

## Javascript( Objects,Set, Map,Strings, Destructing etc...)

### Object

An object in JavaScript is a data structure used to store related data collections. It stores data as key-value pairs, where each key is a unique identifier for the associated value. Objects are dynamic, which means the properties can be added, modified, or deleted at runtime.

You can create objects in several ways:

#### 1) Object Literal Syntax:

the object literal syntax allows you to define and initialize an object with curly braces {}, setting properties as key-value pairs.

```
let obj = {  
  
  name: "Sourav",  
  
  age: 23,  
  
  job: "Developer"  
  
};  
  
console.log(obj);
```

**Output :** { name: 'Sourav', age: 23, job: 'Developer' }

#### 2) Using the new Object() Syntax:

```
let obj = new Object();  
  
obj.name= "Sourav",
```

```
obj.age= 23,  
  
obj.job= "Developer"  
  
console.log(obj);
```

**Output :** { name: 'Sourav', age: 23, job: 'Developer' }

### Accessing and Modifying Object Properties

You can access an object's properties using either **dot notation** or **bracket notation**

```
let obj = { name: "Sourav", age: 23 };
```

**// Using Dot Notation**

```
console.log(obj.name);
```

**// Using Bracket Notation**

```
console.log(obj["name"]);
```

**Output**

**Sourav**

**Sourav**

**Ex2: // Using Dot Notation**

```
console.log(person.name); // John
```

```
person.age = 31; // Modify the age property
```

**// Using Bracket Notation**

```
console.log(person["name"]); // John
```

```
person["age"] = 32; // Modify the age property
```

### Adding Properties to an Object

```
let obj = { model: "Tesla" };  
obj.color = "Red";
```

```
console.log(obj);
```

**Output:** { model: 'Tesla', color: 'Red' }

### Removing Properties from an Object

The delete operator removes properties from an object.

```
let obj = { model: "Tesla", color: "Red" };
```

```
delete obj.color;
```

```
console.log(obj);
```

**Output**

```
{ model: 'Tesla' }
```

### Defining method in JavaScript Object

We can define methods in JavaScript objects.

The **this** keyword in JavaScript refers to the **object that is executing the current function**. It is a way to access properties and methods of the object that owns the method.

**Inside an Object Method:** **this** refers to the object that owns the method.

```
const person = {  
  firstName: "John",  
  lastName: "Doe",  
  fullName() {  
    return this.firstName + " " + this.lastName;  
  },  
};
```

```
console.log(person.fullName()); // Output: "John Doe"
```

Ex2:

```
const car = {  
  brand: "Toyota",  
  model: "Corolla",  
  name: "dhoni",  
  job: "software engineer",  
  salary: 45000,  
  license: true,  
  displayInfo: function () {  
    return `${this.name} has license and he ${this.license ? `can` : `can't`} drive ${this.brand}  
    ${this.model}`;  
  },  
};  
console.log(car.displayInfo());
```

**Output:**

dhoni has license and he can drive Toyota Corolla

**Objects** can store **arrays**, **sets**, and **maps** as its properties, because objects can store any data type as property values, including these complex data structures.

**Example: Storing Arrays, Sets, and Maps in an Object**

```
const obj = {  
  array: [1, 2, 3], // Array  
  set: new Set([4, 5, 6]), // Set  
  map: new Map([['key1', 'value1'], ['key2', 'value2']]) // Map  
};
```

```
console.log(obj);  
  
// Accessing each property  
console.log('Array:', obj.array); // [1, 2, 3]  
console.log('Set:', obj.set);    // Set { 4, 5, 6 }  
console.log('Map:', obj.map);    // Map { 'key1' => 'value1', 'key2' => 'value2' }
```

### Output:

```
{  
  array: [ 1, 2, 3 ],  
  set: Set { 4, 5, 6 },  
  map: Map { 'key1' => 'value1', 'key2' => 'value2' }  
}  
  
Array: [ 1, 2, 3 ]  
Set: Set { 4, 5, 6 }  
Map: Map { 'key1' => 'value1', 'key2' => 'value2' }
```

```
/*  
  
"use strict";  
  
x = 10; // No error, `x` becomes a global variable  
  
console.log(x); // Output: 10  
  
*/  
  
//2. functions  
  
/*  
  
function add(a, b) {
```

```
    return a + b;
}

let result = add(5, 7);

console.log(result); // Output: 12

*/

/*

//3. Arrow Function

//ex1

const sample = age => {

    console.log(age * age);

    console.log(age * age);

}

sample(12);

//single statements) no {} required

const cal = age => age * age;

console.log(cal(9));

//ex3

const ret = year => {

    let age = 2024 - year;

    let YearsLeftretaired = 65 - age;

    return `the age ${age} and age left to retire ${YearsLeftretaired}`;

}

console.log(ret(2000));

*/

/*
```

```
//4. Function calling other function
```

```
function add(a, b) {
```

```
    return a + b;
```

```
}
```

```
function calculateAndPrint() {
```

```
    const result = add(5, 7); // Call `add` with arguments
```

```
    console.log(`The result is: ${result}`);
```

```
}
```

```
calculateAndPrint(); // Outputs: The result is: 12
```

```
*/
```

```
/*
```

```
function sum(a, b, c) {
```

```
    return a + b + c;
```

```
}
```

```
const s = sum(2, 3, 4);
```

```
console.log(s);
```

```
// using function expression or anonymous function
```

```
const sum1 = function (x, y, z) {
```

```
    return x + y + z;
```

```
}
```

```
console.log(sum1(6, 6, 6));
```

```
//using arrow function
```

```
const sum2 = (j, k, l) => {  
    return j + k + l;  
}  
  
console.log(sum2(5, 6, 7));  
  
*/  
  
/*  
  
//challenge  
  
const calcAverage = (x, y, z) => {  
    return ((x + y + z) / 3);  
}  
  
const scoreDolphins = calcAverage(44, 23, 71);  
const scoreKoalas = calcAverage(65, 54, 65);  
console.log(scoreDolphins, scoreKoalas)  
  
function checkWinner(avgDolphins, avgKoalas) {  
    if (avgDolphins >= (2 * avgKoalas)) {  
        console.log(`Dolphins win ${avgDolphins} vs. ${avgKoalas}`);  
    } else if (avgKoalas >= (2 * avgDolphins)) {  
        console.log(`Koalas win ${avgKoalas} vs. ${avgDolphins}`);  
    } else {  
        console.log("No team wins...");  
    }  
}  
  
checkWinner(scoreDolphins, scoreKoalas);
```



```
*/  
  
/*  
  
//Arrays  
  
//JavaScript arrays can hold elements of different types as well. Arrays are zero-indexed, meaning the first  
element is at index 0.  
  
// Creating an Array and Initializing with Values  
  
const a = ["HTML", "CSS", "JS", 2, 3, 4, true, undefined, null];  
  
console.log(a);  
  
console.log(a[0]);  
  
console.log(a[6]);  
  
console.log(a.length);  
  
console.log(a[a.length - 1]); // printing last value in array  
  
a[6] = false; // modifying data  
  
console.log(a);  
  
//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 adds the element to the starting of the array.  
  
a.unshift("dhoni");  
  
console.log(a);  
  
// 3.The pop() method removes an element from the last index of the array.  
  
a.pop();  
  
console.log(a);
```

```
//4. The shift() method removes the element from the first index of the array.

a.shift();

console.log(a);


// 5.indexOf() returns the index of the first occurrence of the element, or -1 if not found.

const l = a.indexOf("CSS");

console.log(l);

const ll = a.indexOf("gg"); //-1(as it is not there in list)

console.log(ll);


//6. includes() returns a boolean (true or false), indicating whether the element exists in the array.


const b = a.includes(3);

console.log(b);//true

const bb = a.includes(7);//false


//Increase and Decrease the Array Length


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


// Increase the array length to 7

arr.length = 7;

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


// Decrease the array length to 2

arr.length = 2;

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

```
//array concatenation

// Creating an Array and Initializing with Values
let array1 = ["HTML", "CSS", "JS", "React"];
let array2 = ["Node.js", "Express.js"];

// Concatenate both arrays
let concatArray = array1.concat(array2);

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

*/

/*

const calcTip = function (x) {
  if (x >= 50 && x <= 300) {
    return x * (15 / 100);
  } else {
    return x * (20 / 100);
  }
}

console.log(calcTip(100));

const bills = [125, 555, 44];

const tips = [calcTip(bills[0]), calcTip(bills[1]), calcTip(bills[2])];

console.log(bills);
```

```
console.log(tips);

const totals = [bills[0] + tips[0], bills[1] + tips[1], bills[2] + tips[2]];

console.log(totals);

*/

/*

//Objects

let obj = {

  name: "Sourav",

  age: 23,

  job: "Developer"

};

//Creation Using new Object() Constructor

console.log(obj);

let obj1 = new Object();

obj1.name = "Sourav",

  obj1.age = 23,

  obj1.job = "Developer"

console.log(obj1);

//accessing data in 2 ways

let obj3 = { name: "Sourav", age: 23 };

// Using Dot Notation

console.log(obj3.name);

// Using Bracket Notation
```

```
console.log(obj3["name"]);

obj3.salary = 7880; //adding new value using Dot
obj3["id"] = 123; //adding new value using bracket
console.log(obj3);

// Object methods
const person = {
  firstName: "John",
  lastName: "Doe",
  job: "software engineer",
  salary: 45000,
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
};

console.log(person.job);
console.log(person["job"]);
console.log(person.fullName());
console.log(person["fullName"]());

const car = {
  brand: "Toyota",
  model: "Corolla",
  name: "dhoni",
  job: "software engineer",
  salary: 45000,
```

```
    license: true,

    displayInfo: function () {

        return `${this.name} has license and he ${this.license ? `can` : `can't`} drive ${this.brand} ${this.model}`;

    },

};
```

```
console.log(car.displayInfo()); // Output: "Toyota Corolla"
```

```
//challenge
```

```
const mark = {

    fullName: "Marks",

    mass: 78,

    height: 1.69,

    calcBMI: function () {

        this.bmi = ((this.mass) / (this.height * this.height));

        return this.bmi;

    }

};
```

```
mark.calcBMI()
```

```
console.log(mark.bmi);
```

```
const john = {

    fullName: "John",

    mass: 92,

    height: 1.95,

    calcBMI: function () {

        this.bmi = ((this.mass) / (this.height * this.height));
```

```
        return this.bmi;
    }
};

john.calcBMI()

console.log(mark.bmi);

// Compare BMIs
if (mark.bmi > john.bmi) {
    console.log(
        `${mark.fullName}'s BMI (${mark.bmi}) is higher than ${john.fullName}'s (${john.bmi})!`
    );
} else if (john.bmi > mark.bmi) {
    console.log(
        `${john.fullName}'s BMI (${john.bmi}) is higher than ${mark.fullName}'s (${mark.bmi})!`
    );
} else {
    console.log(
        `${mark.fullName} and ${john.fullName} have the same BMI (${mark.bmi.toFixed(1)})!`
    );
}

*/

//loops

/*
```

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

```
for (let i = 0; i < a.length; i++) {
```

```
    console.log(a[i]);
```

```
}
```

```
const years = [2000, 2034, 2022, 2027];
```

```
const ages = [];
```

```
for (let j = 0; j < years.length; j++) {
```

```
    ages.push(2060 - years[j]);
```

```
}
```

```
console.log(ages);
```

```
//for of
```

```
for (let str of a) {
```

```
    console.log(str);
```

```
}
```

```
//while
```

```
let i = 1;
```

```
while (i <= 5) {
```

```
    console.log(i);
```

```
    i++;
```

```
}
```

```
//do-while
```

```
let k = 1;
```

```
do {
```

```
    console.log(k);
```

```
    k++;
```

```
} while (k <= 9);
```



```
//challenge

const calcTip = function (bill) {

  return bill >= 50 && bill <= 300 ? bill * 0.15 : bill * 0.2;

}

const bills = [22, 295, 176, 440, 37, 105, 10, 1100, 86, 52];

console.log(bills);

const tips = [];

const totals = [];

for (let i = 0; i < bills.length; i++) {

  const bill = bills[i];

  const tip = calcTip(bills[i]);

  tips.push(tip);

  totals.push(bill + tip);

}

console.log(tips);

console.log(totals);

*/
```

## **Destructing:**

Destructuring in JavaScript was introduced in ECMAScript 6 (ES6), which was released in 2015. This feature allows you to unpack values from arrays or properties from objects into distinct variables.

### **Definition:**

It is basically a way of unpacking values from arrays or objects into separate variables.

Or

Destructuring means to break a complex data structure down into a smaller data structure like a variable.

### **Destructuring Array:**

Destructuring array in JavaScript is a syntax that allows you to break down an array into smaller fragments and assign them to new variables. It's a useful feature for tasks like swapping values and destructuring nested or multiple arrays.

#### **i) //without destructing**

```
const array = [1, 2, 3];  
const a = array[0];  
const b = array[1];  
const c = array[2];  
console.log(a, b, c); // 1 2 3
```

#### **Output**

1 2 3

#### **//with destructing**

```
const array = [1, 2, 3];  
const [x, y, z] = array;  
console.log(x, y, z); // 1 2 3
```

#### **Output**

1 2 3

#### **ii) Skipping Elements:**

You can skip elements in the array by leaving the position empty.

```
const skipElemets = [1, 2, 3, 4];  
const [, , third] = skipElemets;  
console.log(third); // 3
```

#### **iii) Default Values:**

You can assign default values in case the value in the array is `undefined`.

```
const arrayDefault = [1, undefined, 3];  
const [p, q = 2, r = 9, s = 8, t] = arrayDefault;  
console.log(p, q, r, s, t);
```

### Output

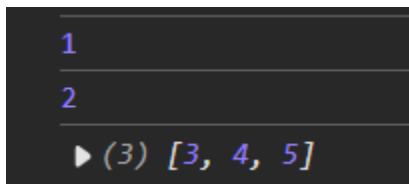
1 2 3 8 undefined

### iv) Rest Syntax:

You can use the rest syntax `...` to collect the remaining elements into an array.

```
//rest syntax  
const restArray = [1, 2, 3, 4, 5];  
const [first, second, ...rest] = restArray;  
console.log(first); // 1  
console.log(second); // 2  
console.log(rest); // [3, 4, 5]
```

### Output



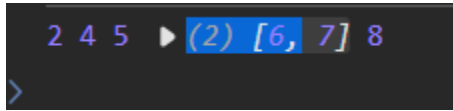
```
1  
2  
▶ (3) [3, 4, 5]
```

### v) Destructuring Nested Arrays:

```
//Destructuring Nested Arrays:  
const nested = [2, 4, 5, [6, 7], 8];  
const [i, j, k, [l, m], n] = nested;
```

```
console.log(i, j, k, [l, m], n);
```

### Output



```
2 4 5 (2) [6, 7] 8
```

### vi) Destructuring with Swap:

#### Before(using third variable to swap)

```
let f = 1;  
let g = 2;  
const temp = f;  
f = g;  
g = temp;  
console.log(f); // 2  
console.log(g); // 1
```

#### After

//Destructuring with Swap:

```
let f = 1;  
let g = 2;  
[f, g] = [g, f];  
console.log(f); // 2  
console.log(g); // 1
```

### Destructuring Objects:

Destructuring objects in JavaScript is a syntax that allows you to break down an object into smaller fragments and assign them to new variables.

### i) Basic Object Destructuring:

This allows you to extract values from an object and assign them to variables with the same name as the properties.

```
const person = {  
  name: 'ram',  
  age: 25,  
};  
  
const { name, age } = person;  
  
console.log(name); // ram  
  
console.log(age); // 25
```

### ii) Renaming Variables:

We can also rename the variables during destructuring to avoid naming conflicts or just for clarity. This is useful when you want to use a more meaningful or convenient name.

```
const person = {  
  name: 'ram',  
  age: 25,  
};  
  
// Renaming 'name' to 'fullName' and 'age' to 'yearsOld'  
  
const { name: fullName, age: yearsOld } = person;
```

```
console.log(fullName); // ram  
  
console.log(yearsOld); // 25
```

In the example above:

The object `person` has keys `name` and `age`.

While destructuring, the `name` key is assigned to the variable `fullName`, and the `age` key is assigned to the variable `yearsOld`.

### iii)Destructuring with Default Values:

If a property doesn't exist, you can provide a default value to avoid `undefined`.

```
const person = { name: 'ram' };  
  
const { name, age = 30 } = person;  
  
console.log(name); // ram  
  
console.log(age); // 30 (default value)
```

### iv)Nested Object Destructuring:

You can destructure nested objects by chaining the destructuring syntax.

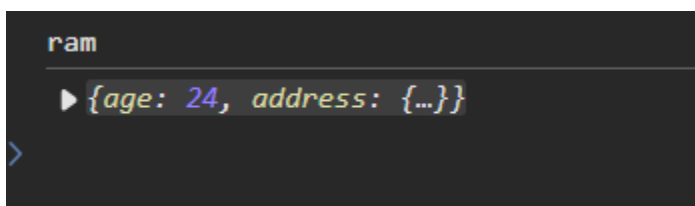
```
const person = {  
  name: 'ram',  
  age: 24,  
  address: { city: 'India', zip: '10001' },  
};  
  
const {name, age, address: { city, zip },} = person;  
  
console.log(name); // ram  
  
console.log(age); // 24  
  
console.log(city); // India  
  
console.log(zip); // 10001
```

ram	<a href="#">script1.js:82</a>
24	<a href="#">script1.js:83</a>
India	<a href="#">script1.js:84</a>
10001	<a href="#">script1.js:85</a>

#### v)Rest Operator in Destructuring:

You can use the rest operator (...) to collect the remaining properties in an object.

```
const person = {  
  name: 'ram',  
  age: 24,  
  address: { city: 'India', zip: '10001' },  
};  
  
const { name, ...rest } = person;  
  
console.log(name); // ram  
  
console.log(rest);
```



ram
▶ {age: 24, address: {...}}
>

#### vi)Destructuring in Function Parameters:

Object destructuring is frequently used in function parameters to extract values directly from the argument object.

```
function greet({ name, age }) {  
  
  console.log(`Hello, my name is ${name} and I'm ${age} years old.`);  
}
```

```
}
```

```
const person = { name: 'ram', age: 25 };
```

```
greet(person); // Hello, my name is ram and I'm 25 years old.
```

### **vii) Destructuring Multiple Objects:**

You can destructure multiple objects into one by combining their properties.

```
const person = { name: 'ram', age: 24, salary: 2000 };
```

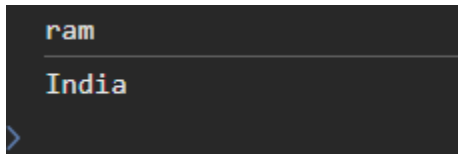
```
const details = { city: 'India', postCode: 1224 };
```

```
// Combine and destructure in a single line, no need to declare location again
```

```
const { name, city } = { ...person, ...details };
```

```
console.log(name); // ram
```

```
console.log(city); // India
```



```
ram
India
```

## **Spread Operator (...)**

Iterables : arrays, strings, maps, sets, not objects.

The JavaScript spread operator (...) expands an iterable (like an array) into more elements.

**This allows us to quickly copy all or parts of an existing array into another array:**

### **i) Copying Arrays**

- Creates a shallow copy of an array.

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

```
const arr2 = [...arr1];
```

```
console.log(arr2); // Output: [1, 2, 3]
```



## ii) Merging Arrays

- Combines multiple arrays into a new one.

```
const arr1 = [1, 2];  
  
const arr2 = [3, 4];  
  
const merged = [...arr1, ...arr2];  
  
console.log(merged); // Output: [1, 2, 3, 4]
```

## iii) Passing Arguments to Functions

- Expands an array into individual arguments.

```
const nums = [10, 20, 30];  
  
const maxNum = Math.max(...nums);  
  
console.log(maxNum); // Output: 30
```

## iv) Copying Objects

- Creates a shallow copy of an object.

```
const obj1 = { a: 1, b: 2 };  
  
const obj2 = { ...obj1 };  
  
console.log(obj2); // Output: { a: 1, b: 2 }
```

## v) Merging Objects

- Combines properties of multiple objects into one.

```
const obj1 = { a: 1, b: 2 };  
  
const obj2 = { b: 3, c: 4 };  
  
const merged = { ...obj1, ...obj2 };  
  
console.log(merged); // Output: { a: 1, b: 3, c: 4 }
```

## vi) Converting Iterables to Arrays

- Transforms strings, sets, or other iterables into arrays.

```
const str = "hello";  
  
const chars = [...str];  
  
console.log(chars); // Output: ['h', 'e', 'l', 'l', 'o']
```

## Rest Pattern and parameters

The spread operator (...) and rest parameters (...) both use the same syntax (...), but they serve different purposes. Here's a detailed comparison to clarify their differences and usage:

### The spread operator (...) is about expanding.

The spread operator is used to **unpack** elements of an iterable (like arrays, objects, or strings) into individual elements or properties.

#### Use Cases:

- Expanding arrays or objects.
- Combining arrays or objects.
- Passing arguments to a function.

#### Array Expansion:

```
const arr1 = [1, 2, 3];  
  
const arr2 = [4, 5, 6];  
  
const combined = [...arr1, ...arr2]; // Combines both arrays  
  
console.log(combined); // [1, 2, 3, 4, 5, 6]
```

#### b. Object Expansion:

```
const obj1 = { a: 1, b: 2 };  
  
const obj2 = { c: 3, d: 4 };  
  
const combinedObj = { ...obj1, ...obj2 }; // Combines objects  
  
console.log(combinedObj); // { a: 1, b: 2, c: 3, d: 4 }
```

#### c. Passing Arguments to Functions:

```
function sum(x, y, z) {  
    return x + y + z;  
}  
  
const numbers = [10, 20, 30];  
  
console.log(sum(...numbers)); // Spread the array into arguments -> Output: 60
```

### **The rest parameter (...) is about collecting.**

The rest parameter is used to collect the remaining arguments into a single array in a function or gather remaining elements/properties in destructuring.

Use Cases:

- Collecting arguments into an array in function parameters.
- Collecting remaining array elements or object properties during destructuring.

**In destructuring, rest syntax cannot be used to collect specific properties or indices; it gathers all remaining properties or elements.**

The rest parameter (...) must be the **last** parameter in a function or destructuring pattern

#### **i) Rest Pattern in Destructuring**

The rest pattern is used to collect remaining elements of an array or properties of an object into a new array or object.

#### **Array Destructuring:**

```
const [first, second, ...rest] = [1, 2, 3, 4, 5];  
  
console.log(first); // 1  
  
console.log(second); // 2  
  
console.log(rest); // [3, 4, 5]
```

#### **Object Destructuring:**

```
const user = { name: "John", age: 30, location: "USA", profession: "Engineer" };  
  
const { name, ...restProps } = user;
```

```
console.log(name);    // John
```

```
console.log(restProps); // { age: 30, location: "USA", profession: "Engineer" }
```

### **Nested Rest in Destructuring:**

```
const data = {  
  id: 1,  
  info: {  
    name: "Alice",  
    age: 25,  
    skills: ["JavaScript", "React"],  
  },  
  meta: "developer",  
};
```

```
const { info: { name, ...restInfo }, ...restData } = data;  
console.log(name);    // Alice  
console.log(restInfo); // { age: 25, skills: ["JavaScript", "React"] }  
console.log(restData); // { id: 1, meta: "developer" }
```

### **ii) Combining Rest Parameters**

Rest parameters and patterns can be combined with other features like default values, nested destructuring, etc.

### **Function with Rest Parameter and Default:**

```
function greet(greeting, ...names) {  
  
  console.log(`${greeting}, ${names.join(", ")}`);  
  
}
```

```
greet("Hello", "Alice", "Bob", "Charlie"); // Output: Hello, Alice, Bob, Charlie
```

### **Flexible Argument Handling:**

```
function multiply(factor, ...numbers) {  
  return numbers.map(num => num * factor);  
}
```

```
console.log(multiply(2, 5, 10, 15)); // Output: [10, 20, 30]
```

## Short circuiting(&& and ||)

### Logical AND (&&)

- Behavior: If the first operand evaluates to **false**, the entire expression short-circuits, and the second operand is not evaluated.
- Usage: **The && operator returns the first falsy value it encounters or the last truthy value if all are truthy.**

```
console.log('Hello' && 42); // 42 (Both are truthy, returns last truthy value)

console.log(null && 42); // null (First operand is falsy, short-circuits)

console.log('text' && 0); // 0 (Second operand is falsy, returns it)

console.log(undefined && false); // undefined (Short-circuits at `undefined`)

console.log(NaN && (console.log("Won't run"), 'world')); // Logs nothing, outputs: NaN
(short-circuits at `NaN`)
```

### Logical OR (||)

- Behavior: If the first operand evaluates to **true**, the entire expression short-circuits, and the second operand is not evaluated.
- Usage: **The || operator returns the first truthy value it encounters or the last falsy value if all are falsy.**

```
console.log('Hello' || 42); // "Hello" (First truthy value is returned)
console.log(null || 42); // 42 (First operand is falsy, returns second operand)
console.log('text' || 0); // 'text' (First operand is truthy, short-circuits)
console.log(undefined || false); // false (Last falsy value is returned)
```

## Nullish coalescing operator(??)

**Nullish:** **null and undefined** (not 0 or ' ')

The **nullish coalescing operator (??)** is a logical operator in JavaScript that returns the right-hand operand when the left-hand operand is **null** or **undefined**.

Unlike the `||` (logical OR) operator, which considers any falsy value (such as `0`, `""`, `false`, `NaN`, `null`, or `undefined`) as a trigger for the right-hand operand, the `??` operator only checks for `null` or `undefined`.

## Syntax

`leftOperand ?? rightOperand`

If `leftOperand` is `null` or `undefined`, the operator returns the `rightOperand`.

If `leftOperand` is any other value (even falsy values like `0`, `false`, `NaN`, or an empty string), the operator returns the `leftOperand`.

Here are the examples of the nullish coalescing operator (`??`) directly in `console.log()` statements:

```
// 1. leftOperand is null
console.log(null ?? 'Default Value'); // Outputs: "Default Value"

// 2. leftOperand is undefined
console.log(undefined ?? 'Default Value'); // Outputs: "Default Value"

// 3. leftOperand is a falsy value (0)
console.log(0 ?? 'Default Value'); // Outputs: 0 (because 0 is not null or undefined)

// 4. leftOperand is a falsy value (empty string)
console.log("" ?? 'Default Value'); // Outputs: "" (because "" is not null or undefined)

// 5. leftOperand is a falsy value (false)
console.log(false ?? 'Default Value'); // Outputs: false (because false is not null or undefined)

// 6. leftOperand is a non-falsy value ("Hello")
console.log('Hello' ?? 'Default Value'); // Outputs: "Hello" (because the left operand is not null or undefined)
```

## Example having `&&`, `||` and `??`

```
const restaurant1 = {
  name: 'Pasta Paradise',
```

```
cuisine: 'Italian',
rating: 4.5,
salary: 0,
};

const restaurant2 = {
  name: 'Burger Haven',
  cuisine: 'American',
  rating: 3.8,
  salary: 5000,
};

// restaurant1.salary is falsy (0), so this will return 0
console.log(restaurant1.name && restaurant1.salary);
// Outputs: 0 (since `salary` is 0, which is falsy)

// If restaurant1.salary was truthy (say 3000), it would return the salary value
restaurant1.salary = 3000;
console.log(restaurant1.name && restaurant1.salary);
// Outputs: 3000 (both `name` and `salary` are truthy, so it returns salary)

// Checking if both properties exist (name and salary) and are truthy
console.log(restaurant2.name && restaurant2.salary);
// Outputs: 5000 (since both `name` and `salary` are truthy)
// restaurant1.salary is falsy (0), so it will fallback to a default value
console.log(restaurant1.price || 'No name available');
// Outputs: "No name available" (price not available and is falsy,)

// restaurant1.salary is falsy (0), so it will fallback to the default salary value
console.log(restaurant1.salary || 2000);
// Outputs: 2000 (because `salary` is 0, which is falsy, so it returns 2000)

// restaurant2.salary is truthy (5000), so it doesn't fallback
console.log(restaurant2.salary || 2000);
// Outputs: 5000 (because salary is truthy)
// Simulate missing salary (null or undefined) for restaurant1
restaurant1.salary = null;
console.log(restaurant1.name ?? 'No name available');
// Outputs: "Pasta Paradise" (since `name` is not null or undefined)
```

```
// restaurant1.salary is null, so it will fallback to the default value
console.log(restaurant1.salary ?? 2000);
// Outputs: 2000 (because `salary` is `null`, so it uses the fallback)

// If restaurant2's salary is not null/undefined, it will return salary itself
console.log(restaurant2.salary ?? 2000);
// Outputs: 5000 (because `salary` is not null or undefined)
```

## Coding challenge

Coding Challenge #1 We're building a football betting app (soccer for my American friends 💎)! Suppose we get data from a web service about a certain game ('game' variable on next page). In this challenge we're gonna work with that data.

Your tasks:

1. Create one player array for each team (variables 'players1' and 'players2')
2. The first player in any player array is the goalkeeper and the others are field players. For Bayern Munich (team 1) create one variable ('gk') with the goalkeeper's name, and one array ('fieldPlayers') with all the remaining 10 field players
3. Create an array 'allPlayers' containing all players of both teams (22 players)
4. During the game, Bayern Munich (team 1) used 3 substitute players. So create a new array ('players1Final') containing all the original team1 players plus 'Thiago', 'Coutinho' and 'Perisic'
5. Based on the game.odds object, create one variable for each odd (called 'team1', 'draw' and 'team2')
6. Write a function ('printGoals') that receives an arbitrary number of player names (not an array) and prints each of them to the console, along with the number of goals that were scored in total (number of player names passed in)
7. The team with the lower odd is more likely to win. Print to the console which team is more likely to win, without using an if/else statement or the ternary operator.

Test data for 6.: First, use players 'Davies', 'Muller', 'Lewandowski' and 'Ki

```
const game = {
```



```
team1: 'Bayern Munich',
team2: 'Borussia Dortmund',
players: [
  [
    'Neuer',
    'Pavard',
    'Martinez',
    'Alaba',
    'Davies',
    'Kimmich',
    'Goretzka',
    'Coman',
    'Muller',
    'Gnarby',
    'Lewandowski',
  ],
  [
    'Burki',
    'Schulz',
    'Hummels',
    'Akanji',
    'Hakimi',
    'Weigl',
    'Witsel',
    'Hazard',
    'Brandt',
    'Sancho',
    'Gotze',
  ],
],
score: '4:0',
scored: ['Lewandowski', 'Gnarby', 'Lewandowski', 'Hummels'],
date: 'Nov 9th, 2037',
odds: {
  team1: 1.33,
  x: 3.25,
  team2: 6.5,
},
};
```

```

//1
const [players1, players2] = game.players;
console.log(players1);
console.log(players2);

//2
const [gk, ...restplayers1] = players1;
console.log(gk, restplayers1);

//3
const allPlayers = [...players1, ...players2];
console.log(allPlayers);

//4
const playerFinal = [...players1, 'Thiago', 'Coutinho', 'Perisic'];
console.log(playerFinal);

//5
const {
  odds: { team1, x: draw, team2 },
} = game;
console.log(team1, draw, team2);

//6
const printGoals = function (...data) {
  console.log(data);
  console.log(`${data.length} goal were scored`);
};
printGoals('Davies', 'Muller', 'Lewandowski', 'Kimmich');
printGoals('Davies', 'Muller');
printGoals(...game.scored);

//7
team1 < team2 && console.log('team1 is more likely to win');
team1 > team2 && console.log('team2 is more likely to win');

```

Output

```
script1.js:241
(11) ['Neuer', 'Pavard', 'Martinez', 'Alaba', 'Davies', 'Kimmich', 'Goretzka', 'Coman', 'Muller', 'Gnarby', 'Lewandowski']
script1.js:242
(11) ['Burki', 'Schulz', 'Hummels', 'Akanji', 'Hakimi', 'Weigl', 'Witsel', 'Hazard', 'Brandt', 'Sancho', 'Gotze']
Neuer script1.js:246
(10) ['Pavard', 'Martinez', 'Alaba', 'Davies', 'Kimmich', 'Goretzka', 'Coman', 'Muller', 'Gnarby', 'Lewandowski']
script1.js:250
(22) ['Neuer', 'Pavard', 'Martinez', 'Alaba', 'Davies', 'Kimmich', 'Goretzka', 'Coman', 'Muller', 'Gnarby', 'Lewandowski', 'Burki', 'Schulz', 'Hummels', 'Akanji', 'Hakimi', 'Weigl', 'Witsel', 'Hazard', 'Brandt', 'Sancho', 'Gotze']
script1.js:254
(14) ['Neuer', 'Pavard', 'Martinez', 'Alaba', 'Davies', 'Kimmich', 'Goretzka', 'Coman', 'Muller', 'Gnarby', 'Lewandowski', 'Thiago', 'Coutinho', 'Perisic']
1.33 3.25 6.5 script1.js:260
script1.js:264
▶ (4) ['Davies', 'Muller', 'Lewandowski', 'Kimmich']
4 goal were scored script1.js:265
▶ (2) ['Davies', 'Muller'] script1.js:264
2 goal were scored script1.js:265
script1.js:264
▶ (4) ['Lewandowski', 'Gnarby', 'Lewandowski', 'Hummels']
4 goal were scored script1.js:265
team1 is more likely to win script1.js:272
>
```

## Enhanced Object Literals

Enhanced Object Literals are a feature introduced in ECMAScript 6 (ES6) that allow for more concise and readable object definitions. They enable shorthand for defining properties and methods within an object. Here's a breakdown of the key features:

### Before (Old JavaScript Syntax)

```
const company = {name: 'hcl'};
```

```
const restaurant = {  
  name: 'ram',  
  location: 'hyd',  
  salary: 120000,  
  company: company,  
  getDetails: function () {  
    return ( this.name + ' lives in ' + this.location + ' having salary of ' + this.salary + ' at '  
this.company );  
  },  
};  
console.log(restaurant.getDetails());
```

### Output

ram lives in hyd having salary of 120000 at hcl

### After (Using Enhanced Object Literals)

```
const company = 'hcl';  
  
const restaurant = {  
  name: 'ram',  
  location: 'hyd',  
  salary: 120000,  
  company, // shorthand for company: company  
  
  getDetails() { // shorthand method definition  
    return `${this.name} lives in ${this.location} having salary of ${this.salary} at  
${this.company}`;  
  },  
};  
  
console.log(restaurant.getDetails());
```

### Shorthand Property Names:

- `company: company` is shortened to just `company` since the property name matches the variable name.

### Shorthand Method Definition:

- `getDetails: function()` is replaced with `getDetails()`. This removes the need to explicitly use the `function` keyword.

## Optional Chaining(?.)

**Optional Chaining (?.)** is a feature in JavaScript that allows you to safely access deeply nested properties or call methods on an object without causing errors if any intermediate value is `null` or `undefined`. It short-circuits the expression and returns `undefined` if any part of the chain is `null` or `undefined`.

This feature is especially useful when dealing with data that may be incomplete or when working with data fetched from APIs where some fields might be missing.

### Syntax

`object?.property`  
`object?.[propertyName]`  
`object?.method()`

Ex;

```
const userData = {
  id: 123,
  name: 'ram',
  profile: {
    picture: 'https://example.com/profile.jpg',
    address: {
      city: 'hyd',
      postalCode: null,
      country: '',
    },
  },
};

// Safely accessing nested properties using Optional Chaining
const profilePicture = userData?.profile?.picture;
const userCity = userData?.profile?.address?.city;
const userZipCode = userData?.profile?.address?.postalCode;
const userPhone = userData?.profile?.contact?.phone;
```

```
const userCountry = userData?.profile?.address?.country;
console.log(profilePicture); // "https://example.com/profile.jpg"
console.log(userCity); // "hyd"
console.log(userZipCode); //undefined
console.log(userPhone); // undefined (no error)
console.log(userCountry); //(empty string)
```

Output

<a href="https://example.com/profile.jpg">https://example.com/profile.jpg</a>	<a href="#">script1.js:338</a>
hyd	<a href="#">script1.js:339</a>
null	<a href="#">script1.js:340</a>
undefined	<a href="#">script1.js:341</a>
	<a href="#">script1.js:342</a>

Ex2:

```
const arr = [1, 2, 3];
console.log(arr?.[1]); // 2
console.log(arr?.[5]); // undefined (no error)
```

Ex3:

```
const user = {
  profile: {
    name: "Bob",
    address: null
  }
};
```

```
console.log(user?.profile?.address?.city); // undefined, without throwing an error
```

## Looping objects

Ex:

```
const userProfile = {
  id: 101,
  name: "John Doe",
  age: 30,
```

```
address: {
  street: "123 Main St",
  city: "New York",
  zip: "10001"
},
skills: ["JavaScript", "React", "Node.js"],
preferences: {
  theme: "dark",
  notifications: {
    email: true,
    sms: false
  }
}
};
```

### **i) Loop Through Top-Level Keys**

```
for (let key in userProfile) {
  console.log(`${key}:`, userProfile[key]);
}
```

### **Output**

```
id: 101
name: John Doe
age: 30
address: { street: '123 Main St', city: 'New York', zip: '10001' }
skills: [ 'JavaScript', 'React', 'Node.js' ]
preferences: { theme: 'dark', notifications: { email: true, sms: false } }
```

### **ii) Object.keys()**

Retrieves an array of the object's keys.

```
const keys = Object.keys(userProfile);
console.log(keys);
```

### **Output**

```
[ 'id', 'name', 'age', 'address', 'skills' ]
```

### Using for..of loop

```
for (const key of Object.keys(userProfile)) {  
  console.log(`Key: ${key}, Value: ${userProfile[key]}`);  
}
```

### Output

Key: id, Value: 101  
Key: name, Value: John Doe  
Key: age, Value: 30  
Key: address, Value: [object Object]  
Key: skills, Value: JavaScript,React,Node.js

### iii)Object.values()

Retrieves an array of the object's values.

```
const values = Object.values(userProfile);  
console.log(values);
```

### Output

```
[  
  101,  
  'John Doe',  
  30,  
  { street: '123 Main St', city: 'New York', zip: '10001' },  
  [ 'JavaScript', 'React', 'Node.js' ]  
]
```

### Using for..of loop

```
for (const value of Object.values(userProfile)) {  
  console.log(`Value: ${value}`);  
}
```

### Output

Value: 101  
Value: John Doe  
Value: 30  
Value: [object Object]  
Value: JavaScript,React,Node.js



#### iv) **Object.entries()**

Retrieves an array of the object's key-value pairs, with each entry as an array **[key, value]**.

```
const entries = Object.entries(userProfile);
console.log(entries);

[
  [ 'id', 101 ],
  [ 'name', 'John Doe' ],
  [ 'age', 30 ],
  [ 'address', { street: '123 Main St', city: 'New York', zip: '10001' } ],
  [ 'skills', [ 'JavaScript', 'React', 'Node.js' ] ]
]
```

#### **Using Object.entries() with for...of**

```
for (const [key, value] of Object.entries(userProfile)) {
  console.log(`Key: ${key}, Value: ${value}`);
}
```

#### Output

```
Key: id, Value: 101
Key: name, Value: John Doe
Key: age, Value: 30
Key: address, Value: [object Object]
Key: skills, Value: JavaScript,React,Node.js
```

### **Coding Challenge #2**

Let's continue with our football betting app! Keep using the 'game' variable from before.

Your tasks:

1. Loop over the game.scored array and print each player name to the console, along with the goal number (Example: "Goal 1: Lewandowski")

2. Use a loop to calculate the average odd and log it to the console (We already studied how to calculate averages, you can go check if you don't remember)

3. Print the 3 odds to the console, but in a nice formatted way, exactly like this:

Odd of victory Bayern Munich: 1.33

Odd of draw: 3.25

Odd of victory Borussia Dortmund: 6.5

Get the team names directly from the game object, don't hardcode them (except for "draw").

Hint: Note how the odds and the game objects have the same property names 💡

4. Bonus: Create an object called 'scorers' which contains the names of the players who scored as properties, and the number of goals as the value. In this game, it will look like this:

```
{ Gnarby: 1,  
  Hummels: 1,  
  Lewandowski: 2  
}
```

```
const game = {  
  team1: 'Bayern Munich',  
  team2: 'Borussia Dortmund',  
  players: [  
    '  
    'Neuer',  
    'Pavard',  
    'Martinez',  
    'Alaba',  
    'Davies',  
    'Kimmich',  
    'Goretzka',  
    'Coman',  
    'Muller',  
    'Gnarby',  
    'Lewandowski',  
  ],  
  [  
    'Burki',  
    'Schulz',  
  ]  
}
```

```

    'Hummels',
    'Akanji',
    'Hakimi',
    'Weigl',
    'Witsel',
    'Hazard',
    'Brandt',
    'Sancho',
    'Gotze',
  ],
],
score: '4:0',
scored: ['Lewandowski', 'Gnarby', 'Lewandowski', 'Hummels'],
date: 'Nov 9th, 2037',
odds: {
  team1: 1.33,
  x: 3.25,
  team2: 6.5,
},
};

//1

for (const [i, player] of game.scored.entries()) {
  console.log(`Goal ${i + 1}: ${player}`);
}

//2

let average = 0;
const length = Object.values(game.odds).length;
for (const odd of Object.values(game.odds)) {
  average += odd;
}
console.group(average);
console.group(length);
console.group(average / length);

//3

```

```

for (const [team, odd] of Object.entries(game.odd)) {
  const teamStr = team === 'x' ? 'draw' : `victory ${game[team]}`;
  // console.log(team, odd);
  console.log(`Odd of ${teamStr} ${odd}`);
}

//4
// BONUS
// So the solution is to loop over the array, and add the array elements as object properties, and then increase
the count as we encounter a new occurrence of a certain element
const scorers = {};
for (const player of game.scored) {
  scorers[player] ? scorers[player]++ : (scorers[player] = 1);
}
console.log(scorers);

```

Goal 1: Lewandowski	<a href="#">script1.js:426</a>
Goal 2: Gnarby	<a href="#">script1.js:426</a>
Goal 3: Lewandowski	<a href="#">script1.js:426</a>
Goal 4: Hummels	<a href="#">script1.js:426</a>
▼ 11.08	<a href="#">script1.js:436</a>
▼ 3	<a href="#">script1.js:437</a>
▼ 3.6933333333333334	<a href="#">script1.js:438</a>
Odd of victory Bayern Munich 1.33	<a href="#">script1.js:444</a>
Odd of draw 3.25	<a href="#">script1.js:444</a>
Odd of victory Borussia Dortmund 6.5	<a href="#">script1.js:444</a>
	<a href="#">script1.js:454</a>
▶ {Lewandowski: 2, Gnarby: 1, Hummels: 1}	

## Set

The **Set** data structure in JavaScript is a built-in object that stores unique values of any type, whether primitive values or object references. It is part of ES6 and is commonly used when uniqueness is needed.

**Unique Values:** A **Set** does not allow duplicate values.

**Order:** Iteration of a **Set** follows the insertion order.

**Here are some common methods available in a Set:**

- **add(value)**: Adds a new value to the **Set**.
- **delete(value)**: Removes a specific value from the **Set**.
- **has(value)**: Returns **true** if the value exists in the **Set**, otherwise **false**.
- **clear()**: Removes all values from the **Set**.
- **size**: Returns the number of values in the **Set**.
- **keys()**: Returns an iterator for the keys in the **Set** (same as values in **Set**).
- **values()**: Returns an iterator for the values in the **Set**.
- **entries()**: Returns an iterator with **[value, value]** pairs.
- **forEach(callback)**: Executes a callback function for each value in the **Set**.

```
// Create a Set
const mySet = new Set();

// Add values
mySet.add(1);
mySet.add(2);
mySet.add(3);
mySet.add(3); // Duplicate value, will not be added

console.log('Set after adding values:', mySet); // Set(3) { 1, 2, 3 }

// Check size
console.log('Size of the Set:', mySet.size); // 3

// Check if a value exists
console.log('Does Set contain 2?', mySet.has(2)); // true

// Delete a value
mySet.delete(2);
console.log('Set after deleting 2:', mySet); // Set(2) { 1, 3 }
```

```
// Clear the Set
mySet.clear();
console.log('Set after clearing:', mySet); // Set(0) {}
```

### i) Converting an Array to a Set

```
const array = [1, 2, 2, 3, 4, 4, 5, 'ram', 'king', 78.9, true, null];
const set = new Set(array);

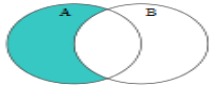
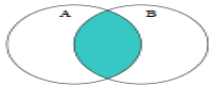
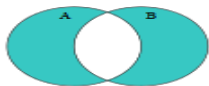
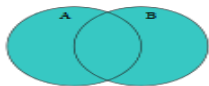

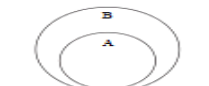
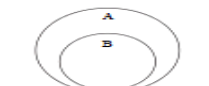
console.log('Array:', array); // [1, 2, 2, 3, 4, 4, 5, 'ram', 'king', 78.9, true, null]
console.log('Set:', set); // { 1, 2, 3, 4, 5, 'ram', 'king', 78.9, true, null }
```

### ii) Converting a Set to an Array

We can convert a [Set](#) back to an array using the [Array.from\(\)](#) method or the spread operator (...).

```
const set = new Set([1, 2, 3, 4, 5]);
const array1 = Array.from(set); // Using Array.from()
const array2 = [...set];       // Using the spread operator

console.log("Set:", set);      // Set(5) { 1, 2, 3, 4, 5 }
console.log("Array (Array.from):", array1); // [1, 2, 3, 4, 5]
console.log("Array (spread):", array2);    // [1, 2, 3, 4, 5]
```

Method	Return type	Mathematical equivalent	Venn diagram
<code>A.difference(B)</code>	Set	$A \setminus B$	
<code>A.intersection(B)</code>	Set	$A \cap B$	
<code>A.symmetricDifference(B)</code>	Set	$(A \setminus B) \cup (B \setminus A)$	
<code>A.union(B)</code>	Set	$A \cup B$	
<code>A.isDisjointFrom(B)</code>	Boolean	$A \cap B = \emptyset$	
<code>A.isSubsetOf(B)</code>	Boolean	$A \subseteq B$	
<code>A.isSupersetOf(B)</code>	Boolean	$A \supseteq B$	

### i) intersection

The intersection of two sets is a set containing all elements that exist in both sets.

```
const setA = new Set([1, 2, 3, 4]);
const setB = new Set([3, 4, 5, 6]);

const intersectionSet = setA.intersection(setB);
console.log('intersection', intersectionSet); // {3,4}
```

### ii) union

A union set is the combination of two elements. In mathematical terms, the union of two sets is shown by  $A \cup B$ . It means all the elements in set A and set B should occur in a single array. In JavaScript, a union set means adding one set element to the other set.

```
const setA = new Set([1, 2, 3, 4]);
const setB = new Set([3, 4, 5, 6]);
//using union
const unionSet = setA.union(setB);
```

```
console.log('union', unionSet); //{1,2,3,4,5,6}
//using spread
const unionUsingSpred = new Set([...setA, ...setB]);
console.log(unionUsingSpred); //{1,2,3,4,5,6}
```

### iii) difference

When comparing `setA` with `setB`, calling `setA.difference(setB)` returns the elements that are present in `setA` but not in `setB` (It will return the elements that are in `setA` but not in `setB`, excluding the common elements between them.).

Conversely, when comparing `setB` with `setA`, calling `setB.difference(setA)` returns the elements that are present in `setB` but not in `setA` (It will return the elements that are in `setB` but not in `setA`, excluding the common elements between them.).

```
const setA = new Set([1, 2, 2, 3, 4]);
const setB = new Set([3, 4, 5, 6]);

const differenceSet = setA.difference(setB);
console.log('differenceSet', differenceSet); //{1,2}
```

```
const setA = new Set([1, 2, 2, 3, 4]);
const setB = new Set([3, 4, 5, 6]);

const differenceSet = setB.difference(setA);
console.log('differenceSet', differenceSet); //{5,6}
```

### iv) symmetricDifference

The `symmetricDifference()` method of [Set](#) instances takes a set and returns a new set containing elements which are in either this set or the given set, but not in both.

It will return elements from both sets (apart from common elements) even though we perform `setA.symmetricDifference(setB)` or `setB.symmetricDifference(setA)`.

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



```
const setB = new Set([3, 4, 5, 6]);

const differenceSet = setA.symmetricDifference(setB);

console.log('differenceSet', differenceSet); //{1,2,5,6}
```

## Maps

A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.

### How to Create a Map

You can create a JavaScript Map by:

- Passing an Array to `new Map()`
- Create a Map and use `Map.set()`

### Methods / Properties of JavaScript Map

- `set(key, val)`: Adds or updates an element with a specified key and value.
- `get(key)`: Returns the value associated with the specified key.
- `has(key)`: Returns a boolean indicating whether an element with the specified key exists.
- `delete(key)`: Removes the element with the specified key.
- `clear()`: Removes all elements from the Map.
- `size`: Returns the number of key-value pairs in the Map.

### When to Use Objects vs. Maps:

In JavaScript, both **objects** and **Maps** are used to store collections of key-value pairs. However, there are important differences between them

- **Use Objects:**
  - When you need a simple key-value pair collection.
  - When the keys are strings or symbols.
  - When you don't need to preserve the order of keys.
  - No built-in methods like `.set()`, `.get()`, `.delete()`
- **Use Maps:**
  - When keys can be any type (e.g., objects, functions).
  - When you need to preserve the insertion order of keys.
  - When you need better performance for frequently modifying collections.
  - When you need built-in methods for common operations (e.g., `.set()`, `.get()`, `.has()`).

## Ex1

```
const myMap = new Map();

// Adding key-value pairs
myMap.set('name', 'Alice');
myMap.set('age', 30);

// Accessing values
console.log(myMap.get('name')); // "Alice"
console.log(myMap.get('age')); // 30

// Checking if a key exists
console.log(myMap.has('name')); // true
console.log(myMap.has('address')); // false

// Getting the size of the Map
console.log(myMap.size); // 2

// Deleting a key-value pair
myMap.delete('age');
console.log(myMap.get('age')); // undefined

// Clearing all pairs
myMap.clear();
console.log(myMap.size); // 0
```

## Converting array to map and Map to Arrays

```
const arr = [
  ['name', 'Alice'],
  ['age', 30],
  ['city', 'New York'],
];
let map = new Map(arr);

console.log(map);
// Output: Map { 'name' => 'Alice', 'age' => 30, 'city' => 'New York' }

//map to array

const myMap = new Map();
myMap.set('name', 'Alice');
myMap.set('age', 30);

let entriesArray = Array.from(myMap.entries());
console.log(entriesArray);
// Output: [['name', 'Alice'], ['age', 30]]
```

## Object to map and map to object

Object to map using `Object.entries(obj)`

To convert a `Map` to an object in JavaScript, you can use the `Object.fromEntries()` method, which takes an iterable (such as a `Map`) and converts it to an object.

```
const question = new Map([
  ['name', 'ram'],
  [1, 'A'],
  [2, 'B'],
  [true, 'correct'],
  [false, 'wrong'],
  [null, 'nothing'],
  [undefined, 'nothing'],
]);
```

```

});

//obj to map

const obj = { name: 'ram', age: 25, city: 'wgl' };
const map = new Map(Object.entries(obj));
console.log(map); //Map(3) {'name' => 'ram', 'age' => 25, 'city' => 'wgl'}

//map to obj

const mapToObj = Object.fromEntries(question);
console.log(mapToObj); //{name: 'ram', '1': 'A', '2': 'B', true: 'correct', false: 'wrong', null:
'nothing', undefined: 'nothing'}

```

### Ex3:

```

const contacts = new Map();
contacts.set('Jessie', { phone: '213-555-1234', address: '123 N 1st Ave' });
console.log(contacts);
console.log(contacts.has('Jessie')); // true
console.log(contacts.get('Hilary')); // undefined
contacts.set('Hilary', { phone: '617-555-4321', address: '321 S 2nd St' });
console.log(contacts); //Map(2) {'Jessie' => {...}, 'Hilary' => {...}}
contacts.get('Jessie'); // {phone: "213-555-1234", address: "123 N 1st Ave"}
contacts.delete('Raymond'); // false
contacts.delete('Jessie'); // true
console.log(contacts.size); // 1

```

### Map iteration

```

const question = new Map([
  ['name', 'ram'],
  [1, 'A'],
  [2, 'B'],
  [true, 'correct'],
  [false, 'wrong'],
  [null, 'nothing'],

```

```

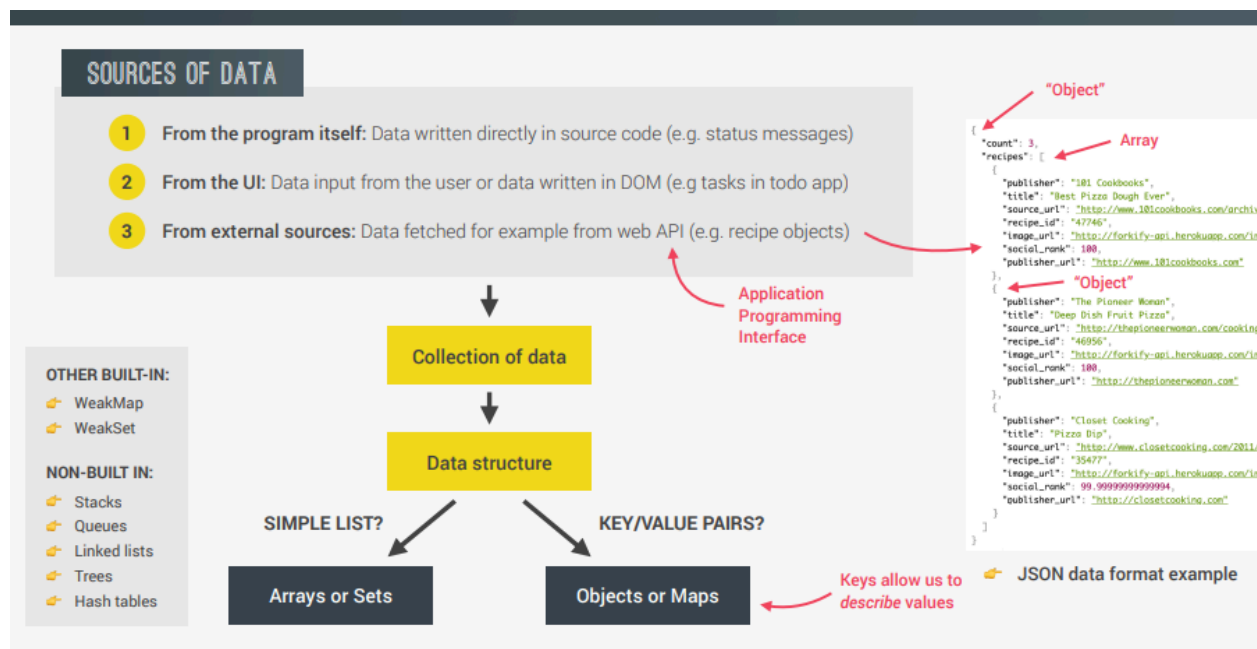
[undefined, 'nothing'],
]);
//using foreach
question.forEach((value, key) => {
  console.log(key, value);
});

//Using for...of loop:
for (let [key, value] of question) {
  console.log(`${key}: ${value}`);
}

```

## Data source

Will get data generally from below ways(3 ways) and stores in data structure.



ARRAYS VS. SETS AND OBJECTS VS. MAPS			
ARRAYS	VS.	SETS	
<pre>tasks = ['Code', 'Eat', 'Code']; // ["Code", "Eat", "Code"]</pre>		<pre>tasks = new Set(['Code', 'Eat', 'Code']); // {"Code", "Eat"}</pre>	
<ul style="list-style-type: none"> <li>Use when you need <b>ordered</b> list of values (might contain duplicates)</li> <li>Use when you need to <b>manipulate</b> data</li> </ul>		<ul style="list-style-type: none"> <li>Use when you need to work with <b>unique</b> values</li> <li>Use when <b>high-performance</b> is <i>really</i> important</li> <li>Use to <b>remove duplicates</b> from arrays</li> </ul>	
		VS.	
		OBJECTS	MAPS
		<pre>task = {   task: 'Code',   date: 'today',   repeat: true };</pre>	<pre>task = new Map([   ['task', 'Code'],   ['date', 'today'],   [false, 'Start coding!'] ]);</pre>
		<ul style="list-style-type: none"> <li>More "traditional" key/value store ("abused" objects)</li> <li>Easier to write and access values with <code>.</code> and <code>[]</code></li> <li>Use when you need to include <b>functions</b> (methods)</li> <li>Use when working with JSON (can convert to map)</li> </ul>	<ul style="list-style-type: none"> <li>Better performance</li> <li>Keys can have <b>any</b> data type</li> <li>Easy to iterate</li> <li>Easy to compute size</li> <li>Use when you simply need to map key to values</li> <li>Use when you need keys that are <b>not</b> strings</li> </ul>

### Coding challenge 3

Let's continue with our football betting app!

This time, we have a map called 'gameEvents' (see below) with a log of the events that happened during the game. The values are the events themselves, and the keys are the minutes in which each event happened (a football game has 90 minutes plus some extra time).

Your tasks:

1. Create an array 'events' of the different game events that happened (no duplicates)
2. After the game has finished, it was found that the yellow card from minute 64 was unfair. So remove this event from the game events log.
3. Compute and log the following string to the console: "An event happened, on average, every 9 minutes" (keep in mind that a game has 90 minutes)
4. Loop over 'gameEvents' and log each element to the console, marking whether it's in the first half or second half (after 45 min) of the game, like this: [FIRST HALF] 17: ⚽ GOA

```
const gameEvents = new Map([
  [17, '⚽ GOAL'],
  [36, '🔄 Substitution'],
```

```

[47, '⚽ GOAL'],
[61, '🔄 Substitution'],
[64, '🟡 Yellow card'],
[69, '🔴 Red card'],
[70, '🔄 Substitution'],
[72, '🔄 Substitution'],
[76, '⚽ GOAL'],
[80, '⚽ GOAL'],
[92, '🟡 Yellow card'],
]);

//1 map to set
const events = [...new Set(gameEvents.values())]; //Array.from(gameEvents.entries());
console.log(events);

//2
gameEvents.delete(64);

//3
console.log(`An event happened, on
average, every ${90 / gameEvents.size} minutes`);

//4

for (const [min, event] of gameEvents) {
  const half = min <= 45;
  console.log(`${half} HALF ${min}: ${event}`);
}

```

## Strings

Declaring Strings

**Strings can be declared using:**

1. Double quotes: "Hello"
2. Single quotes: 'World'
3. Template literals: `Hello, \${name}!` (introduced in ES6)

## 1. Length Property

Returns the number of characters in a string.

```
const str = "Hello, World!";  
console.log(str.length); // Output: 13
```

- **Negative Case:**

If the string is empty, `.length` will return 0.

```
const emptyStr = "";  
console.log(emptyStr.length); // Output: 0
```

## 2. Accessing Characters

Access a character using its index.

```
const str = "Hello";  
console.log(str[0]); // Output: H  
console.log(str.charAt(1)); // Output: e
```

**Negative Case:**

If the index is out of range, it will return `undefined` or an empty string.

```
console.log(str[10]); // Output: undefined  
console.log(str.charAt(10)); // Output: ""
```

## 3. Case Conversion

- `toUpperCase()` – Converts string to uppercase.
- `toLowerCase()` – Converts string to lowercase.

```
const str = "Hello";  
console.log(str.toUpperCase()); // Output: HELLO  
console.log(str.toLowerCase()); // Output: hello
```

- **Negative Case:**

These methods work only on strings and will throw an error if called on `null` or `undefined`.

```
const invalidStr = null;  
try {  
  console.log(invalidStr.toUpperCase());  
}
```



```
} catch (error) {  
  console.log(error.message); // Output: Cannot read properties of null  
}
```

## 4. Searching in Strings

- `indexOf()` – Returns the index of the first occurrence of a specified value.
- `lastIndexOf()` – Returns the index of the last occurrence.
- `includes()` – Checks if a string contains a specified value.
- `startsWith()` – Checks if a string starts with a specified value.
- `endsWith()` – Checks if a string ends with a specified value.

```
const str = "Hello, World!";  
console.log(str.indexOf("World")); // Output: 7  
console.log(str.lastIndexOf("o")); // Output: 8  
console.log(str.includes("Hello")); // Output: true  
console.log(str.startsWith("Hello")); // Output: true  
console.log(str.endsWith("!")); // Output: true
```

- **Negative Case:**

If the value is not found, methods like `indexOf()` and `lastIndexOf()` return `-1`, while `includes()`, `startsWith()`, and `endsWith()` return `false`.

```
console.log(str.indexOf("Planet")); // Output: -1  
console.log(str.includes("Planet")); // Output: false
```

## 5. Extracting Substrings

- `slice(start, end)` – Extracts a section of the string.

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

```
const airLine = 'AIR India Airline';  
  
console.log(airLine.slice(-1, -4));
```

Output



- `substring(start, end)` – Similar to `slice`, but does not accept negative indices.
- `substr(start, length)` – Extracts a substring based on start index and length.

```
const str = "JavaScript";
console.log(str.slice(0, 4)); // Output: Java
console.log(str.substring(0, 4)); // Output: Java
console.log(str.substr(0, 4)); // Output: Java
```

- **Negative Case:**  
Using out-of-bound indices or invalid arguments may yield an empty string.

```
console.log(str.slice(20, 25)); // Output: ""
console.log(str.substring(-5, 2)); // Output: "Ja"
```

## 6. Modifying Strings

- `replace()` – Replaces the first match of a substring or pattern.
- `replaceAll()` – Replaces all matches of a substring or pattern.
- `trim()` – Removes whitespace from both ends of a string.

```
const str = " Hello, World! ";
console.log(str.replace("World", "JavaScript")); // Output: Hello, JavaScript!
console.log(str.replaceAll(" ", "")); // Output: Hello,World!
console.log(str.trim()); // Output: Hello, World!
```

- **Negative Case:**  
If the substring or pattern is not found, the original string is returned unchanged.

```
console.log(str.replace("Universe", "Planet")); // Output: Hello, World!
```

## 7. Splitting and Joining Strings

- `split()` – Splits a string into an array.
- `concat()` – Joins two or more strings.
- `repeat()` – Repeats a string a specified number of times.

```
const str = "Apple, Banana, Cherry";
console.log(str.split(", ")); // Output: ["Apple", "Banana", "Cherry"]
console.log("Hello".concat(", ", "World!")); // Output: Hello, World!
console.log("Hi".repeat(3)); // Output: HiHiHi
```

```
////////
let str = "apple,banana,orange";
```

```
let result = str.split(","); // Splits the string by commas
console.log(result); // Output: ["apple", "banana", "orange"]
/////
let sentence = "Hello World!";
let words = sentence.split(" "); // Splits the string by spaces
console.log(words); // Output: ["Hello", "World!"]
```

////To join elements of an array into a string, use the `.join()` method.

```
let words = ["apple", "banana", "orange"];
let result = words.join(", "); // Joins the array elements with commas
console.log(result); // Output: "apple, banana, orange"
```

```
//////////
let chars = ["H", "e", "l", "l", "o"];
let word = chars.join(""); // Joins with no space between
console.log(word); // Output: "Hello"
```

- **Negative Case:**

Using invalid arguments for `split()` or `repeat()` may produce errors or unexpected results.

```
console.log(str.split(";")); // Output: ["Apple, Banana, Cherry"]
try {
  console.log("Hi".repeat(-1));
} catch (error) {
  console.log(error.message); // Output: Invalid count value
}
```

## 8. String Comparisons

- `localeCompare()` – Compares two strings.

```
const str1 = "a";
const str2 = "b";
console.log(str1.localeCompare(str2)); // Output: -1
```

- **Negative Case:**

Passing `undefined` or invalid arguments can cause unexpected behavior.

```
console.log(str1.localeCompare(undefined)); // Output: NaN
```

**Ex:**

```
// Strings Example

const airLine = 'AIR India Airline';

// Length of the string
console.log('Length:', airLine.length); // 17

// Accessing Characters
console.log('First Character:', airLine[0]); // A
console.log('Character at Index 5:', airLine.charAt(5)); // n

// Case Conversion
console.log('Lowercase:', airLine.toLowerCase()); // air india airline
console.log('Uppercase:', airLine.toUpperCase()); // AIR INDIA AIRLINE

// Searching in Strings
console.log("Index of 'IN':", airLine.indexOf('IN')); // -1 (not found)
console.log("Index of 'In':", airLine.indexOf('In')); // 4 (found at position 4)
console.log("Last Index of 'i':", airLine.lastIndexOf('i')); // 14
console.log("Includes 'India':", airLine.includes('India')); // true
console.log("Starts with 'AI':", airLine.startsWith('AI')); // true
console.log("Ends with 'ne':", airLine.endsWith('ne')); // true

// Extracting Substrings
console.log('Slice (0, 4):', airLine.slice(0, 4)); // AIR
console.log('Slice (2, 4):', airLine.slice(2, 4)); // R
console.log('Slice (2, -4):', airLine.slice(2, -4)); // R India Air
console.log('Slice (2):', airLine.slice(2)); // R India Airline
console.log(
  'Lowercase First Character:',
  airLine[0].toLowerCase() + airLine.slice(1)
); // aIR India Airline
console.log('Substring (1, 3):', airLine.substring(1, 3)); // IR
console.log('Substr (1, 6):', airLine.substr(1, 6)); // IR Ind
```

```
// Modifying Strings
console.log("Replace 'AIR':", airLine.replace('AIR', 'Air')); // Air India Airline
console.log("Replace All 'i' with 'I':", airLine.replaceAll('i', 'I')); // AIR India AIRline
const str = ' Hello, World! ';
console.log('Trimmed String:', str.trim()); // Hello, World!

// Splitting and Joining Strings
console.log("Split by ':'", airLine.split(':')); // ['AIR India Airline']
console.log("Split by ' ',", str.split(' ')); // [' Hello', 'World! ']
console.log('Concatenate Strings:', 'aeroplane'.concat(' ', airLine)); // aeroplane AIR India Airline
console.log("Concatenate with ',good':", airLine.concat(' ', 'good')); // AIR India Airline ,good
console.log('Repeat String 3 Times:', airLine.repeat(3)); // AIR India AirlineAIR India AirlineAIR India Airline
```

## Padding

padding a string refers to the process of adding characters to the beginning or the end of a string to achieve a desired length. This is commonly used when you need strings to be of a specific length for formatting purposes.

You can use the `String.prototype.padStart()` and `String.prototype.padEnd()` methods for padding a string.

### 1. `padStart(targetLength, padString)`

This method adds padding to the beginning of the string until it reaches the `targetLength`.

- `targetLength`: The length of the resulting string after padding.
- `padString`: The string used for padding. If it's too short, it is repeated as necessary.

#### Example:

```
let str = '42';

let paddedStr = str.padStart(5, '0');

console.log(paddedStr); // Output: '00042'
```

### 2. `padEnd(targetLength, padString)`

This method adds padding to the end of the string until it reaches the `targetLength`.

- **targetLength**: The length of the resulting string after padding.
- **padString**: The string used for padding.

#### Example:

```
let str = '42';

let paddedStr = str.padEnd(5, '0');

console.log(paddedStr); // Output: '42000'
```

```
///padding

let string = 'good morning';

console.log(string.padStart(20, '*')); //*****good morning

console.log(string.padEnd(20, '*')); //good morning*****

//ex2

const creditCard = function (number) {

  const str = number + ' ';

  const last = str.slice(-4);

  return last.padStart(str.length, '*');

};

console.log(creditCard(112333363)); //*****363

console.log(creditCard(11233336366666)); //*****666

console.log(creditCard(11200)); /**200
```

#### Coding challenge

Write a program that receives a list of variable names written in `underscore_case` and convert them to `camelCase`. The input will come from a textarea inserted into the DOM

(see code below to insert the elements), and conversion will happen when the button is pressed.

**Test data (pasted to textarea, including spaces):**

Underscore\_case

first\_name

Some\_Variable

calculate AGE

delayed\_departure

**Should produce this output (5 separate console.log outputs):**

underscoreCase 

firstName  



someVariable   

calculateAge    

delayedDeparture     

Hints: § Remember which character defines a new line in the textarea 

§ The solution only needs to work for a variable made out of 2 words, like a\_b

§ Start without worrying about the . Tackle that only after you have the variable name conversion working 

§ This challenge is difficult on purpose, so start watching the solution in case you're stuck. Then pause and continue

```
const st = 'underscore_case';

const [a, b] = st.split('_');

console.log(a, b);
```





```

const flights =

'_Delayed_Departure;fao93766109;txl2133758440;11:25+_Arrival;bru0943384722;fao93766109;11:45+_Dela
yed_Arrival;hel7439299980;fao93766109;12:05+_Departure;fao93766109;lis2323639855;12:30';

// 🚫 Delayed Departure from FAO to TXL (11h25)
//      Arrival from BRU to FAO (11h45)
// 🚫 Delayed Arrival from HEL to FAO (12h05)
//      Departure from FAO to LIS (12h30)

const getCode = str => str.slice(0, 3).toUpperCase();

for (const flight of flights.split('+')) {
  const [type, from, to, time] = flight.split(';');
  const output = `${type.startsWith('_Delayed') ? '🚫' : ''}${type.replaceAll(
    '\_',
    ''
  )} ${getCode(from)} ${getCode(to)} (${time.replace(':', 'h')})`.padStart(36);
  console.log(output);
}

```