JavaScript - Day 3

Table of Contents:

ES6 (ECMAScript 2015):

ES6 (also called ECMAScript 2015) is a major update to JavaScript that introduced new features to make coding easier, cleaner, and more powerful.

i) let:

Used to declare block-scoped variables.

Value can be changed after declaration.

```
let name = "Sam";
name = "Chris"; // valid
```

ii)const:

Also block-scoped, but value cannot be reassigned.

```
const age = 25;
age = 30; // Error: Assignment to constant variable
```

key Difference:

Feature	`var`	`let` / `const`
Scope	Function-scoped	Block-scoped
Hoisting	Hoisted + initialized as `undefined`	Hoisted but not initialized
Re-declare	Allowed	❌ Not allowed in same scope

iii) Arrow Function:

Arrow functions are a shorter syntax for writing function expressions, and they do not have their own this context.

```
// Traditional function
function add(a, b) {
  return a + b;
}

// Arrow function version
const add = (a, b) => a + b;
```

Example:

```
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
```

Usage: Used for callbacks, array methods, and shorter functions without their own this.

iv) Template Literals:

Template literals allow us to embed variables and expressions inside strings using backticks (`) instead of quotes.

```
const name = "Sam";
const greeting = `Hello, ${name}!`;
console.log(greeting);  // Output: Hello, Sam!
```

Usage: Used for easy string formatting with embedded expressions.

v) Destructuring:

Destructuring is a way to unpack values from arrays or properties from objects into separate variables.

Array Destructuring:

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

Object Destructuring:

```
const user = { name: "Sam", age: 22 };
const { name, age } = user;
console.log(name); // Sam
console.log(age);  // 22
```

vi) Spread Operator (...):

The spread operator is used to expand elements of an array or object.

Example – Spread with Arrays:

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

Example – Spread with Objects:

```
const obj1 = { name: "Sam" };
const obj2 = { ...obj1, age: 22 };
console.log(obj2); // { name: "Sam", age: 22 }
```

vii) Rest Operator (...):

The rest operator is used to group remaining elements into an array or gather function arguments.

Example – Rest in Arrays:

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

Example – Rest in Functions:

```
function sum(...nums) {
  return nums.reduce((a, b) => a + b);
}
console.log(sum(1, 2, 3)); // 6
```

Usage: Spread expands, rest gathers.

viii) Default Parameters:

Default parameters allow functions to have default values if no arguments are passed.

```
function greet(name = "Guest") {
  console.log(`Hello, ${name}!`);
}
greet();         // Hello, Guest!
greet("Chris");  // Hello, Chris!
```

Usage: Provides fallback values in functions when no argument is given.

ix) Class:

A class in JavaScript is a blueprint for creating objects with shared properties and methods.

Example:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  }
}

const sam = new Person("Sam", 22);
sam.greet(); // Output: Hi, I'm Sam and I'm 22 years old.
```

Usage: Used to create multiple objects with the same structure and behavior.

x) Modules:

Modules allow you to split your code into separate files and use export to share code and import to use it elsewhere.

Example:

math.js

```
export const add = (a, b) => a + b;
export const sub = (a, b) => a - b;
```

main.js

```
import { add, sub } from './math.js';

console.log(add(2, 3)); // 5
console.log(sub(5, 2)); // 3
```

Usage: Used to organize and reuse code across multiple files cleanly.

xi) Promises:

A Promise is a JavaScript object that represents the eventual completion (or failure) of an asynchronous operation.

Example:

```
let promise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
    resolve("Success!");
  } else {
    reject("Failed!");
  }
});

promise
  .then(result => console.log(result))  // Success!
  .catch(error => console.log(error));
```

Usage: Used to handle asynchronous tasks like API calls in a cleaner way than callbacks.

xii) for...of loop:

for...of is used to loop over iterable objects like arrays, strings, maps, sets, etc.

Syntax:

```
for (let item of iterable) {
  // code block
}
```

Example:

```
let fruits = ["apple", "banana", "mango"];
for (let fruit of fruits) {
  console.log(fruit);
}
// Output:
// apple
// banana
// mango
```

xiii) Enhanced Object Literals:

Enhanced object literals allow you to create objects using a shorter syntax and dynamic property names.

Example:

```
let name = "Sam";
let age = 25;

let person = {
  name,      // same as name: name
  age,       // same as age: age
  greet() {  // method shorthand
    console.log("Hello!");
  }
};

console.log(person.name); // Sam
person.greet();           // Hello!
```

Features:

- Shorthand for properties and methods
- Computed property names ([propName]: value)

xiv) Symbol:

A Symbol is a unique and immutable data type used as a key for object properties to avoid name conflicts.

Example:

```
const id = Symbol("id");
const user = {
  name: "Alice",
  [id]: 101
};

console.log(user[id]); // 101
```

Why Use Symbols?

- Each symbol is unique, even if it has the same description.
- Useful for defining hidden or special object properties.

Sets:

A JavaScript Set is a collection of unique values. Each value can only occur once in a Set.

You can create a JavaScript Set by:

- Passing an array to new Set()
- Create an empty set and use add() to add values

Create a Set and add values:

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

// Add Values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
```

Create a Set and add variables:

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

// Create Variables
const a = "a";
const b = "b";
const c = "c";

// Add Variables to the Set
letters.add(a);
letters.add(b);
letters.add(c);
```

Set Methods:

i) new Set() Method:

Pass an array to the new Set() constructor:

```
// Create a Set
const letters = new Set(["a","b","c"]);
```

ii) add() Method:

The add() method adds values to a Set.

```
letters.add("d");
letters.add("e");
```

If you add equal elements, only the first will be saved:

```
letters.add("a");
letters.add("b");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
```

Listing Set Elements:

You can list all Set elements (values) with a for..of loop:

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Elements
let text = "";
for (const x of letters) {
  text += x;
}
```

//Output:

a

b

c

iii) has() Method:

The has() method returns true if a specified value exists in a set.

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Does the Set contain "d"?
answer = letters.has("d");
```

iv) forEach() Method:

The forEach() method invokes a function for each Set element.

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all entries
let text = "";
letters.forEach (function(value) {
  text += value;
})
```

v) values() Method:

The values() method returns an Iterator object with the values in a Set.

Example 1:

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Get all Values
const myIterator = letters.values();

// List all Values
let text = "";
for (const entry of myIterator) {
  text += entry;
}
```

Example 2:

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Values
let text = "";
for (const entry of letters.values()) {
  text += entry;
}
```

//Output:

a

b

c

vi) keys() Method:

The keys() method returns an Iterator object with the values in a Set.

Note: A Set has no keys, so keys() returns the same as values().

Example 1:

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Create an Iterator
const myIterator = letters.keys();

// List all Elements
let text = "";
for (const x of myIterator) {
  text += x;
}
```

Example 2:

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Elements
let text = "";
for (const x of letters.keys()) {
  text += x;
}
```

//Output:

a

b

c

vii) The entries() Method:

The entries() method returns an Iterator with [value,value] pairs from a Set.

Note: The entries() method is supposed to return a [key,value] pair from an object. A Set has no keys, so the entries() method returns [value,value].

Example 1:

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Get all Entries
const myIterator = letters.entries();

// List all Entries
let text = "";
for (const entry of myIterator) {
  text += entry;
}
```

Example 2:

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Entries
let text = "";
for (const entry of letters.entries()) {
  text += entry;
}
```

//Output:

a,a

b,b

c,c

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:

i) new Map() Method:

You can create a Map by passing an Array to the new Map() constructor:

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);

let numb = fruits.get("apples");//500
```

ii) Map.get():

You get the value of a key in a map with the get() method.

```
fruits.get("apples");
```

//Output: 500

iii) Map.set():

You can add elements to a map with the set() method:

```
// Create a Map
const fruits = new Map();

// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);

let numb = fruits.get("apples"); //500
```

The set() method can also be used to change existing map values:

```
fruits.set("apples", 500);
```

iv) Map.size:

The size property returns the number of elements in a map:

```
fruits.size;
```

//Output: 3

v) Map.delete():

The delete() method removes a map element:

```
fruits.delete("apples");
```

vi) Map.clear():

The clear() method removes all the elements from a map:

```
fruits.clear();
```

vii) Map.has():

The has() method returns true if a key exists in a map:

```
fruits.has("apples");
```

//true

viii) Map.forEach():

The forEach() method invokes a callback for each key/value pair in a map:

```
// List all entries
let text = "";
fruits.forEach (function(value, key) {
  text += key + ' = ' + value;
})
```

//Output:

apples = 500

bananas = 300

oranges = 200

ix) Map.entries():

The entries() method returns an iterator object with the [key,values] in a map:

```
// List all entries
let text = "";
for (const x of fruits.entries()) {
  text += x;
}
```

//Output:

apples,500

bananas,300

oranges,200

x) Map.keys():

The keys() method returns an iterator object with the keys in a map:

```
// List all keys
let text = "";
for (const x of fruits.keys()) {
  text += x;
}
```

//Output:

apples

bananas

oranges

xi) Map.values():

The values() method returns an iterator object with the values in a map:

```
// List all values
let text = "";
for (const x of fruits.values()) {
  text += x;
}
```

//Output:

500

300

200

You can use the values() method to sum the values in a map:

```
// Sum all values
let total = 0;
for (const x of fruits.values()) {
  total += x;
}
```

//Output: 1000

Objects as Keys:

```
// Create Objects
const apples = {name: 'Apples'};
const bananas = {name: 'Bananas'};
const oranges = {name: 'Oranges'};

// Create a Map
const fruits = new Map();

// Add new Elements to the Map
fruits.set(apples, 500);
fruits.set(bananas, 300);
fruits.set(oranges, 200);

console.log(fruits.get(apples));
```

Remember: The key is an object (apples), not a string ("apples"):

```
fruits.get("apples");  // Returns undefined
```

Iterators:

An iterator is an object that allows you to go through a collection (like an array) one item at a time using a .next() method.

Example:

```
const myArray = [10, 20, 30];
const iterator = myArray[Symbol.iterator]();

console.log(iterator.next()); // { value: 10, done: false }
console.log(iterator.next()); // { value: 20, done: false }
console.log(iterator.next()); // { value: 30, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

Generator:

A generator is a special function that can pause (yield) and resume its execution, producing values one at a time on demand.

Example:

```
function* myGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = myGenerator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Usage: Used for custom iteration logic, lazy evaluation, and handling infinite sequences or asynchronous data flows.

Array Methods:

Method	Description (Short)	Example
`find()`	Returns first value that passes a test.	`[1,2,3].find(x => x > 1)` → 2
`findIndex()`	Returns index of first value that passes a test.	`[1,2,3].findIndex(x => x > 1)` → 1
`includes()`	Checks if array contains a value.	`[1,2,3].includes(2)` → true
`some()`	Checks if **any** value passes a test.	`[1,2,3].some(x => x > 2)` → true
`every()`	Checks if **all** values pass a test.	`[1,2,3].every(x => x > 0)` → true
`fill()`	Fills elements with a static value.	`[1,2,3].fill(0)` → [0,0,0]
`copyWithin()`	Copies part of array within itself.	`[1,2,3,4].copyWithin(0,2)` → [3,4,3,4]

String Methods:

Method	Description	Example
`startsWith()`	Checks if string starts with given text.	`"hello".startsWith("he")` → true
`endsWith()`	Checks if string ends with given text.	`"hello".endsWith("lo")` → true
`includes()`	Checks if string contains given text.	`"hello".includes("ll")` → true
`repeat()`	Repeats string.	`"hi".repeat(3)` → `"hihihi"`