ES6 Features

Amr A Khllaf

June 25, 2025

Contents

1 ES6 Features

This document outlines some of the key features introduced in ECMAScript 6 (ES6), also known as ECMAScript 2015. These features enhance the JavaScript language, making it more powerful and easier to work with.

2 Use Strict Mode (ES5)

Strict mode is a way to opt in to a restricted variant of JavaScript, which helps catch common coding errors and "unsafe" actions such as defining global variables unintentionally. It can be applied to entire scripts or individual functions. To enable strict mode, you can add the directive `"use strict";` at the beginning of your script or function.

- Use strict mode to enforce stricter parsing and error handling in your JavaScript code.

2.1 Some Problem With JS which Solve by Strict Mode

1. Declare Variable Without **var** Keyword

```
x = "Ahmed";
console.log(x); // Ahmed
// SyntaxError: Use of an uninitialized variable in strict mode
// ReferenceError: x is not defined in strict mode
```

2. Delete Any Declaration in Run Time

```
x = "Ahmed";
delete x;
console.log(x); // Ahmed
// SyntaxError: Delete of an unqualified identifier in strict mode
```

3. Declare a Function with the Same 2 Parameters Name

```
function myFunction(a, a) {
  console.log(a);
  // SyntaxError: Duplicate parameter name not allowed in this context
  ↪  With Strict Mode
  // This Cause an Ambiguity in the function's parameters
  // In non-strict mode, the second 'a' would be ignored, but in strict
  ↪  mode, it throws an error
}
```

4. Declare a Variable With the name of some Reserved Keywords

```
var let = "Ahmed";
console.log(let); // Ahmed
// SyntaxError: Unexpected token let With Strict Mode
```

2.2 Solution by Strict Mode

To enable strict mode, you can add the directive `"use strict";` at the beginning of your script or function. This will enforce stricter parsing and error handling in your JavaScript code.

```
"use strict";
function myFunction() {
  // Code here is in strict mode
}
```

- I can use the `"use strict";` directive at the beginning of a script or function to enable strict mode.

```
function myFunction() {
  "use strict";
  // Code here is in strict mode
}
```

---

3 var, let, and const

In ES6, three keywords are used to declare variables: `var`, `let`, and `const`. Each has its own scope and behavior.

3.1 var

- `var` is function-scoped or globally scoped, meaning it is accessible throughout the function or globally if declared outside a function.
- Variables declared with `var` can be re-declared and updated.

```
var x = 10;
console.log(x); // 10
var x = "Hello";
console.log(x); // Hello
// Re-declaring a variable with var is allowed
```

3.1.1 Hoisting with var keyword

- Variables declared with `var` are hoisted to the top of their scope, meaning they can be accessed before their declaration, but they will be `undefined` until the line where they are defined is executed.

```
console.log(a); // undefined
// Accessing a variable before its declaration with var will return
↪  undefined
var a = 5;
console.log(a); // 5
```

3.1.2 Scopes in JavaScript

JavaScript has two main types of scopes: global scope and local scope (which includes function scope and block scope).

- **Global Scope**: Variables declared outside of any function or block are in the global scope and can be accessed from anywhere in the code.
- **Function Scope**: Variables declared within a function are only accessible within that function.
- **Block Scope**: Variables declared within a block (e.g., inside an `if` statement or a loop) are only accessible within that block. However, `var` does not create block scope; it is function-scoped.

```
function myFunction() {
  var localVar = "I am local";
  console.log(localVar); // I am local
}
console.log(localVar); // ReferenceError: localVar is not defined
---
if (true) {
  var x = "Amr";
  console.log(x); // Amr
  // x is accessible here because var is function-scoped
}
console.log(x); // Amr
// x is still accessible here because var does not create block scope
```

3.2 let

- `let` is block-scoped, meaning it is only accessible within the block it is defined in (e.g., inside a loop or an if statement).
- Variables declared with `let` can be updated but not re-declared in the same scope (though they can be re-declared in different scopes).

```
let y = 20;
console.log(y); // 20
```

3.3 const

- `const` is block-scoped, similar to `let`, but is used to declare variables that cannot be reassigned.
- Variables declared with `const` must be initialized at the time of declaration.

```
const z = 30;
console.log(z); // 30
```

3.4 Hoisting with let and const

Variables declared with `let` and `const` are also hoisted, but they are not initialized until their declaration is reached in the code. This means that accessing them before their declaration will result in a `ReferenceError`.

```
console.log(y); // ReferenceError: Cannot access 'y' before
↪  "initialization" which is called Temporal Dead Zone (TDZ)
let y = 20;
```

```
console.log(z); // ReferenceError: Cannot access 'z' before
↪  "initialization" which is called Temporal Dead Zone (TDZ)
const z = 30;
```

```
const x = 10;
x = 20; // TypeError: Assignment to constant variable const variables
↪  cannot be reassigned
```

3.5 Note About Const:

- Declaring a variable with `const` only prevents reassignment of the variable itself; it does not make the value immutable. If the `const` variable refers to an object or array, the contents (properties or elements) can still be changed at **runtime**.
- Don't use `=` to reassign a `const` variable; it will throw a `TypeError` but you can modify the contents of an object or array declared with `const`.

3.6 Differences Between var, let, and const

Feature	var	let	const
Scope	Function or global	Blocked	Blocked
Hoisting	Yes	Yes	Yes
Re-declaration	Yes	No	No

Feature	var	let	const
Re-assignment	Yes	Yes	No
Initialization	Optional	Optional	Required
Temporal Dead Zone	No	Yes	Yes
Performance	Generally slower	Generally faster	Generally The Best

---

4 For of & For in & For Each Loop

```
let allAges = [10, 20, 30, 40, 50];
for (var i = 0; i < allAges.length; i++) {
  console.log(allAges[i]); // 10, 20, 30, 40, 50
}
console.log(i); // 5 (i is accessible outside the loop because var is
↪  function-scoped)
// Why 5 => Because the loop iterates 5 times, and i is incremented after
↪  the last iteration.
```

4.1 for. . . of Loop

The `for...of` loop is used to iterate over iterable objects like arrays, strings, and other collections. It provides a simple way to access each element in the collection.

```
const array = [10, 20, 30, 40, 50];
for (const element of array) {
  console.log(element); // 10, 20, 30, 40, 50
}

const string = "Hello";
for (const char of string) {
  console.log(char); // H, e, l, l, o
}

const set = new Set([10, 20, 30]);
for (const value of set) {
  console.log(value); // 10, 20, 30
}
```

- **Note**: Iterator in for . . . of is equivalent to array[i] in a traditional for loop, but it works with any iterable object, not just arrays.

4.2 for. . . in Loop

The `for...in` loop is used to iterate over the enumerable properties of an object. It is not recommended to use `for...in` with arrays, as it can lead to unexpected behavior.

```
const person = {
  name: "John",
  age: 30,
```

```
  city: "New York",
};

for (const key in person) {
  console.log(key, person[key]); // name John, age 30, city New York
}
```

- **Note**: The `for...in` loop iterates over the keys of an object, not the values. It is primarily used for objects, not arrays.

4.3 forEach Loop

The `forEach` method is an array method that executes a provided function once for each array element. It is a more functional approach to iterating over arrays.

```
const array = [10, 20, 30, 40, 50];
array.forEach((element) => {
  console.log(element); // 10, 20, 30, 40, 50
});
```

- **Note**: The `forEach` method does not return a new array and cannot be used with `break` or `continue` statements. It is purely for side effects.

4.4 Differences Between for...of, for...in, and forEach

Feature	for...of	for...in	forEach
Iterates over	Values of iterable objects	Keys of objects	Elements of arrays
Use case	Arrays, strings, sets, etc.	Objects	Arrays
Break/continue	Yes	Yes	No
Return value	No	No	No
Performance	Generally faster	Generally slower	Generally fast
Scope	Block-scoped	Block-scoped	Block-scoped
Initialization	Yes	Yes	Yes
Re-declaration	No	No	No
Re-assignment	Yes	Yes	No

---

5 Template Literals (ES6)

- Template literals are a new way to work with strings in ES6. They allow for multi-line strings and string interpolation, making it easier to create complex strings without the need for concatenation.

```
const name = "John";
const age = 30;
const message = `My name is ${name} and I am ${age} years old.`;
console.log(message);
```

- Template literals are enclosed by backticks (`) instead of single or double quotes, allowing for easier string interpolation and multi-line strings.
- You can use `${expression}` to embed expressions within a template literal, which will be evaluated and included in the resulting string.

```
const a = 5;
const b = 10;
const sumWithPlus = "The sum of " + a + " and " + b + " is " + a + b + ".";
console.log(sumWithPlus); // The sum of 5 and 10 is 510 cause this is a
↪  string concatenation
const sumWithPlusEquals =
  "The sum of " + a + " and " + b + " is " + (a + b) + ".";
console.log(sumWithPlusEquals); // The sum of 5 and 10 is 15
const sumWithTemplate = `The sum of ${a} and ${b} is ${a + b}.`;
console.log(sumWithTemplate); // The sum of 5 and 10 is 15
```

- Template literals can span multiple lines without the need for concatenation or escape characters.

```
const multiLine = `This is a string
that spans multiple
lines.`;
console.log(multiLine);
```

- You can also use template literals to create HTML strings, making it easier to generate dynamic content.

```
const title = "My Page";
const content = "Welcome to my page!";
const html = `
  <h1>${title}</h1>
  <p>${content}</p>
`;
console.log(html);
```

- Template literals can also be used with tagged templates, allowing you to create custom string processing functions.

```
function tag(strings, ...values) {
  return strings.reduce((result, str, i) => {
    return result + str + (values[i] || "");
  }, "");
}
```

---

6 Parameter Defaults Values (ES6)

- In ES6, you can set default values for function parameters. This allows you to specify a default value that will be used if the argument is not provided or is `undefined`.

```
function multiply(a, b = 3) {
  return a * b;
}
```

```
console.log(multiply()); // NaN (a is undefined)
// NaN Comes From a * b => undefined * 3 => NaN
console.log(multiply(5)); // 15
console.log(multiply(5, 2)); // 10


---
function multiply(a=3, b) {
  return a * b;
}
console.log(multiply()); // NaN (b is undefined)
// NaN Comes From a * b => 3 * undefined => NaN
console.log(multiply(5)); // NaN (b is undefined)
// NaN Comes From a * b => 5 * undefined => NaN
console.log(multiply(5, 2)); // 10

// - If you set a default value for the first parameter, it will be used
↪  if the argument is not provided or is `undefined`.
```

6.1 Order of Parameters

- The order of parameters matters when using default values. If a parameter with a default value is placed before a parameter without a default value, the default value will be used if the first parameter is not provided.

```
function multiply(a = 3, b) {
  return a * b;
}
console.log(multiply()); // NaN (b is undefined)
// NaN Comes From a * b => 3 * undefined => NaN
console.log(multiply(5)); // NaN (b is undefined)
// NaN Comes From a * b => 5 * undefined => NaN
console.log(multiply(5, 2)); // 10
```

---

7 Arrow Functions (ES6)

- Arrow functions are a more concise way to write function expressions in JavaScript. They provide a shorter syntax and lexically bind the `this` value, making them particularly useful in certain contexts.

```
const add = (a, b) => a + b;
console.log(add(5, 10)); // 15
```

- Can we use `this` in arrow functions? Yes, but it behaves differently than in regular functions. In arrow functions, `this` is lexically bound, meaning it refers to the context in which the arrow function was defined, not the context in which it is called.

```
const obj = {
  value: 42,
  regularFunction: function () {
```

```
    console.log(this.value);
  },
  arrowFunction: () => {
    console.log(this.value);
  },
};

obj.regularFunction(); // 42
obj.arrowFunction(); // undefined
```

- In the example above, `this` in the `regularFunction` refers to the `obj` object, while `this` in the `arrowFunction` refers to the global context (or undefined in strict mode).
- Arrow functions can also be used for single-parameter functions without parentheses.

```
const square = (x) => x * x;
console.log(square(5)); // 25
```

- If an arrow function has no parameters, you must use empty parentheses.

```
const greet = () => "Hello, World!";
console.log(greet()); // Hello, World!
```

- If an arrow function has a single expression, you can omit the curly braces and the `return` keyword.

```
const double = (x) => x * 2;
console.log(double(5)); // 10
```

- If an arrow function has multiple parameters, you can use parentheses to enclose them.

```
const add = (a, b) => a + b;
console.log(add(5, 10)); // 15
```

- If a function does not take any parameters, you can use an underscore (_) as a placeholder for the parameter list. This is a common convention to indicate that the parameter is intentionally unused.

```
const greet = (_) => "Hello, World!";
console.log(greet()); // Hello, World!
```

---

8 This Keyword

- The `this` keyword in JavaScript refers to the context in which a function is called. It can be tricky to understand, especially with different types of functions (regular functions, arrow functions, etc.).
- The Value of `this` is always **Object**, but the value of `this` depends on how a function is called, not where it is defined.

```
this; // Window ==> Super global object in non-strict mode
// In non-strict mode, `this` refers to the global object (e.g., `window`
↪  in browsers).
console.log(this); // Super global object (or undefined in strict mode)
```

8.1 Where we can use `this` keyword

1. In the global scope, `this` refers to the global object (e.g., `window` in browsers).

```
console.log(this); // Super global object (In The Global Scope it always
↪  refers to the global object ==> Window in browsers)
```

2. In a regular function, `this` refers to the object that called the function.

```
function regularFunction() {
  console.log(this); // global object (or undefined in strict mode)
}
```

3. In an object method, `this` refers to **the object itself**.

```
const obj = {
  value: 42,
  method: function () {
    console.log(this.value); // 42
  },
};

obj.method(); // 42
```

4. In Events, `this` refers to the element that triggered the event.

```
document.getElementById("myButton").addEventListener("click", function () {
  console.log(this); // The button element that was clicked
});
```

5. In a constructor function, `this` refers to the newly created object.

```
function Person(name) {
  this.name = name;
}
const person = new Person("John");
console.log(person.name); // John
```

6. In an arrow function, `this` is lexically bound, meaning it refers to the context in which the arrow function was defined, not the context in which it is called.

   - **Note:** Arrow functions store (save) the value of `this` before it (the value of `this` inside the arrow function is equal to the same value of `this` outside the arrow function).

```
const obj = {
  value: 42,
  regularFunction: function () {
    console.log(this.value);
  },
  arrowFunction: () => {
    console.log(this.value);
  },
};

obj.regularFunction(); // 42
obj.arrowFunction(); // undefined
```

7. In a class method, `this` refers to the instance of the class.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

const person = new Person("John");
person.greet(); // Hello, my name is John
```

---

9 Deep Accessing Example

```
let obj = {
  name: "Ahmed",
  age: 20,
  salary: 3000,
  sayHello: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
  getSalary: function () {
    function getTaxes() {
      this; // undefined
      // In this case, `this` does not refer to the object, but to the
      ↪  global context (or undefined in strict mode).
      // This is because `getTaxes` is a regular function, not an arrow
      ↪  function.
      return this.salary * 0.1; // This will not work as expected
      // Error: Cannot read properties of undefined (reading 'salary')
    }
  },
};

// Repair the getTaxes function to use the correct context
obj.getSalary = function () {
  const getTaxes = () => {
    return this.salary * 0.1; // Now `this` refers to the object
  };
  return getTaxes();
};
```

9.1 Note To know :

- Lw 3ayz te3r men sa7b el `this`, Es2l nfsk so2aleen :
  1. men sa7b el `this`?

2. men mas2ol 3n calling el function Dy?

```
const x = () => {
  const y = () => {
    const z = () => {
      console.log(this);
    };
    z();
  };
  y();
};

x(); // Window (Even it's in strict mode, it will still refer to the
 ↪  global object)
```

---

10 Destructuring (ES6)

- **Hwa 3amlya 3axya l2y Declaration of References Datatype**
- Destructuring is a convenient way to extract values from arrays or properties from objects into distinct variables. It allows for cleaner and more readable code.
- Normal Nested Object Access:

```
const user = {
  id: 1,
  name: "John",
  address: {
    street: "123 Main St",
    city: "New York",
    zip: "10001",
  },
};

const street = user.address.street;
const city = user.address.city;
const zip = user.address.zip;

console.log(street, city, zip); // 123 Main St New York 10001
```

10.1 Destructuring Objects

- You can use destructuring to extract properties from an object into variables.

```
const user = {
  id: 1,
  name: "John",
  address: {
    street: "123 Main St",
    city: "New York",
    zip: "10001",
```

```
  },
};

const { street, city, zip } = user.address;
// street, city, and zip are now variables that hold the respective values
↪  from the user.address object
// So they are 3 vars of the same name of the properties in the object

console.log(street, city, zip); // 123 Main St New York 10001

---

// If i have a variable with the same name as the property, i can use the
↪  colon to rename it

let street = "456 Elm St"; // This is a different variable
const { street: userStreet, city: userCity, zip: userZip } = user.address;
// Now userStreet, userCity, and userZip are variables that hold the
↪  respective values from the user.address object

console.log(userStreet, userCity, userZip); // 123 Main St New York 10001
```

10.2 Destructuring Arrays

- You can also use destructuring to extract values from an array into variables.

```
const numbers = [10, 20, 30, 40, 50];
const [first, second, third] = numbers;
console.log(first, second, third); // 10 20 30
// first, second, and third are now variables that hold the respective
↪  values from the numbers array
const [first, , , fourth] = numbers; // Skip the second and third elements
console.log(first, fourth); // 10 40
const [first, ...rest] = numbers; // rest will contain the remaining
↪  elements
console.log(first, rest); // 10 [20, 30, 40, 50]
```

---

11 Shallow Copying and Deep Copying

- Shallow copying creates a new object, but the properties are still references to the original object's values.
- Deep copying creates a new object and recursively copies all properties and nested objects.
- In JavaScript, you can create a shallow copy using the spread operator or `Object.assign()`, while deep copying can be done using libraries like Lodash or by using JSON methods.

11.1 Shallow Copying

- Shallow copying creates a new object, but the properties are still references to the original object's values. This means that if you modify a nested object in the copied object, it will

also affect the original object.

- Shallow copying can be done using the spread operator (`...`) or `Object.assign()`.

```
const original = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
  },
};

const shallowCopy = { ...original };
shallowCopy.name = "Jane";
shallowCopy.address.city = "Los Angeles";

console.log(original.name); // John
console.log(original.address.city); // Los Angeles
```

- Shallow Copying With Array

```
let allAges = [10, 20, 30, 40, 50];

let newAges = allAges; // This is a shallow copy, both variables point to
↪  the same array
newAges[0] = 100; // Modifying the first element in newAges will also
↪  modify allAges
console.log(allAges); // [100, 20, 30, 40, 50]
console.log(newAges); // [100, 20, 30, 40, 50]
```

11.2 Spread Operator

- The spread operator (`...`) can be used to create a shallow copy of an object or array. It allows you to expand the properties of an object or elements of an array into a new object or array.

```
const original = { a: 1, b: 2, c: 3 };
const copy = { ...original };
console.log(copy); // { a: 1, b: 2, c: 3 }

const numbers = [1, 2, 3];
console.log(...numbers); // 1,2,3
const newNumbers = [...numbers]; // [1, 2, 3]
console.log(newNumbers); // [1, 2, 3]
```

11.3 Object.assign()

- The `Object.assign()` method can also be used to create a shallow copy of an object. It copies the values of all enumerable own properties from one or more source objects to a target object.

```
const original = { a: 1, b: 2, c: 3 };
```

```
const copy = Object.assign({}, original);
console.log(copy); // { a: 1, b: 2, c: 3 }

const numbers = [1, 2, 3];
const newNumbers = Object.assign([], numbers);
console.log(newNumbers); // [1, 2, 3]
```

11.4 Deep Copying

- Deep copying creates a new object and recursively copies all properties and nested objects. This means that changes made to the copied object will not affect the original object.

```
const original = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
  },
};

const deepCopy = JSON.parse(JSON.stringify(original)); // Create a deep
↪  copy using JSON methods
deepCopy.name = "Jane";
deepCopy.address.city = "Los Angeles";

console.log(original.name); // John
console.log(original.address.city); // New York
```

- Deep copying can be done using libraries like Lodash or by using JSON methods (`JSON.parse()` and `JSON.stringify()`).

11.5 JSON Methods

- The `JSON.parse()` and `JSON.stringify()` methods can be used to create a deep copy of an object. This method works well for objects that can be represented as JSON, but it does not handle functions or special object types like `Date` or `RegExp`.

```
const original = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
  },
};

const deepCopy = JSON.parse(JSON.stringify(original));
deepCopy.name = "Jane";
deepCopy.address.city = "Los Angeles";
```

```
console.log(original.name); // John
console.log(original.address.city); // New York
```

11.6 Spread Operator for Deep Copying

- The spread operator can also be used for deep copying, but it only works for shallow copies. For deep copying, you need to use a combination of the spread operator and recursion or use libraries like Lodash.

```
const original = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
  },
};

const deepCopy = { ...original, address: { ...original.address } };
deepCopy.name = "Jane";
deepCopy.address.city = "Los Angeles";

console.log(original.name); // John
console.log(original.address.city); // New York
```

11.7 Spread Operator for Deep Copying

- The spread operator can also be used for deep copying, but it only works for shallow copies. For deep copying, you need to use a combination of the spread operator and recursion or use libraries like Lodash.

```
const original = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
  },
};
const deepCopy = { ...original, address: { ...original.address } };
deepCopy.name = "Jane";
deepCopy.address.city = "Los Angeles";
console.log(original.name); // John
console.log(original.address.city); // New York
```

---

12 Rest Parameters & Spread Operator

- Rest parameters allow you to represent an indefinite number of arguments as an array. This is useful when you want to create functions that can accept a variable number of

arguments.

```
function getSum(arrayOfNumbers) {
  let sum = 0;
  for (const number of arrayOfNumbers) {
    sum += number;
  }
  console.log(sum);
}
getSum([1, 2, 3, 4, 5]); // 15

// We Can Do This Using "Rest Parameters"
function getSum(...arrayOfNumbers) {
  let sum = 0;
  for (const number of arrayOfNumbers) {
    sum += number;
  }
  console.log(sum);
}
getSum(1, 2, 3, 4, 5); // 15
getSum(...[1, 2, 3, 4, 5]); // 15
```

12.0.1 Order of Params in Function

- When using rest parameters, they must be the last parameter in the function definition. You can have other parameters before the rest parameter, but the rest parameter must come last.

```
function getSum(a, b, ...rest) {
  // Start by summing a and b, then add the rest of the numbers in the
  ↪  loop
  let sum = a + b;
  for (const number of rest) {
    sum += number;
  }
  console.log(sum);
}
getSum(1, 2, 3, 4, 5); // 15

getSum(1, 2, ...[3, 4, 5]); // 15
```

Note: If a parameter has a default value, it must be declared before the rest parameter in the function definition. The rest parameter must always be the last parameter.

```
function getSum(a, b = 0, ...rest) {
  let sum = a + b;
  for (const number of rest) {
    sum += number;
  }
  console.log(sum);
}
```

```
// Example calls:
getSum(1, 2, 3, 4, 5); // 15
getSum(1, 2, ...[3, 4, 5]); // 15

// If you omit the second argument, b will take the default value 0, and
↪  rest will be empty:
getSum(1); // 1 (a = 1, b = 0, rest = [])
getSum(1, 2, 3, 4, 5); // 15
getSum(1, 2, ...[3, 4, 5]); // 15
```

---

13 ES6 Array Methods (Map, Filter, Reduce)

- ES6 introduced several new array methods that make it easier to work with arrays. These methods include `map`, `filter`, and `reduce`, which allow for functional programming-style operations on arrays.
- These methods are often used to transform, filter, or aggregate data in arrays.

13.1 Filter Array Method

- The `filter` method **creates a new array with all elements** that pass the test implemented by the provided function. It does not modify the original array.
- It is used to filter out elements from an array based on a condition.
- It takes a callback function as an argument, which is called for each element in the array. If the callback returns `true`, the element is included in the new array; otherwise, it is excluded.
- It returns a new array containing only the elements that satisfy the condition specified in the callback function.

```
let allAges = [30, 20, 10, 40, 50];
let filteredAges = allAges.filter((age) => age > 20);
console.log(filteredAges); // [30, 40, 50]
```

13.2 Map Array Method

- The `map` method creates a new array with the results of calling a provided function on every element in the original array. It does not modify the original array.
- It is used to transform each element in an array into a new value.
- It takes a callback function as an argument, which is called for each element in the array. The return value of the callback function is used to create the new array.
- The `map` method applies a Operation to each element in the array and returns a new array of the same length, where each element is the result of the Operation.

For loop equivalent of the map method:

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

```
for (let i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2);
}
console.log(doubled); // [2, 4, 6, 8, 10]
```

Using map:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((num) => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]

---
const doubled = numbers.map((num) => "Test");
console.log(doubled); // ["Test", "Test", "Test", "Test", "Test"]

---


const doubled = numbers.map((num) => `<li>${age}</li>`);
console.log(doubled); // ["<li>1</li>", "<li>2</li>", "<li>3</li>",
↪  "<li>4</li>", "<li>5</li>"]
```

13.3 Reduce Array Method

- The `reduce` method executes a reducer function (that you provide) on each element of the array, resulting in a single output value. It is used to reduce an array to a single value by applying a function to each element.
- It takes a callback function as an argument, which is called for each element in the array. The callback function takes two arguments: an accumulator (which accumulates the result) and the current value being processed.
- The `reduce` method applies a Operation to each element in the array and returns a single value, which is the result of the Operation applied to all elements.
- It takes 2 Parameters: a callback function and an initial value for the accumulator.

```
let allAges = [10, 20, 30, 40, 50];
let totalAge = allAges.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
}, 0); // Initial value is 0
// Trace : [10, 30, 60, 100, 150]
// return Only One Value
console.log(totalAge); // 150
```

13.4 Deference between Map, Filter, and Reduce

Feature	map	filter	reduce
Purpose	Transforms each element	Filters elements based on condition	Reduces array to single value
Returns	New array of transformed elements	New array of filtered elements	Single value (accumulated result)

Feature	map	filter	reduce
Modifies Original Array	No	No	No
Parameters	callback function	callback function	callback function, initial value

13.5 Summary

- **map**: Use when you want to transform each element in an array.
- **filter**: Use when you want to remove elements from an array based on a condition.
- **reduce**: Use when you want to combine all elements in an array into a single value.

---

14 Set Objects

- A `Set` is a built-in JavaScript object that allows you to store unique values of any type, whether primitive values or object references. It is similar to an array but does not allow duplicate values.
- Sets are useful when you want to ensure that a collection of values does not contain duplicates.
- You can create a `Set` using the `Set` constructor, and you can add values to it using the `add` method. You can also check for the existence of a value using the `has` method and remove values using the `delete` method.
- Set is Iterable Object, meaning you can use it in loops like `for...of` or `forEach`.

```
// Flag Concept
let userValue = 15;
let allAges = [30,15,50];
let flag = false;
for(let age of allAges) {
  if(age === userValue) {
    flag = true;
    break;
  }
}
// If the value does not exist in the array, add it
if(flag == false) {
  allAges.push(userValue); // Unique value added
} else {
  console.log("Value already exists");
}

---
> **Note:** When you create a `Set` from an array with duplicate elements,
↪  only unique values are kept. Any repeated elements in the array are
↪  automatically removed by the `Set`.
```

```
let allAges = new Set([30, 15, 50, 30]); // Create a Set with Unique
↪  initial values
allAges.add(15); // Adding a duplicate value (will not be added)
allAges.add(20); // Adding a new value
console.log(allAges); // Set { 30, 15, 50, 20 }
console.log(allAges.has(15)); // true (check if value exists)
allAges.delete(15); // Remove a value
console.log(allAges); // Set { 30, 50, 20 }

// We can use the concept of chaining to add multiple values at once
allAges.add(25).add(35).add(45); // Adding multiple values
console.log(allAges); // Set { 30, 50, 20, 25, 35, 45 }

console.log(allAges.size); // 6 (get the number of unique values in the
↪  Set)

console.log(allAges.has(30)); // true (check if value exists)

console.log(allAges.clear()); // Clear all values in the Set
console.log(allAges); // Set {} (empty Set)

---

> **Note:** The `Set` object is iterable, meaning you can use it in loops
↪  like `for...of` or `forEach`.

for (const age of allAges) {
  console.log(age); // 30, 15, 50, 20, 25, 35, 45
}

allAges.forEach((age) => {
  console.log(age); // 30, 15, 50, 20, 25, 35, 45
});
```

14.1 Converting Set to Array

- You can convert a `Set` to an array using the spread operator or the `Array.from()` method.

```
let allAges = new Set([30, 15, 50, 20]);
let agesArray = [...allAges]; // Using spread operator
console.log(agesArray); // [30, 15, 50, 20]

---

let agesArray2 = Array.from(allAges); // Using Array.from()
console.log(agesArray2); // [30, 15, 50, 20]
```

---

15 Map Objects

- A `Map` is a **built-in JavaScript object** that allows you to store key-value pairs.
- It is similar to an object but provides better performance for certain operations and allows keys of any type, including objects.
- Maps are useful when you want to associate values with keys and need to maintain the order of insertion.
- You can create a `Map` using the `Map` constructor, and you can add key-value pairs using the `set` method. You can also retrieve values using the `get` method and check for the existence of a key using the `has` method.

```
let ageMap = new Map();
ageMap.set("Alice", 30);
ageMap.set("Bob", 25);
ageMap.set("Charlie", 35);

console.log(ageMap.get("Alice")); // 30
console.log(ageMap.has("Bob")); // true
ageMap.delete("Charlie");
console.log(ageMap.size); // 2

// "has" method returns true if the key exists in the Map
console.log(ageMap.has("Charlie")); // false
ageMap.clear(); // Clear all key-value pairs
console.log(ageMap.size); // 0 (empty Map)

// Adding key-value pairs using chaining
ageMap.set("Dave", 40).set("Eve", 28).set("Frank", 32);
console.log(ageMap); // Map { 'Dave' => 40, 'Eve' => 28, 'Frank' => 32 }

// Get the value associated with a key
console.log(ageMap.get("Eve")); // 28
```

15.1 Iterating Over a Map

- You can iterate over a `Map` using the `for...of` loop or the `forEach` method.

```
for (const [key, value] of ageMap) {
  console.log(`${key}: ${value}`); // Dave: 40, Eve: 28, Frank: 32
}

ageMap.forEach((value, key) => {
  console.log(`${key}: ${value}`); // Dave: 40, Eve: 28, Frank: 32
});

---

for(const x of ageMap) {
  console.log(x); // ['Dave', 40], ['Eve', 28], ['Frank', 32]
  console.log(x[0], x[1]); // Dave 40, Eve 28, Frank 32
}
```

```
// If i want to get Keys Only

const keys = [...ageMap.keys()];
console.log(keys); // ['Dave', 'Eve', 'Frank']

for(const key of ageMap.keys()) {
  console.log(key); // Dave, Eve, Frank
}

// If i want to get Values Only
const values = [...ageMap.values()];
console.log(values); // [40, 28, 32]

for(const value of ageMap.values()) {
  console.log(value); // 40, 28, 32
}
// If i want to get Entries Only
const entries = [...ageMap.entries()];
console.log(entries); // [['Dave', 40], ['Eve', 28], ['Frank', 32]]

for(const entry of ageMap.entries()) {
  console.log(entry); // ['Dave', 40], ['Eve', 28], ['Frank', 32]
  console.log(entry[0], entry[1]); // Dave 40, Eve 28, Frank 32
}
```

15.2 If i have an Object and i want to convert it to a Map

- You can convert an object to a `Map` using the `Object.entries()` method, which returns an array of key-value pairs, and then passing that array to the `Map` constructor.
- **Entry:** is an array of two elements, the first element is the key and the second element is the value.

```
const ageObject = {
  Dave: 40,
  Eve: 28,
  Frank: 32,
};
Object.entries(ageObject); // [['Dave', 40], ['Eve', 28], ['Frank', 32]]
const ageMapFromObject = new Map(Object.entries(ageObject));
console.log(ageMapFromObject); // Map { 'Dave' => 40, 'Eve' => 28, 'Frank'
↪  => 32 }
```

15.3 If i have a Map and i want to convert it to an Object

- You can convert a `Map` to an object using the `Object.fromEntries()` method, which takes an iterable of key-value pairs (like a `Map`) and returns an object.

```
const ageMap = new Map([
  ["Dave", 40],
  ["Eve", 28],
```

```
  ["Frank", 32],
]);
const ageObjectFromMap = Object.fromEntries(ageMap);
console.log(ageObjectFromMap); // { Dave: 40, Eve: 28, Frank: 32 }
```

15.4 Difference between Map and Object

```
// 1. Key Types
const obj = {};
const map = new Map();

obj["stringKey"] = "value"; // string keys only
map.set("stringKey", "value");
map.set(42, "value"); // keys can be of any type

// 2. Iteration
for (const key in obj) {
  console.log(key, obj[key]);
}

map.forEach((value, key) => {
  console.log(key, value);
});

// 3. Performance
// Maps are generally more performant for frequent additions and removals
↪  of key-value pairs.
// Objects are faster for simple key-value pairs and when the keys are
↪  known in advance.
```

15.5 Interview Question

- **If you have a string and you need to know what is the most repeated char ?**
- You can use a Map to count the occurrences of each character in the string and then find the character with the maximum count.

```
function mostRepeatedChar(str) {
  const charCount = new Map(); // Create a Map to store character counts

  for (const char of str) {
    charCount.set(char, (charCount.get(char) || 0) + 1);
  }

  let maxChar = "";
  let maxCount = 0;

  for (const [char, count] of charCount.entries()) {
    if (count > maxCount) {
      maxCount = count;
      maxChar = char;
    }
```

```
  }

  return maxChar;
}
// Explain and Trace Code
// 1. Create a Map to store character counts
// 2. Loop through each character in the string
// set(char, (charCount.get(char) || 0) + 1) : This line updates the count
↪  for each character, if it exists, or initializes it to 1
// (charCount.get(char) || 0) : Get the current count of the character
↪  from the Map. If the character doesn't exist yet (i.e., get() "returns
↪  undefined"), treat the count as 0.
// 3. If the character is already in the Map, increment its count
// 4. If it's not, add it to the Map with a count of 1
// 5. After counting, find the character with the maximum count
// 6. Return the character with the highest count
console.log(mostRepeatedChar("hello world")); // "l" (or "o", depending on
↪  the implementation)


---
// Using get
function mostRepeatedChar(str) {
  const charCount = new Map();

  for (const char of str) {
    if (charCount.has(char)) {
      charCount.set(char, charCount.get(char) + 1);
    } else {
      charCount.set(char, 1);
    }
  }

  let maxChar = "";
  let maxCount = 0;

  for (const [char, count] of charCount.entries()) {
    if (count > maxCount) {
      maxCount = count;
      maxChar = char;
    }
  }

  return maxChar;
}

// If there an 2 chars with the same count ?

function mostRepeatedChar(str) {
  const charCount = new Map();
```

```
  for (const char of str) {
    charCount.set(char, (charCount.get(char) || 0) + 1);
  }

  let maxChar = "";
  let maxCount = 0;

  for (const [char, count] of charCount.entries()) {
    if (count > maxCount) {
      maxCount = count;
      maxChar = char;
    } else if (count === maxCount) {
      maxChar += `, ${char}`; // Append if there's a tie
    }
  }

  return maxChar;

}
console.log(mostRepeatedChar("hhii")); // "h, i"
```

15.5.1 Can we Trace this code Step by Step ?

- Let's trace the following code for `mostRepeatedChar("hello world")`:
- Step-by-step Trace
- 1. Create a Map to store character counts
- `charCount = Map {}`
- 2. Loop through each character in the string

For `"hello world"`:

Step	char	charCount after step
1	'h'	Map { 'h' => 1 }
2	'e'	Map { 'h' => 1, 'e' => 1 }
3	'l'	Map { 'h' => 1, 'e' => 1, 'l' => 1 }
4	'l'	Map { 'h' => 1, 'e' => 1, 'l' => 2 }
5	'o'	Map { 'h' => 1, 'e' => 1, 'l' => 2, 'o' => 1 }
6	' '	Map { ..., ' ' => 1 }
7	'w'	Map { ..., 'w' => 1 }
8	'o'	Map { ..., 'o' => 2 }
9	'r'	Map { ..., 'r' => 1 }
10	'l'	Map { ..., 'l' => 3 }
11	'd'	Map { ..., 'd' => 1 }

Final `charCount`:

```
Map {
  'h' => 1,
  'e' => 1,
  'l' => 3,
  'o' => 2,
  ' ' => 1,
  'w' => 1,
  'r' => 1,
  'd' => 1
}
```

- 3. Find the character with the maximum count

Initialize: `maxChar = ""`, `maxCount = 0`

Iterate through `charCount.entries()`:

char	count	maxCount before	maxChar before	maxCount after	maxChar after
'h'	1	0	""	1	'h'
'e'	1	1	'h'	1	'h'
'l'	3	1	'h'	3	'l'
'o'	2	3	'l'	3	'l'
' '	1	3	'l'	3	'l'
'w'	1	3	'l'	3	'l'
'r'	1	3	'l'	3	'l'
'd'	1	3	'l'	3	'l'

- 4. Return the character with the highest count
- Result: `'l'`

Summary:

- After counting, `charCount` holds the frequency of each character.
- The loop finds `'l'` as the most repeated character (appearing 3 times).

---

16 Difference between `Map` and `Set`

Feature	Map	Set
Purpose	Key-value pairs	Unique values
Key Access	Keys can be any value	Values must be unique
Order	Maintains insertion order	No guaranteed order
Iteration	Iterates over key-value pairs	Iterates over values only
Size	Size can be obtained via `size`	Size can be obtained via `size`

16 Difference between `Map` and `Set`

Feature	Map	Set
Methods	`set`, `get`, `has`, `delete`	`add`, `has`, `delete`, `clear`
Use Cases	Associative arrays, caching	Unique collections, flags

---