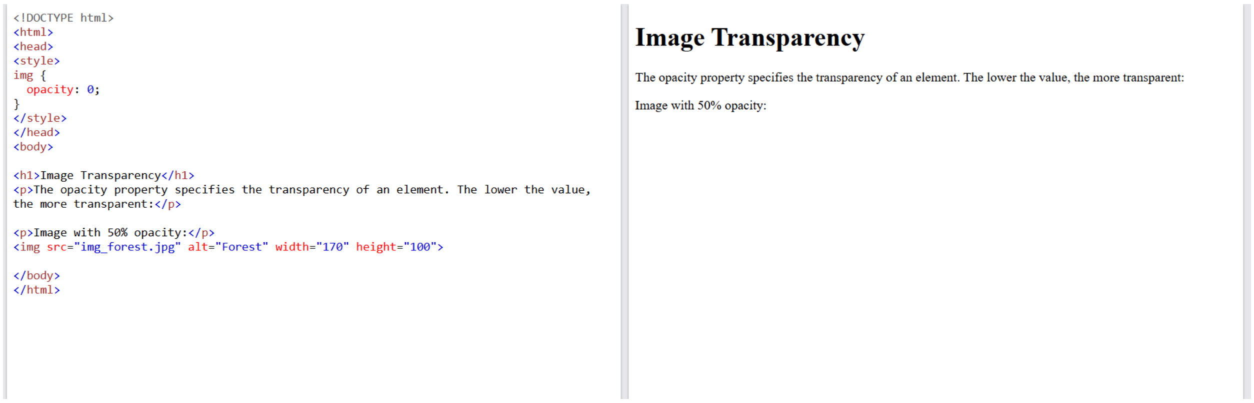
```
<!DOCTYPE html>
<html>
<head>
<style>
img {
  opacity: 0.5;
}
</style>
</head>
<body>

<h1>Image Transparency</h1>
<p>The opacity property specifies the transparency of an element. The lower the value,
the more transparent:</p>

<p>Image with 50% opacity:</p>
<img src="img_forest.jpg" alt="Forest" width="170" height="100">

</body>
</html>
```

Image Transparency

The opacity property specifies the transparency of an element. The lower the value, the more transparent:

Image with 50% opacity:

0(image is there but invisible)



```
<!DOCTYPE html>
<html>
<head>
<style>
img {
  opacity: 0;
}
</style>
</head>
<body>

<h1>Image Transparency</h1>
<p>The opacity property specifies the transparency of an element. The lower the value,
the more transparent:</p>

<p>Image with 50% opacity:</p>
<img src="img_forest.jpg" alt="Forest" width="170" height="100">

</body>
</html>
```

Image Transparency

The opacity property specifies the transparency of an element. The lower the value, the more transparent:

Image with 50% opacity:

insertAdjacentHTML()

The insertAdjacentHTML() method inserts HTML code into a specified position.

Syntax

***element*.insertAdjacentHTML(*position, html*)**

or

***node*.insertAdjacentHTML(*position, html*)**

The DOM insertAdjacentHTML() method is used to insert a text as HTML file to a specified position. This method is used to change or add text as HTML.

Syntax :

node.insertAdjacentHTML(specify-position, text-to-enter)

Return Value : This will return the page with the specified change. There are four legal position values that can be used.

- afterbegin
- afterend
- beforebegin
- beforeend

Positions	**Effect**
afterbegin :	**This will add text when the selected element just begin.**
afterend :	**This will add text when the selected element just end.**
beforebegin :	**This will add text when the selected element about to begin.**
beforeend :	**This will add text when the selected element about to end.**

Example-1: This is the example for the “afterbegin” position.

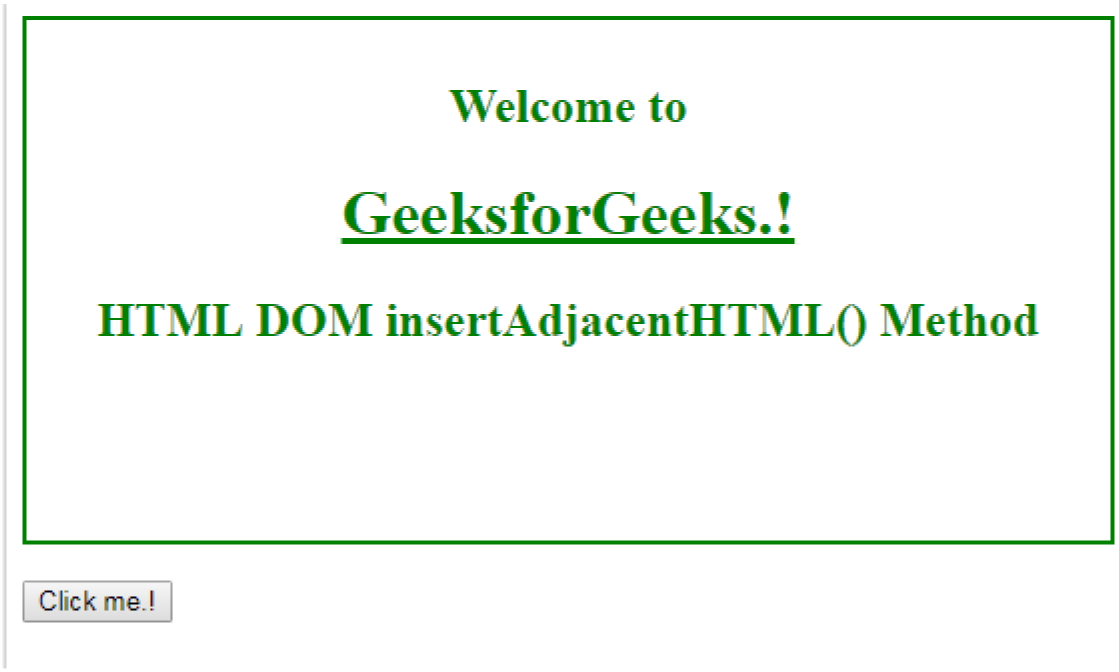
```
<!DOCTYPE html>
<html>

<head>
        <title>
```

```
                HTML | DOM insertAdjacentHTML() Method
        </title>
        <style>
                        h1,
                h2 {
                        color: green;
                        text-align: center;
                }

                div {
                        width: 80%;
                        height: 240px;
                        border: 2px solid green;
                        padding: 10px;
        </style>
</head>

<body>
        <div>
                <h2>Welcome to</h2>
                <h1>
                <u>GeeksforGeeks.!</u>
                </h1>
                <h2 id="main">
                HTML DOM insertAdjacentHTML() Method
                </h2>
        </div>
        <br>
        <button onclick="myFunction()">Click me.!</button>

        <script>
                function myFunction() {
                        var h = document.getElementById("main");
                        h.insertAdjacentHTML("afterbegin",
                                "<span style='color:green; " +
                                "background-color: lightgrey; " +
                                "width: 50%;'>This is Example of</span>");
                }
        </script>
```

</body>

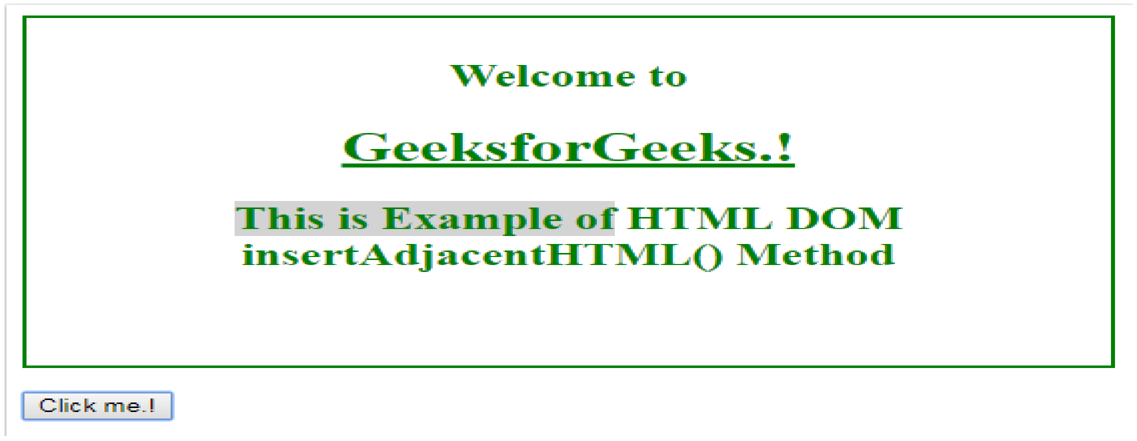
</html>

Output : Before click on the button:



```
Welcome to

GeeksforGeeks.!

HTML DOM insertAdjacentHTML() Method
```

After click on the button:



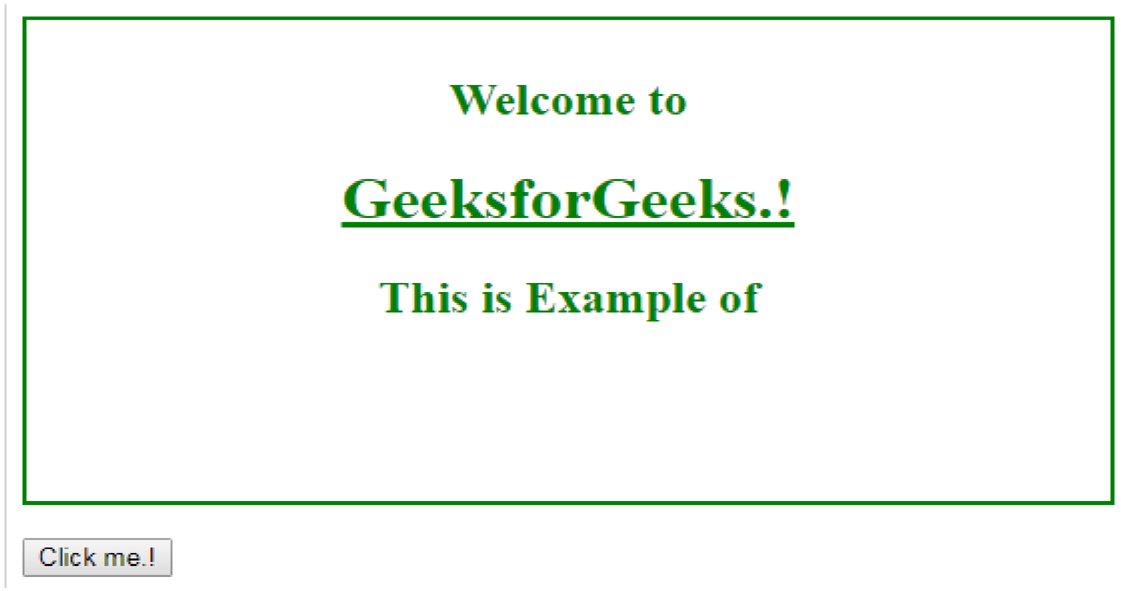
```
Welcome to

GeeksforGeeks.!

This is Example of HTML DOM
insertAdjacentHTML() Method
```

Example-2: This is the example for the **"afterend"** position.
<!DOCTYPE html>
<html>

<head>

```
        <title>
                HTML | DOM insertAdjacentHTML() Method
        </title>
        <style>
                        h1,
                h2 {
                        color: green;
                        text-align: center;
                }

                div {
                        width: 80%;
                        height: 240px;
                        border: 2px solid green;
                        padding: 10px;
        </style>
</head>

<body>
        <div>
                <h2>Welcome to</h2>
                <h1><u>GeeksforGeeks.!</u></h1>
                <h2 id="main"> This is Example of</h2>
        </div>
        <br>
        <button onclick="myFunction()">Click me.!</button>

        <script>
                function myFunction() {
                        var h = document.getElementById("main");
                        h.insertAdjacentHTML("afterend",
                                "<span style='color:green; " +
                                "background-color: lightgrey;" +
                                "font-size: 25px; " +
                                "padding-left: 30px;" +
                                "padding-right: 30px;" +
                                "width: 50%;'>" +
                                "HTML DOM insertAdjacentHTML() Method" +
                                "</span>");
                }
        </script>

</body>

</html>
```

Output : Before click on the button:

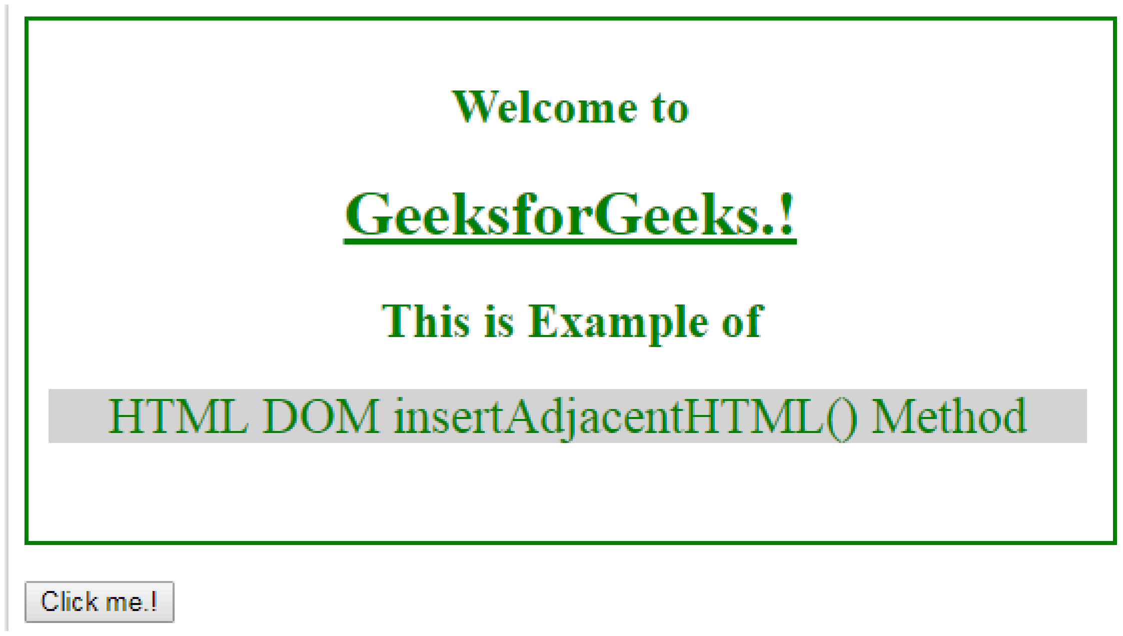


Welcome to

GeeksforGeeks.!

This is Example of

After click on the button:



Welcome to

GeeksforGeeks.!

This is Example of

HTML DOM insertAdjacentHTML() Method

Coding Challenge #

Working With Arrays Coding Challenge #1 Julia and Kate are doing a study on dogs. So each of them asked 5 dog owners about their dog's age, and stored the data into an array (one array for each). For now, they are just interested in knowing whether a dog is an adult or a puppy. A dog is an adult if it is at least 3 years old, and it's a puppy if it's less than 3 years old.

Your tasks:

Create a function 'checkDogs', which accepts 2 arrays of dog's ages ('dogsJulia' and 'dogsKate'), and does the following things:

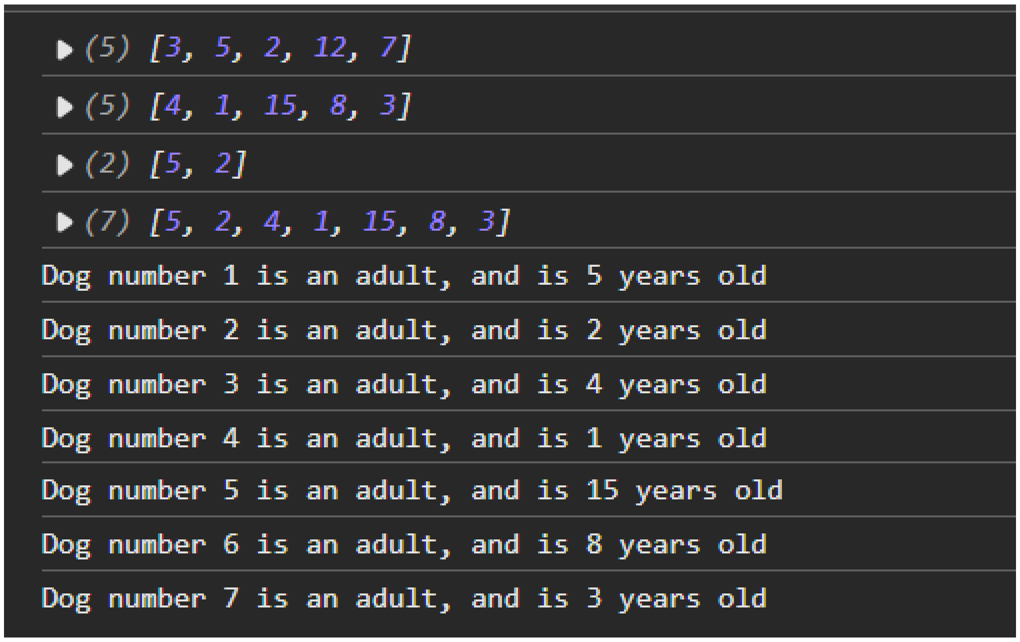
1. Julia found out that the owners of the first and the last two dogs actually have cats, not dogs! So create a shallow copy of Julia's array, and remove the cat ages from that copied array (because it's a bad practice to mutate function parameters)
2. Create an array with both Julia's (corrected) and Kate's data

3. For each remaining dog, log to the console whether it's an adult ("Dog number 1 is an adult, and is 5 years old") or a puppy ("Dog number 2 is still a puppy � ")

4. Run the function for both test datasets

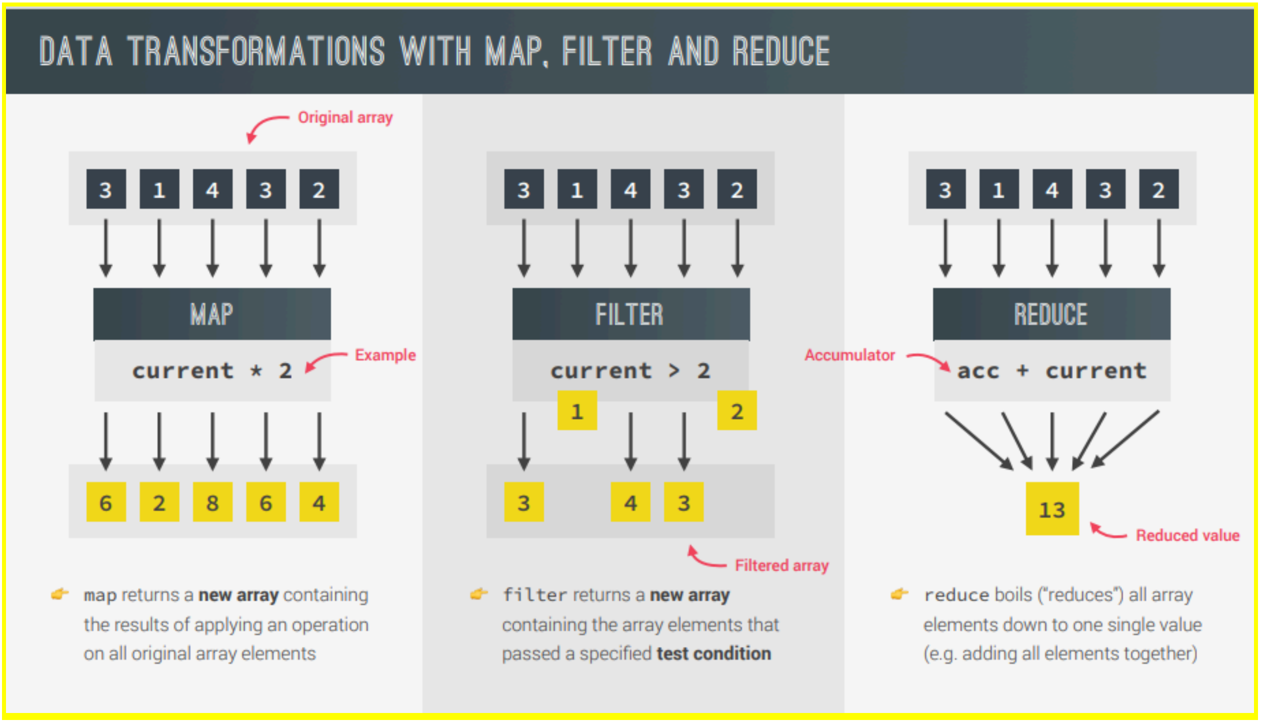
Test data:
Data 1: Julia's data [3, 5, 2, 12, 7], Kate's data [4, 1, 15, 8, 3]
Data 2: Julia's data [9, 16, 6, 8, 3], Kate's data [10, 5, 6, 1, 4]

```
//challenge
const dogsJulia = [3, 5, 2, 12, 7];
const dogsKate = [4, 1, 15, 8, 3];
console.log(dogsJulia);
console.log(dogsKate);
const checkDogs = function (dogsJulia, dogsKate) {
  const shallowdogsJulia = [...dogsJulia]; //slice()
  shallowdogsJulia.shift(); //shallowdogsJulia.splice(0, 1);
  shallowdogsJulia.splice(-2, 2);
  console.log(shallowdogsJulia);
  const dogs = shallowdogsJulia.concat(dogsKate); //[...shallowdogsJulia, ...dogsKate];
  console.log(dogs);
  //"Dog number 1 is an adult, and is 5 years old") or a puppy ("Dog number 2 is still a puppy �")
  dogs.forEach(function (dog, i, dogs) {
    let final = `${dog >= 3}`
      ? `Dog number ${i + 1} is an adult, and is ${dog} years old`
      : `Dog number ${i + 1} is still a puppy �`;

    console.log(final);
  });
};
checkDogs(dogsJulia, dogsKate);
```



```
▶ (5) [3, 5, 2, 12, 7]
▶ (5) [4, 1, 15, 8, 3]
▶ (2) [5, 2]
▶ (7) [5, 2, 4, 1, 15, 8, 3]
Dog number 1 is an adult, and is 5 years old
Dog number 2 is an adult, and is 2 years old
Dog number 3 is an adult, and is 4 years old
Dog number 4 is an adult, and is 1 years old
Dog number 5 is an adult, and is 15 years old
Dog number 6 is an adult, and is 8 years old
Dog number 7 is an adult, and is 3 years old
```

Array methods
Map,filter,reduce


DATA TRANSFORMATIONS WITH MAP, FILTER AND REDUCE
Original array
3 1 4 3 2
MAP
current * 2
Example
6 2 8 6 4
map returns a new array containing the results of applying an operation on all original array elements
3 1 4 3 2
FILTER
current > 2
1 2
3 4 3
Filtered array
filter returns a new array containing the array elements that passed a specified test condition
3 1 4 3 2
REDUCE
Accumulator
acc + current
13
Reduced value
reduce boils ("reduces") all array elements down to one single value (e.g. adding all elements together)

1) Map Method

It is similar to the forEach method but it creates and returns a new array with the results of applying the callback function to each element in the original array.

map: Used when you need to **transform** each element of an array and create a new array based on the transformation.
forEach: Used when you want to **perform side effects** like logging, updating variables, or modifying elements in place, without creating a new array.

Purpose: Creates a new array by applying a callback function to each element of the original array.
Key Points:

- Does not modify the original array.
- Returns a new array of the same length.

Syntax

```
const newArray = array.map(callback(element, index, array));
```

Ex1;

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

const doubleNumbers = numbers.map(function(num) {
  return num * 2;
});

console.log(doubleNumbers); // [2, 4, 6, 8]
```

Using Arrow Function:

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

const doubleNumbers = numbers.map(num => num * 2);

console.log(doubleNumbers); // [2, 4, 6, 8]
```

Ex2:

```
const names = ['alice', 'bob', 'charlie'];

// Normal Function
const uppercased = names.map(function(name) {
  return name.toUpperCase();
});

// Arrow Function
const uppercasedArrow = names.map(name => name.toUpperCase());

console.log(uppercased); // ['ALICE', 'BOB', 'CHARLIE']
console.log(uppercasedArrow); // ['ALICE', 'BOB', 'CHARLIE']
```

Ex3:

```
const numbers = [10, 20, 30, 40];

// Adding index to the number
const withIndex = numbers.map(function(value, index) {
  return `Index ${index}: ${value}`;
});
```

console.log(withIndex);
// ["Index 0: 10", "Index 1: 20", "Index 2: 30", "Index 3: 40"]

forEach and map

The return value from map is added to acc.username (which will create the username property in the respective user object and assign the return value from map), and this loop is iterating without returning a new array."

```
// Data
const account1 = {
  owner: 'Jonas Schmedtmann',
  movements: [200, 450, -400, 3000, -650, -130, 70, 1300],
  interestRate: 1.2, // %
  pin: 1111,
};

const account2 = {
  owner: 'Jessica Davis',
  movements: [5000, 3400, -150, -790, -3210, -1000, 8500, -30],
  interestRate: 1.5,
  pin: 2222,
};

const account3 = {
  owner: 'Steven Thomas Williams',
  movements: [200, -200, 340, -300, -20, 50, 400, -460],
  interestRate: 0.7,
  pin: 3333,
};

const account4 = {
  owner: 'Sarah Smith',
  movements: [430, 1000, 700, 50, 90],
  interestRate: 1,
  pin: 4444,
};

const accounts = [account1, account2, account3, account4];
const createUsernames = function (accs) {
  accs.forEach(function (acc) {
    acc.username = acc.owner
      .toLowerCase()
      .split(' ')
      .map(name => name[0])
      .join('');
  });
};
createUsernames(accounts);
```

Output

```
[
  {
    owner: 'Jonas Schmedtmann',
    movements: [200, 450, -400, 3000, -650, -130, 70, 1300],
    interestRate: 1.2,
    pin: 1111,
    username: 'js'
  },
  {
    owner: 'Jessica Davis',
    movements: [5000, 3400, -150, -790, -3210, -1000, 8500, -30],
    interestRate: 1.5,
    pin: 2222,
    username: 'jd'
  },
  {
    owner: 'Steven Thomas Williams',
    movements: [200, -200, 340, -300, -20, 50, 400, -460],
    interestRate: 0.7,
    pin: 3333,
    username: 'stw'
  },
  {
    owner: 'Sarah Smith',
    movements: [430, 1000, 700, 50, 90],
    interestRate: 1,
    pin: 4444,
    username: 'ss'
  }
]
```

Filter()

Purpose: Creates a new array containing only the elements that satisfy the condition specified in the callback function.
Key Points:

- Does not modify the original array.
- Returns a new array that may have fewer elements.

Syntax:

const filteredArray = array.filter(callback(element, index, array));

Ex1:

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

const filteredNumbers = numbers.filter((number, index, array) => {
  console.log(`Index: ${index}, Number: ${number}`);
  return number > 3;
});

console.log(filteredNumbers); // Output: [4, 5, 6]
```

Reduce()

The reduce() method executes a reducer function for array elements.

The reduce() method returns a single value: the function's accumulated result.

The reduce() method does not execute the function for empty array elements.

The reduce() method does not change the original array.

The reduce() method in JavaScript is used to apply a function to an accumulator and each element in the array (from left to right) to reduce it to a single value. It does not modify the original array. The syntax for the reduce() method is:

array.reduce(callback(accumulator, currentValue, index, array), initialValue)

- accumulator: The accumulated result from the previous iteration (or initialValue on the first iteration).
- currentValue: The current element being processed.
- index (optional): The index of the current element.
- array (optional): The array that reduce() was called upon.

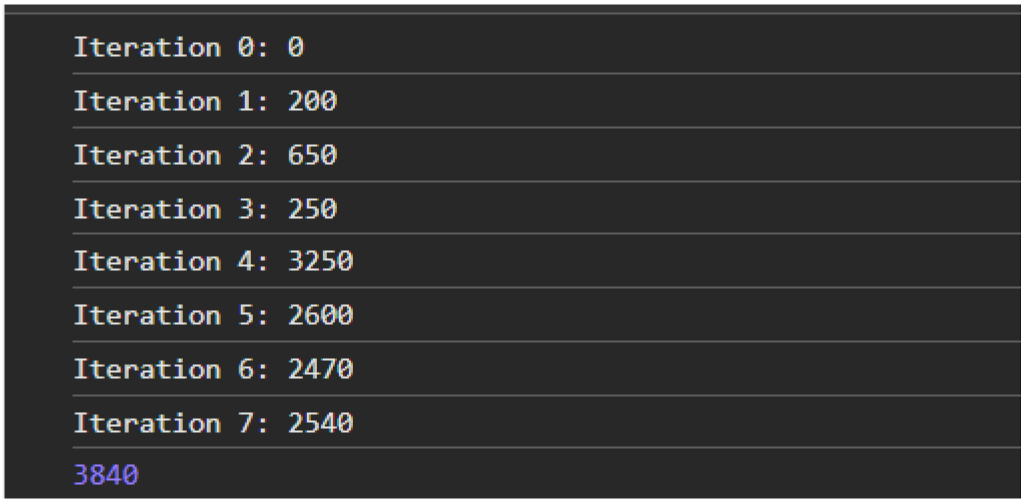
initialValue (optional): A value to start the accumulator. If omitted, the first element of the array is used as the initial value.

Ex

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

const sum = numbers.reduce(function(accumulator, currentValue) {

  return accumulator + currentValue;

}, 0); //0 is the initial value of accumulator



console.log(sum); // Output: 15
```

Ex2:

```
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

const balance = movements.reduce(function (acc, cur, i, arr) {

  console.log(`Iteration ${i}: ${acc}`);

  return acc + cur;

}, 0); // 0 is the initial accumulator value

console.log(balance);
```


```
Iteration 0: 0
Iteration 1: 200
Iteration 2: 650
Iteration 3: 250
Iteration 4: 3250
Iteration 5: 2600
Iteration 6: 2470
Iteration 7: 2540
3840
```

Arrow function

```
//arrow function



const balanc = movements.reduce((acc, cur, i) => {

  console.log(`Iteration ${i}: ${acc}`);

  return acc + cur;

}, 0);
```

Using for..of

```
//using for..of

let sum = 0;

for (const mov of movements) {

  sum = sum + mov;

}
```

```
console.log(sum);
```

Challenge

Let's go back to Julia and Kate's study about dogs. This time, they want to convert dog ages to human ages and calculate the average age of the dogs in their study.

Your tasks:

Create a function 'calcAverageHumanAge', which accepts an arrays of dog's ages ('ages'), and does the following things in order:

1. Calculate the dog age in human years using the following formula: if the dog is <= 2 years old, humanAge = 2 * dogAge. If the dog is > 2 years old, humanAge = 16 + dogAge * 4

2. Exclude all dogs that are less than 18 human years old (which is the same as keeping dogs that are at least 18 years old)

3. Calculate the average human age of all adult dogs (you should already know from other challenges how we calculate averages �)

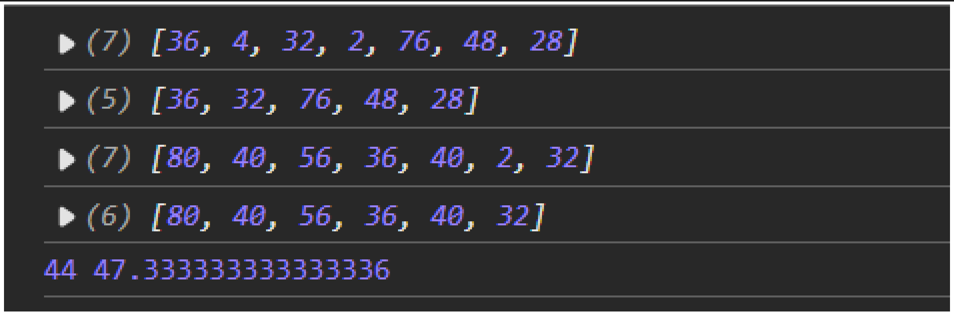
4. Run the function for both test datasets

Test data:

Data 1: [5, 2, 4, 1, 15, 8, 3]

Data 2: [16, 6, 10, 5, 6, 1, 4]

```
//Challenge 2

const calcAverageHumanAge = function (ages) {

 const humanAges = ages.map(age => (age <= 2 ? 2 * age : 16 + age * 4));

 console.log(humanAges);

 const adults = humanAges.filter(age => age >= 18);
 console.log(adults);

 const average = adults.reduce((acc, age) => acc + age / adults.length, 0);

 return average;

};

const avg1 = calcAverageHumanAge([5, 2, 4, 1, 15, 8, 3]); //[36, 4, 32, 2, 76, 48, 28]

const avg2 = calcAverageHumanAge([16, 6, 10, 5, 6, 1, 4]);

console.log(avg1, avg2);
```



```
▶ (7) [36, 4, 32, 2, 76, 48, 28]
▶ (5) [36, 32, 76, 48, 28]
▶ (7) [80, 40, 56, 36, 40, 2, 32]
▶ (6) [80, 40, 56, 36, 40, 32]
44 47.333333333333336
```

Changing Methods all at a time(map,filter, reduce):

```
const calcDisplaySummary = function (acc) {

 const incomes = acc.movements

  .filter(mov => mov > 0)

  .reduce((acc, mov) => acc + mov, 0);

 labelSumIn.textContent = `${incomes}€`;

 const out = acc.movements

  .filter(mov => mov < 0)

  .reduce((acc, mov) => acc + mov, 0);

 labelSumOut.textContent = `${Math.abs(out)}€`;
 const interest = acc.movements

  .filter(mov => mov > 0)

  .map(deposit => (deposit * acc.interestRate) / 100)

  .filter((int, i, arr) => {

    // console.log(arr);

    return int >= 1;

  })

  .reduce((acc, int) => acc + int, 0);

 labelSumInterest.textContent = `${interest}€`;

};
```

Coding challenge

Rewrite the 'calcAverageHumanAge' function from Challenge #2, but this time as an arrow function, and using chaining!

Test data:

Data 1: [5, 2, 4, 1, 15, 8, 3]

Data 2: [16, 6, 10, 5, 6, 1, 4]

```
//challenge 3

const calcAverageHumanAge = ages =>

 ages

   .map(age => (age <= 2 ? 2 * age : 16 + age * 4))
```

```
  .filter(age => age >= 18)

  .reduce((acc, age, i, arr) => acc + age / arr.length, 0);

const avg1 = calcAverageHumanAge([5, 2, 4, 1, 15, 8, 3]); //[36, 4, 32, 2, 76, 48, 28]

const avg2 = calcAverageHumanAge([16, 6, 10, 5, 6, 1, 4]);

console.log(avg1, avg2); //44 47.333333333333336
```

Find () Method

The find() method in JavaScript is an array method used to return the **first element** in an array that satisfies a provided **testing function**. If no element satisfies the condition, it returns undefined.

`array.find(function(currentValue, index, arr))`

Return Value

- The **first element** in the array that satisfies the condition in the callback.
- Returns undefined if no elements match.

Other find methods

- If you need the index of the found element in the array, use findIndex().
- If you need to find the index of a value, use indexOf(). (It's similar to findIndex(), but checks each element for equality with the value instead of using a testing function.)
- If you need to find if a value exists in an array, use includes(). Again, it checks each element for equality with the value instead of using a testing function.

```
// Input array contains some elements.

let array = [10, 20, 30, 40, 50];

// Method (return element > 10).

let found = array.find(function (element) {
    return element > 20;
});
// Printing desired values.
console.log(found); // 30
```

Ex2:

```
const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Charlie' }
];
const user = users.find(user => user.name === 'Bob');
console.log(user); // Output: { id: 2, name: 'Bob' }
```

- **findIndex().**

The findIndex() method in JavaScript is used to find the index of the first element in an array that satisfies a provided testing function. If no element satisfies the testing function, it returns -1.

findIndex() does not mutate the original array. It stops iterating as soon as it finds the first match.

Useful for identifying elements when their position in the array is important.

```
array.findindex(function(currentValue, index, arr))
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }

];

// Find the index of the user with name 'Bob'

const index = users.findIndex(user => user.name === 'Bob');

console.log(index); // Output: 1
```

Finding the First VIP Customer (examples for find() and findindex())

You have a list of customers, and you want to find the first VIP customer and their index.

```
const customers = [

  { id: 1, name: 'Alice', vip: false },

  { id: 2, name: 'Bob', vip: true },

  { id: 3, name: 'Charlie', vip: false },

  { id: 4, name: 'Dave', vip: true },

];

// Find the first VIP customer

const firstVIP = customers.find(customer => customer.vip === true);

console.log(firstVIP);

// Output: { id: 2, name: 'Bob', vip: true }
// Find the index of the first VIP customer

const firstVIPIndex = customers.findIndex(customer => customer.vip === true);

console.log(firstVIPIndex);

// Output: 1
```

Ex2: **Finding the First Adult in a List**

You have a list of people with ages, and you want to find the first adult (age 18 or older).

```
const people = [

  { name: 'Alice', age: 16 },

  { name: 'Bob', age: 20 },

  { name: 'Charlie', age: 15 },

  { name: 'Dave', age: 30 },

];

// Find the first adult

const firstAdult = people.find(person => person.age >= 18);

console.log(firstAdult);

// Output: { name: 'Bob', age: 20 }

// Find the index of the first adult
```

```
const firstAdultIndex = people.findIndex(person => person.age >= 18);


console.log(firstAdultIndex);


// Output: 1
```

findLast()and findLastindex()

In JavaScript, findLast and findLastIndex are methods used to find the last matching element or its index in an array, starting from the end of the array and moving backward. These methods were introduced in ECMAScript 2023 (ES14).

findLast (//like last withdraw ,money type of conditions)

The findLast() method returns the last element in an array that satisfies the provided testing function. If no element matches, it returns undefined.

Syntax:
array.findLast(callback(element, index, array), thisArg);

findLastIndex

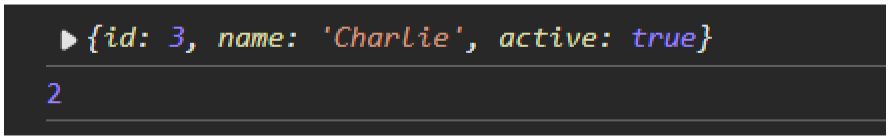
The findLastIndex() method returns the **index of the last element** in an array that satisfies the provided testing function. If no element matches, it returns -1.

EX: Finding the Last Active User

In an array of users, you want to find the last active user and their position.

```
const users = [
  { id: 1, name: 'Alice', active: true },
  { id: 2, name: 'Bob', active: false },
  { id: 3, name: 'Charlie', active: true },
  { id: 4, name: 'Dave', active: false },
];

// Find the last active user
const lastActiveUser = users.findLast(user => user.active === true);
```

```
console.log(lastActiveUser);
// Output: { id: 3, name: 'Charlie', active: true }

// Find the index of the last active user
const lastActiveUserIndex = users.findLastIndex(user => user.active === true);
console.log(lastActiveUserIndex);
// Output: 2
```



```
▶ {id: 3, name: 'Charlie', active: true}
2
```

Ex2: **Finding the Last Login from a Specific Country**

If you have user login records, you may want to identify the last login attempt from a specific country (e.g., USA).

```
const logins = [
  { user: 'Alice', country: 'UK' },
  { user: 'Bob', country: 'USA' },
  { user: 'Charlie', country: 'Canada' },
  { user: 'Dave', country: 'USA' },
];

// Find the last login from the USA
const lastUSALogin = logins.findLast(login => login.country === 'USA');
console.log(lastUSALogin);
// Output: { user: 'Dave', country: 'USA' }

// Find the index of the last login from the USA
const lastUSALoginIndex = logins.findLastIndex(
  login => login.country === 'USA'
);
console.log(lastUSALoginIndex);
// Output: 3
```

Math.abs()

The Math.abs() method in JavaScript returns the **absolute value** of a number. The absolute value of a number is its **distance from zero**, meaning it is always non-negative.

- If the input is:
  - A **positive number** or **zero**, it returns the number itself.
  - A **negative number**, it returns its positive counterpart.
  - NaN or a non-numeric value, it returns NaN.

```
console.log(Math.abs(10)); // Output: 10
console.log(Math.abs(-10)); // Output: 10
console.log(Math.abs(0)); // Output: 0
console.log(Math.abs(-0)); // Output: 0
console.log(Math.abs(5.5)); // Output: 5.5
console.log(Math.abs(-5.5)); // Output: 5.5
console.log(Math.abs('5')); // Output: 5 (string is converted to number)
console.log(Math.abs('-5')); // Output: 5
console.log(Math.abs('hello')); // Output: NaN (cannot convert 'hello' to a number)
console.log(Math.abs()); // Output: NaN (no argument passed)
```

Distance Between Two Numbers

You can use Math.abs() to calculate the distance between two numbers.

```
const a = 5;
const b = 15;
const distance = Math.abs(a - b);
console.log(distance); // Output: 10
```

Calculating Net Profit or Loss

Imagine you are calculating the **net profit or loss** for a business over several transactions. You need to:

1. Handle negative values as losses and calculate the absolute magnitude of losses.
2. Ensure the final profit or loss is displayed as a non-negative distance from zero.

```
const transactions = [200, -150, 300, -50, -100]; // Positive = profit, Negative = loss
```

```
// Calculate the total net amount
const netAmount = transactions.reduce((total, value) => total + value, 0);
console.log('Net Amount:', netAmount); // Output: 200 (net profit)

// Calculate the absolute net amount (distance from zero)
const absoluteNetAmount = Math.abs(netAmount);
console.log('Absolute Net Amount:', absoluteNetAmount); // Output: 200

// Display whether it's a profit or loss
if (netAmount > 0) {
  console.log(`Net Profit: ${absoluteNetAmount}`); // Output: Net Profit: 200
} else if (netAmount < 0) {
  console.log(`Net Loss: ${absoluteNetAmount}`);
} else {
  console.log('No Profit, No Loss');
}

// Calculate the absolute values of each transaction
const absoluteTransactions = transactions.map(value => Math.abs(value));
console.log('Absolute Transactions:', absoluteTransactions);
// Output: [200, 150, 300, 50, 100]
```

Some and every methods

The some and every methods in JavaScript are array methods used to test elements in an array based on a provided condition. These methods return a boolean (true or false) based on whether the condition is satisfied.

Array.prototype.some()

The some method tests whether **at least one element** in the array passes the provided condition (callback function). If any element satisfies the condition, it returns true. Otherwise, it returns false.

Stops iterating as soon as it finds the first element that satisfies the condition.

Returns true if at least one element passes; otherwise false.

Check if an array has any even numbers

```
const numbers = [1, 3, 5, 7, 8];

const hasEven = numbers.some((num) => num % 2 === 0);

console.log(hasEven); // Output: true (because 8 is even)
```

Example: Check if any student scored above 90

```
const students = [

  { name: "Alice", score: 85 },

  { name: "Bob", score: 78 },

  { name: "Charlie", score: 92 }

];

const hasHighScorer = students.some((student) => student.score > 90);

console.log(hasHighScorer); // Output: true (because Charlie scored 92)
```

Array.prototype.every()

The every method tests whether **all elements** in the array pass the provided condition (callback function). If all elements satisfy the condition, it returns true. Otherwise, it returns false.

Stops iterating as soon as it finds the first element that does not satisfy the condition.

Returns true if all elements pass the condition; otherwise false.

Check if all numbers are positive

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

const allPositive = numbers.every((num) => num > 0);

console.log(allPositive); // Output: true (all numbers are positive)
```

Example: Check if all students passed (score >= 50)

```
const students = [

  { name: "Alice", score: 85 },

  { name: "Bob", score: 78 },

  { name: "Charlie", score: 45 }

];

const allPassed = students.every((student) => student.score >= 50);

console.log(allPassed); // Output: false (Charlie scored 45)
```

flat and flatMap

The flat() method of Array instances creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

The flat method creates a new array by flattening nested arrays. It removes nested levels of arrays up to the specified depth.

Syntax

```
array.flat([depth]);
```

- depth *(optional)*: Specifies how deep the flattening should go. The default is 1.

The depth value is 1 by default and you can leave it empty. The depth value takes a number as its data type. array.flat(1) is equal to array.flat().

flat is useful when dealing with deeply nested arrays, e.g., combining data from multiple APIs.

```
const arr1 = [0, 1, 2, [3, 4]];

console.log(arr1.flat());

// expected output: Array [0, 1, 2, 3, 4]

const arr2 = [0, 1, [2, [3, [4, 5]]]];

console.log(arr2.flat());
```

```
// expected output: Array [0, 1, 2, Array [3, Array [4, 5]]]

console.log(arr2.flat(2));

// expected output: Array [0, 1, 2, 3, Array [4, 5]]

//Completely Flattening

console.log(arr2.flat(Infinity));

// expected output: Array [0, 1, 2, 3, 4, 5]
```

flatMap Method

The flatMap method first maps each element using a callback function, then flattens the result into a new array. It only flattens by one level, regardless of the depth.

flatMap is ideal when transforming and flattening data in a single operation, e.g., tokenizing sentences into words.

```
const arr1 = [1, 2, 1];

const result = arr1.flatMap(num => (num === 2 ? [2, 2] : 1));

console.log(result);

// Expected output: Array [1, 2, 2, 1]
```

Difference Between map and flatMap

Feature	`map`	`flatMap`
Primary Use	Transforms elements of an array using a callback.	Combines mapping and flattening into one operation.
Output	Always returns an array of the same length as the input.	Returns a flattened array (only one level deep).
Flattening	Does **not** flatten nested arrays.	Flattens the resulting array **by one level only**.

```
//Using map would result in nested arrays:

const numbers = [1, 2, 3];

const mapped = numbers.map(num => [num, num * 2]);

console.log(mapped);

// Output: [[1, 2], [2, 4], [3, 6]]

//Using flatMap flattens the result:

const flatMapped = numbers.flatMap(num => [num, num * 2]);

console.log(flatMapped);

// Output: [1, 2, 2, 4, 3, 6]
```

Splitting Strings

```
//Using map

const phrases = ['hello world', 'flatMap example'];

const mapped = phrases.map(phrase => phrase.split(' '));

console.log(mapped);
```

```
// Output: [["hello", "world"], ["flatMap", "example"]] (nested array)

//Using flatMap

const phrases1 = ['hello world', 'flatMap example'];

const flatMapped = phrases1.flatMap(phrase => phrase.split(' '));

console.log(flatMapped);

// Output: ["hello", "world", "flatMap", "example"] (flattened array)
```

Sorting Arrays

The sort() method of Array instances sorts the elements of an array *in place* and returns the reference to the same array, now sorted. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code unit values.

```
const months = ['March', 'Jan', 'Feb', 'Dec'];

months.sort();

console.log(months);

// Expected output: Array ["Dec", "Feb", "Jan", "March"]

const array1 = [1, 30, 4, 21, 100000];

array1.sort();// sorted by  converting into string , fo numerical value we need to use other way

console.log(array1);

// Expected output: Array [1, 100000, 21, 30, 4]
```

By default, sort() converts elements to strings and sorts them lexicographically, which may not be desirable for numbers. Use a comparator function for numeric sorting.

```
//Ascending Order:
```

```
let numbers = [10, 2, 30, 5];

numbers.sort((a, b) => a - b);

console.log(numbers); // Output: [2, 5, 10, 30]

//Descending Order:

let numbers1 = [10, 2, 30, 5];

numbers1.sort((a, b) => b - a);

console.log(numbers1); // Output: [30, 10, 5, 2]
```

Sort Array of Strings in Reverse Order

We can use the string.localeCompare() method in the comparator function to sort in reverse order.

```
let a = ["JS", "HTML", "CSS"];

console.log(a);

a.sort((x, y) => x.localeCompare(y))

console.log(a);
```

Output

```
[ 'JS', 'HTML', 'CSS' ]

[ 'CSS', 'HTML', 'JS' ]
```

We can also use the array.reverse() method to sort the array in descending order.

```
let a = ["JS", "CSS", "HTML"];

a.sort();
```

```
a.reverse();

console.log(a);
```

Output

```
[ 'JS', 'HTML', 'CSS' ]
```

Sorting Array of Objects

The array of objects can be sorted on the basis of property values.

```
// Array of a objects with different names and ages

let a = [

  { name: 'Rahul', age: 28 },

  { name: 'Jatin', age: 25 },

  { name: 'Vikas', age: 32 },

  { name: 'Rohit', age: 35 }

];

// Sort the objects for age

a.sort((x, y) => x.age - y.age);

console.log(a);

// Sort object for names

a.sort((x, y) => x.name.localeCompare(y.name));

console.log(a);
```

Output

```
[

  { name: 'Jatin', age: 25 },
  { name: 'Rahul', age: 28 },

  { name: 'Vikas', age: 32 },

  { name: 'Rohit', age: 35 }

]

[

  { name: 'Jatin', age: 25 },

  { name: 'Rahul', age: 28 },

  { name: 'Rohi...
```

Array Grouping

Array grouping in JavaScript refers to the process of organizing or grouping elements of an array based on some criteria. It's commonly used for data transformation to group similar items together.

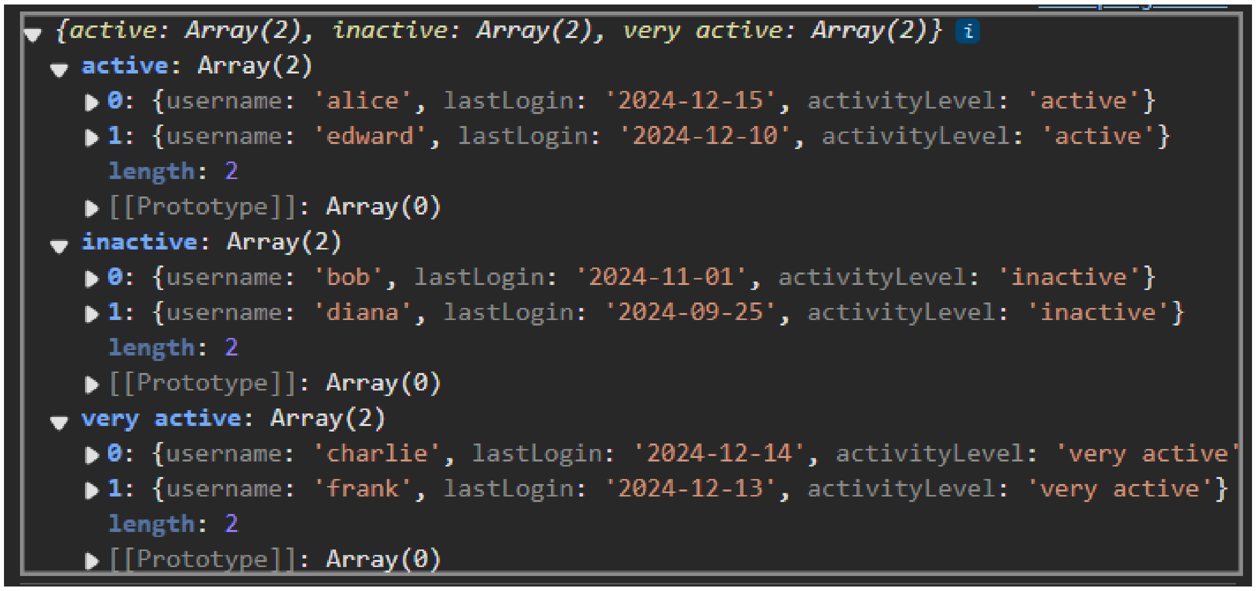
Syntax

Object.groupBy(items, callbackFn)

Grouping Numbers by Odd or Even

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

const groupedByOddEven = numbers.groupBy(num => (num % 2 === 0 ? 'even' : 'odd'));

console.log(groupedByOddEven);

// Output:

// {

//   odd: [1, 3, 5],

//   even: [2, 4, 6]
// }
```

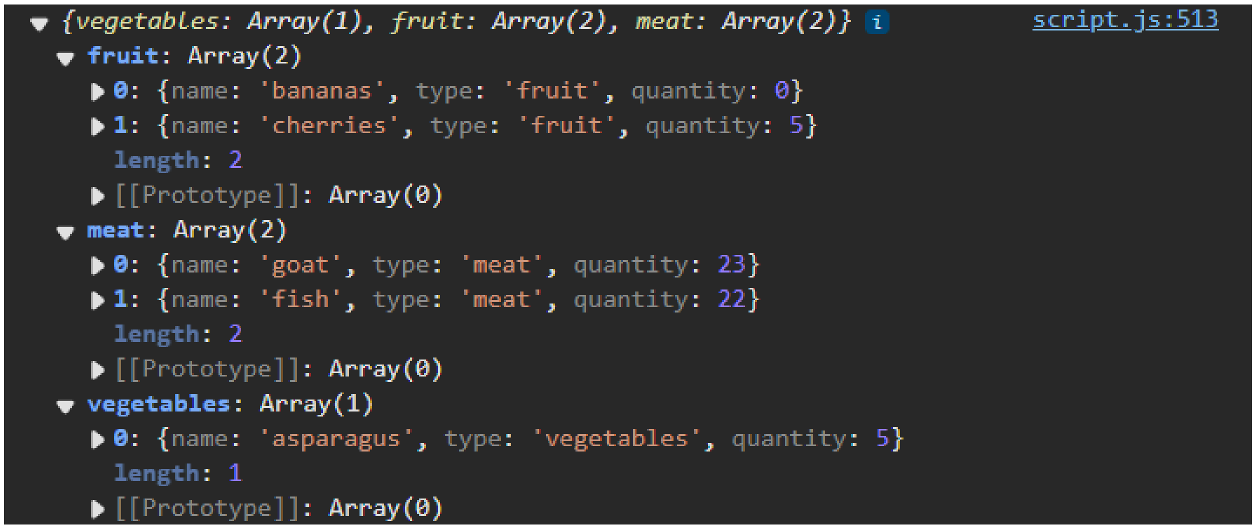
```
const accountsDetails = [
  { username: 'alice', lastLogin: '2024-12-15', activityLevel: 'active' },
  { username: 'bob', lastLogin: '2024-11-01', activityLevel: 'inactive' },
  {
    username: 'charlie',
    lastLogin: '2024-12-14',
    activityLevel: 'very active',
  },
  { username: 'diana', lastLogin: '2024-09-25', activityLevel: 'inactive' },
  { username: 'edward', lastLogin: '2024-12-10', activityLevel: 'active' },
  { username: 'frank', lastLogin: '2024-12-13', activityLevel: 'very active' },
];
// Grouping accounts by their activity level
const groupedByActivity = Object.groupBy(
  accountsDetails,
  acc => acc.activityLevel
);
console.log(groupedByActivity);
```



```
▼ {active: Array(2), inactive: Array(2), very active: Array(2)} i
  ▼ active: Array(2)
    ▶ 0: {username: 'alice', lastLogin: '2024-12-15', activityLevel: 'active'}
    ▶ 1: {username: 'edward', lastLogin: '2024-12-10', activityLevel: 'active'}
      length: 2
    ▶ [[Prototype]]: Array(0)
  ▼ inactive: Array(2)
    ▶ 0: {username: 'bob', lastLogin: '2024-11-01', activityLevel: 'inactive'}
    ▶ 1: {username: 'diana', lastLogin: '2024-09-25', activityLevel: 'inactive'}
      length: 2
    ▶ [[Prototype]]: Array(0)
  ▼ very active: Array(2)
    ▶ 0: {username: 'charlie', lastLogin: '2024-12-14', activityLevel: 'very active'
    ▶ 1: {username: 'frank', lastLogin: '2024-12-13', activityLevel: 'very active'}
      length: 2
    ▶ [[Prototype]]: Array(0)
```

Ex2:

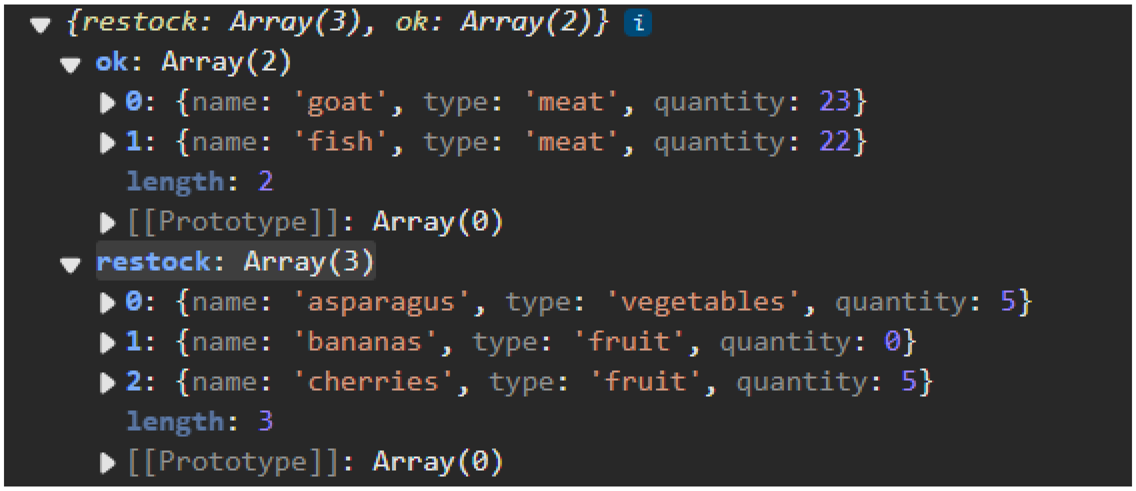
```
const inventory = [

 { name: 'asparagus', type: 'vegetables', quantity: 5 },

 { name: 'bananas', type: 'fruit', quantity: 0 },

 { name: 'goat', type: 'meat', quantity: 23 },

 { name: 'cherries', type: 'fruit', quantity: 5 },

 { name: 'fish', type: 'meat', quantity: 22 },

];

const group = Object.groupBy(inventory, function (inv) {

 return inv.type;

});

console.log(group);
```



```
▼ {vegetables: Array(1), fruit: Array(2), meat: Array(2)} i          script.js:513
  ▼ fruit: Array(2)
    ▶ 0: {name: 'bananas', type: 'fruit', quantity: 0}
    ▶ 1: {name: 'cherries', type: 'fruit', quantity: 5}
      length: 2
    ▶ [[Prototype]]: Array(0)
  ▼ meat: Array(2)
    ▶ 0: {name: 'goat', type: 'meat', quantity: 23}
    ▶ 1: {name: 'fish', type: 'meat', quantity: 22}
      length: 2
    ▶ [[Prototype]]: Array(0)
  ▼ vegetables: Array(1)
    ▶ 0: {name: 'asparagus', type: 'vegetables', quantity: 5}
      length: 1
    ▶ [[Prototype]]: Array(0)
```

Ex3

```
const inventory = [

 { name: 'asparagus', type: 'vegetables', quantity: 5 },

 { name: 'bananas', type: 'fruit', quantity: 0 },

 { name: 'goat', type: 'meat', quantity: 23 },

 { name: 'cherries', type: 'fruit', quantity: 5 },

 { name: 'fish', type: 'meat', quantity: 22 },

];

function myCallback({ quantity }) {

 return quantity > 5 ? 'ok' : 'restock';

}

const result2 = Object.groupBy(inventory, myCallback);

console.log(result2);
```



```
▼ {restock: Array(3), ok: Array(2)} i
  ▼ ok: Array(2)
    ▶ 0: {name: 'goat', type: 'meat', quantity: 23}
    ▶ 1: {name: 'fish', type: 'meat', quantity: 22}
      length: 2
    ▶ [[Prototype]]: Array(0)
  ▼ restock: Array(3)
    ▶ 0: {name: 'asparagus', type: 'vegetables', quantity: 5}
    ▶ 1: {name: 'bananas', type: 'fruit', quantity: 0}
    ▶ 2: {name: 'cherries', type: 'fruit', quantity: 5}
      length: 3
    ▶ [[Prototype]]: Array(0)
```

More Ways to create and filling Arrays

i) Using Array Literals

The simplest and most common way to create an array is by using array literals.

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

console.log(arr); //[1, 2, 3, 4, 5]

const array = [5, 6, 7, 8];

console.log(array); //[5, 6, 7, 8]
```

ii) Using the Array Constructor

You can use the Array constructor to create arrays with a specified length or predefined elements.

```
const arr1 = new Array(5);

console.log(arr1); // [empty × 5]

let arr2 = new Array(1, 2, 3, 4); // creates an array with specified elements

console.log(arr2); //[1, 2, 3, 4]

console.log(new Array(3, 4, 5, 6, 7)); //[3, 4, 5, 6, 7]
```

iii) Using Array.of()

The Array.of() method creates a new array instance with a variable number of elements.

```
const arr = Array.of(10, 20, 30); // creates an array [10, 20, 30]

console.log(arr);// [10, 20, 30]
```

iv) Using Array.from()

The Array.from() method creates a new array from an array-like or iterable object.

syntax

Array.from(arrayLike)

Array.from(arrayLike, mapFn)

Array.from(arrayLike, mapFn, thisArg)

convert a string into an array of characters.

```
let str = 'ramesh';

console.log(str); //ramesh

let array = Array.from(str);

console.log(array);//['r', 'a', 'm', 'e', 's'
```

convert a Set (which contains unique values) into an array.

```
const mySet = new Set([1, 2, 3, 4]);

console.log(mySet); //{1, 2, 3, 4}

const arr = Array.from(mySet);

console.log(arr); // [1, 2, 3, 4]
```

Create an Array with a Range of Numbers

```
const arr = Array.from({ length: 5 }, (v, i) => i + 1);

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

Array.prototype.fill()

The Array.prototype.fill() method in JavaScript is used to fill all the elements in an array with a static value, from a specified start index to an optional end index. The original array is modified in place.

Syntax:

```
arr.fill(value, startIndex, endIndex);
```

value: The value to fill the array with.

startIndex (optional): The index at which to start filling the array (default is 0).

endIndex (optional): The index at which to stop filling the array (default is the array length).

Filling an entire array:

```
const arr = new Array(4);

console.log(arr); // [empty × 4]

const filledArr = arr.fill(6);

console.log(filledArr); // [6, 6, 6, 6]
```

Filling from a specific start index:

```
const arr = new Array(4);

console.log(arr); // [empty × 4]

const filledArr = arr.fill(6, 2);
```

```
console.log(filledArr); // [empty × 2, 6, 6]
```

```
let arr1 = [1, 2, 3, 4, 5];

arr1.fill(0, 2);

console.log(arr1); // [1, 2, 0, 0, 0]
```

Filling within a range (start index to end index):

```
let arr1 = [1, 2, 3, 4, 5];

arr1.fill(0, 2, 4);

console.log(arr1); //[1, 2, 0, 0, 5]

let arr2 = [1, 2, 3, 4, 5, 9, 8];

arr2.fill(0, -3, -1);

console.log(arr2); //  [1, 2, 3, 4, 0, 0, 8]
```

Non-Destructive Alternatives : toReversed , toSorted , toSpliced ,with

Array.prototype.toReversed():

Array.prototype.toReversed() is a proposed method for JavaScript that would create a new array with the elements of the original array reversed, without modifying the original array

array.toReversed();

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

const reversedArr = arr.toReversed();

console.log(reversedArr); // [4, 3, 2, 1]

console.log(arr); // [1, 2, 3, 4] (original array is unchanged)
```

Array.prototype.toSorted()

The Array.prototype.toSorted() method in JavaScript is a new method introduced in ECMAScript 2024. It creates a **new array** that is a sorted copy of the original array without modifying the original array.

Syntax

toSorted()

toSorted(compareFn)

```
// Example with numbers

const numbers = [3, 1, 4, 1, 5, 9];

const sortedNumbers = numbers.toSorted();

console.log(sortedNumbers); // [1, 1, 3, 4, 5, 9]

console.log(numbers); // [3, 1, 4, 1, 5, 9] (original array is unchanged)



// Example with a custom compare function (descending order)

const sortedNumbersDesc = numbers.toSorted((a, b) => b - a);

console.log(sortedNumbersDesc); // [9, 5, 4, 3, 1, 1]

console.log(numbers); // [3, 1, 4, 1, 5, 9] (original array is unchanged)
```

Array.prototype.with()

Array.prototype.with() is a non-standard method that was proposed in an early version of ECMAScript but has not been included in the official JavaScript specification (ECMA-262).

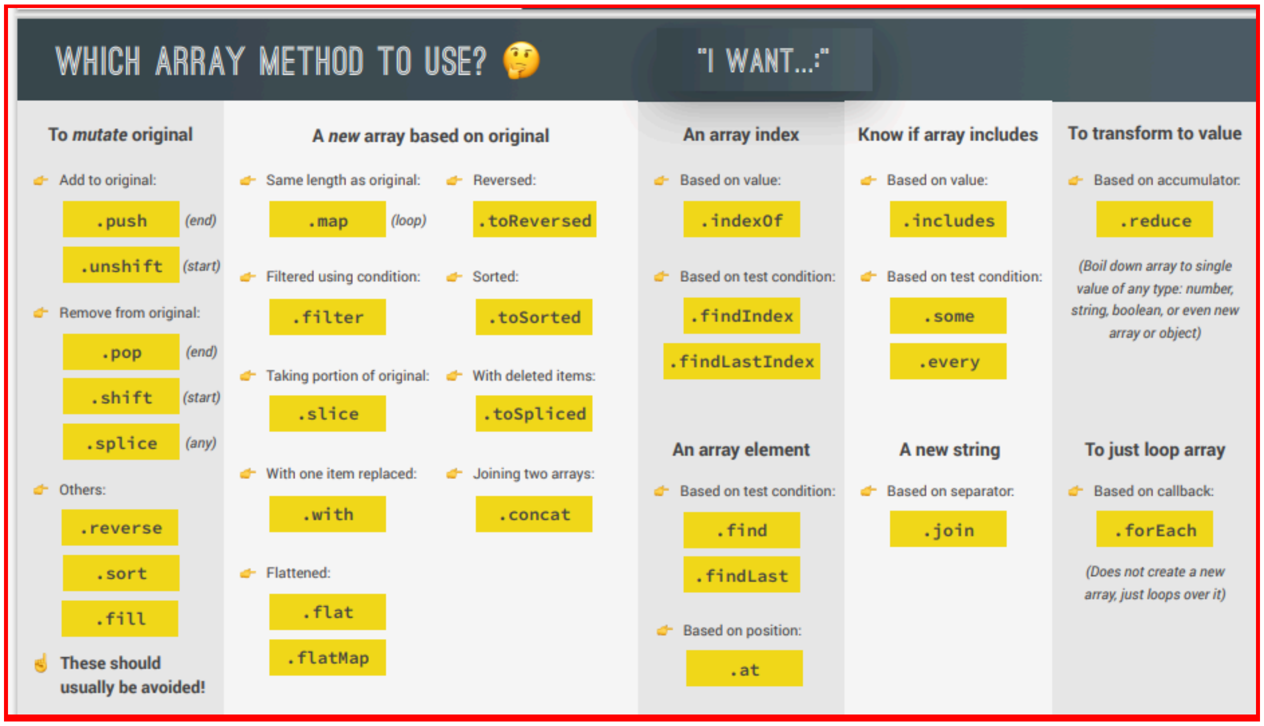
The with() method updates a specified array element.

The with() method returns a new array.

The with() method does not change the original array.

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

// Assuming Array.prototype.with() existed:

let newArr = arr.with(2, 10); // Replace the value at index 2 with 10

console.log(newArr); // Output: [1, 2, 10, 4, 5]

console.log(arr); // Output: [1, 2, 3, 4, 5] (original array remains unchanged)
```

Which Array Method to Use



WHICH ARRAY METHOD TO USE? 🤔 "I WANT...:"

To *mutate* original

👉 Add to original:
- `.push` *(end)*
- `.unshift` *(start)*

👉 Remove from original:
- `.pop` *(end)*
- `.shift` *(start)*
- `.splice` *(any)*

👉 Others:
- `.reverse`
- `.sort`
- `.fill`

☝ **These should usually be avoided!**

A *new* array based on original

👉 Same length as original:
- `.map` *(loop)*

👉 Filtered using condition:
- `.filter`

👉 Taking portion of original:
- `.slice`

👉 With one item replaced:
- `.with`

👉 Flattened:
- `.flat`
- `.flatMap`

👉 Reversed:
- `.toReversed`

👉 Sorted:
- `.toSorted`

👉 With deleted items:
- `.toSpliced`

👉 Joining two arrays:
- `.concat`

An array index

👉 Based on value:
- `.indexOf`

👉 Based on test condition:
- `.findIndex`
- `.findLastIndex`

An array element

👉 Based on test condition:
- `.find`
- `.findLast`

👉 Based on position:
- `.at`

Know if array includes

👉 Based on value:
- `.includes`

👉 Based on test condition:
- `.some`
- `.every`

A new string

👉 Based on separator:
- `.join`

To transform to value

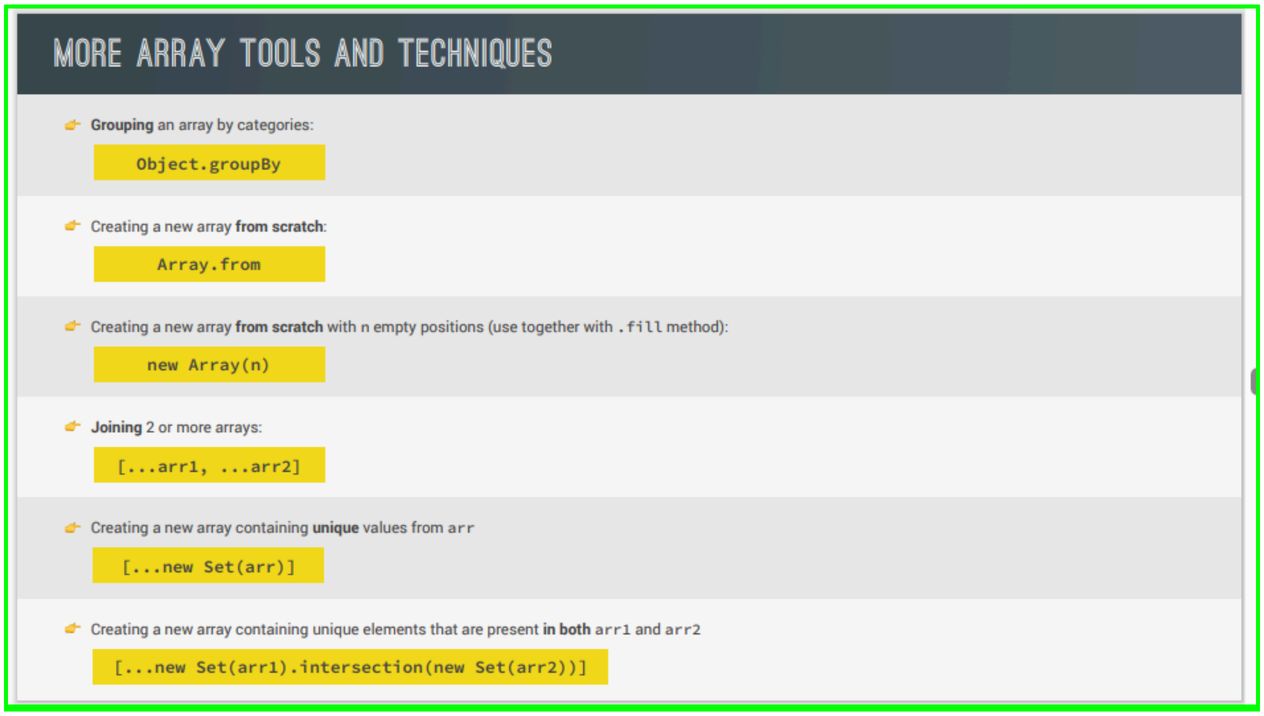
👉 Based on accumulator:
- `.reduce`

(Boil down array to single value of any type: number, string, boolean, or even new array or object)

To just loop array

👉 Based on callback:
- `.forEach`

(Does not create a new array, just loops over it)



MORE ARRAY TOOLS AND TECHNIQUES

- **Grouping** an array by categories:

```
Object.groupBy
```

- Creating a new array **from scratch**:

```
Array.from
```

- Creating a new array **from scratch** with n empty positions (use together with `.fill` method):

```
new Array(n)
```

- **Joining** 2 or more arrays:

```
[...arr1, ...arr2]
```

- Creating a new array containing **unique** values from `arr`

```
[...new Set(arr)]
```

- Creating a new array containing unique elements that are present **in both** `arr1` and `arr2`

```
[...new Set(arr1).intersection(new Set(arr2))]
```

preventDefault()

The preventDefault() method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur.

For example, this can be useful when:

- Clicking on a "Submit" button, prevent it from submitting a form
- Clicking on a link, prevent the link from following the URL

Note: Not all events are cancelable. Use the cancelable property to find out if an event is cancelable.

Note: The preventDefault() method does not prevent further propagation of an event through the DOM. Use the stopPropagation() method to handle this.

Coding Challenge

Julia and Kate are still studying dogs, and this time they are studying if dogs are eating too much or too little. Eating too much means the dog's current food portion is larger than the recommended portion, and eating too little is the opposite. Eating an okay amount means the dog's current food portion is within a range 10% above and 10% below the recommended portion (see hint).

Your tasks:

1. Loop over the 'dogs' array containing dog objects, and for each dog, calculate the recommended food portion and add it to the object as a new property. Do not create a new array, simply loop over the array. Forumla: recommendedFood = weight ** 0.75 * 28. (The result is in grams of food, and the weight needs to be in kg)

2. Find Sarah's dog and log to the console whether it's eating too much or too little. Hint: Some dogs have multiple owners, so you first need to find Sarah in the owners array, and so this one is a bit tricky (on purpose) �

3. Create an array containing all owners of dogs who eat too much ('ownersEatTooMuch') and an array with all owners of dogs who eat too little ('ownersEatTooLittle').

4. Log a string to the console for each array created in 3., like this: "Matilda and Alice and Bob's dogs eat too much!" and "Sarah and John and Michael's dogs eat too little!"

5. Log to the console whether there is any dog eating exactly the amount of food that is recommended (just true or false)

6. Log to the console whether there is any dog eating an okay amount of food (just true or false)

7. Create an array containing the dogs that are eating an okay amount of food (try to reuse the condition used in 6.)

8. Create a shallow copy of the 'dogs' array and sort it by recommended food portion in an ascending order (keep in mind that the portions are inside the array's objects �)

Hints:

Use many different tools to solve these challenges, you can use the summary lecture to choose between them �

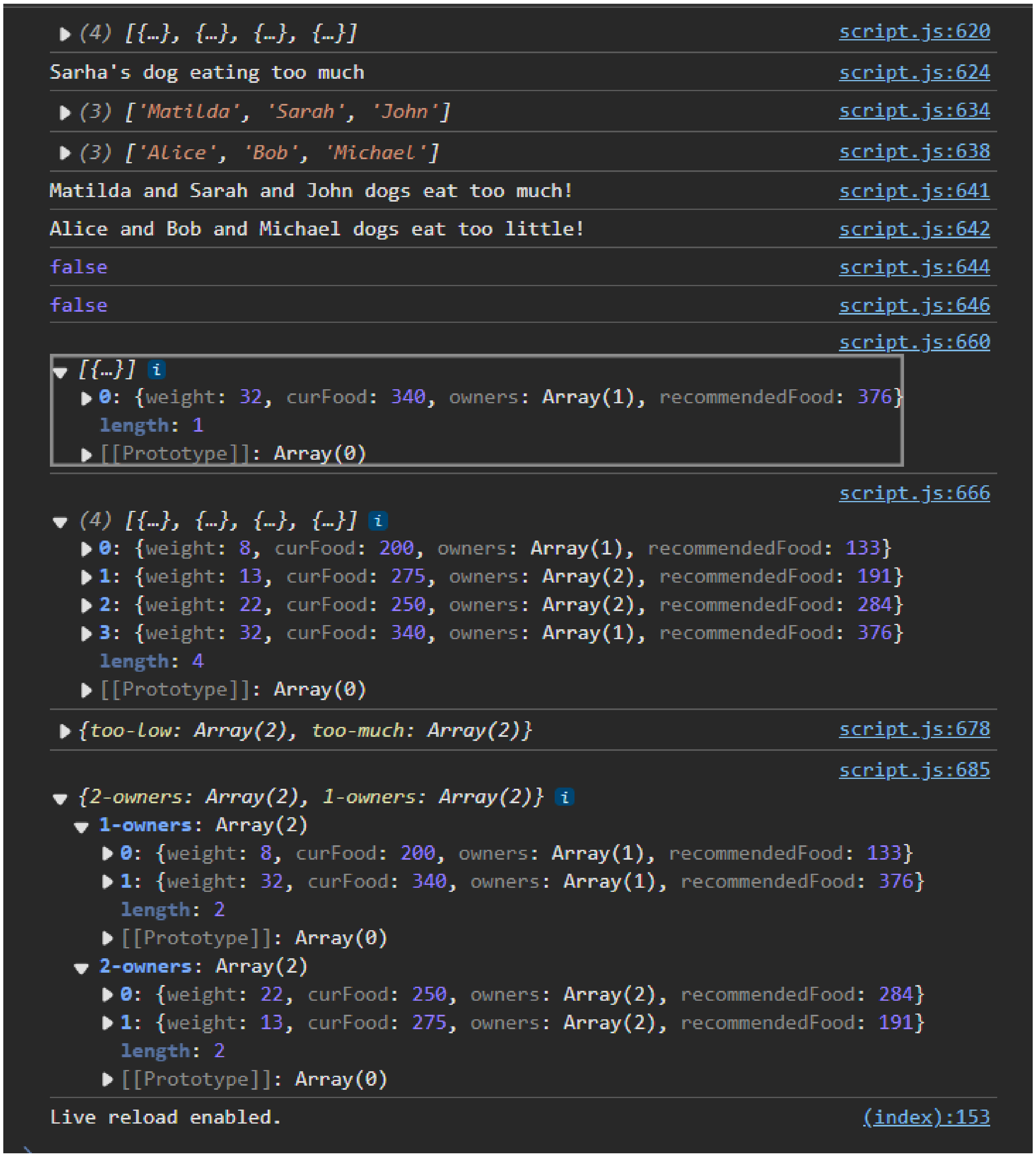
Being within a range 10% above and below the recommended portion means: current > (recommended * 0.90) && current < (recommended * 1.10).

Basically, the current portion should be between 90% and 110% of the recommended portion.

Test data:

const dogs = [{ weight: 22, curFood: 250, owners: ['Alice', 'Bob'] }, { weight: 8, curFood: 200, owners: ['Matilda'] }, { weight: 13, curFood: 275, owners: ['Sarah', 'John'] }, { weight: 32, curFood: 340, owners: ['Michael'] },];

```
//coding challlenge


const dogs = [
  { weight: 22, curFood: 250, owners: ['Alice', 'Bob'] },
  { weight: 8, curFood: 200, owners: ['Matilda'] },
  { weight: 13, curFood: 275, owners: ['Sarah', 'John'] },
  { weight: 32, curFood: 340, owners: ['Michael'] },
];


// dogs.forEach(function (dog) {
//   dog.recFood = dog.weight ** 0.75 * 28;
// });
// console.log(dogs);
dogs.forEach(
  dog => (dog.recommendedFood = Math.floor(dog.weight ** 0.75 * 28))
);
console.log(dogs);


//2
const dogSarah = dogs.find(dog => dog.owners.includes('Sarah'));
console.log(
  `Sarha's dog eating too ${
    dogSarah.curFood > dogSarah.recommendedFood ? 'much' : 'little'

  }`

);



//3

const ownersEatTooMuch = dogs

  .filter(dog => dog.curFood > dog.recommendedFood)

  .flatMap(dog => dog.owners);

console.log(ownersEatTooMuch); //['Matilda', 'Sarah', 'John']

const ownersEatTooLittle = dogs

  .filter(dog => dog.curFood < dog.recommendedFood)

  .flatMap(dog => dog.owners);

console.log(ownersEatTooLittle); //['Alice', 'Bob', 'Michael']



//4

console.log(`${ownersEatTooMuch.join(' and ')} dogs eat too much!`);

console.log(`${ownersEatTooLittle.join(' and ')} dogs eat too little!`);

//5

console.log(dogs.some(dog => dog.curFood === dog.recommendedFood));

//6

console.log(

  dogs.every(
    dog =>

      dog.curFood > dog.recommendedFood * 0.9 &&

      dog.curFood < dog.recommendedFood * 1.1

  )

);

//7

const okayFood = dogs.filter(

  dog =>

    dog.curFood > dog.recommendedFood * 0.9 &&

    dog.curFood < dog.recommendedFood * 1.1

);

console.log(okayFood);

//8

const shallowDogs = dogs

  .slice()

  .toSorted((a, b) => a.recommendedFood - b.recommendedFood);

console.log(shallowDogs);

//group

const dogsGroupByportion = Object.groupBy(dogs, dog => {

  if (dog.curFood === dog.recommendedFood) {

    return 'okay amount';

  } else if (dog.curFood > dog.recommendedFood) {
    return 'too-much';

  } else {

    return 'too-low';

  }

});

console.log(dogsGroupByportion);

//group by owners

const dogsGroupByOwners = Object.groupBy(

  dogs,

  dog => `${dog.owners.length}-owners`

);

console.log(dogsGroupByOwners);
```



```
▶ (4) [{…}, {…}, {…}, {…}]                                          script.js:620
Sarha's dog eating too much                                          script.js:624
▶ (3) ['Matilda', 'Sarah', 'John']                                   script.js:634
▶ (3) ['Alice', 'Bob', 'Michael']                                    script.js:638
Matilda and Sarah and John dogs eat too much!                        script.js:641
Alice and Bob and Michael dogs eat too little!                       script.js:642
false                                                                script.js:644
false                                                                script.js:646
                                                                     script.js:660
▼ [{…}] i
  ▶ 0: {weight: 32, curFood: 340, owners: Array(1), recommendedFood: 376}
    length: 1
  ▶ [[Prototype]]: Array(0)
                                                                     script.js:666
▼ (4) [{…}, {…}, {…}, {…}] i
  ▶ 0: {weight: 8, curFood: 200, owners: Array(1), recommendedFood: 133}
  ▶ 1: {weight: 13, curFood: 275, owners: Array(2), recommendedFood: 191}
  ▶ 2: {weight: 22, curFood: 250, owners: Array(2), recommendedFood: 284}
  ▶ 3: {weight: 32, curFood: 340, owners: Array(1), recommendedFood: 376}
    length: 4
  ▶ [[Prototype]]: Array(0)
▶ {too-low: Array(2), too-much: Array(2)}                            script.js:678
                                                                     script.js:685
▼ {2-owners: Array(2), 1-owners: Array(2)} i
  ▼ 1-owners: Array(2)
    ▶ 0: {weight: 8, curFood: 200, owners: Array(1), recommendedFood: 133}
    ▶ 1: {weight: 32, curFood: 340, owners: Array(1), recommendedFood: 376}
      length: 2
    ▶ [[Prototype]]: Array(0)
  ▼ 2-owners: Array(2)
    ▶ 0: {weight: 22, curFood: 250, owners: Array(2), recommendedFood: 284}
    ▶ 1: {weight: 13, curFood: 275, owners: Array(2), recommendedFood: 191}
      length: 2
    ▶ [[Prototype]]: Array(0)
Live reload enabled.                                                 (index):153
```