JavaScript - Day 2

Table of Contents:

JavaScript Functions:

1. Desc

A function is a reusable block of code designed to perform a specific task when called.

Syntax:

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Usage: Functions are used to reuse and organize code.

Advantage: Functions make code reusable and easier to manage.

Disadvantage: Too many functions can make code harder to trace and debug.

2. Parameters and arguments:

Function parameters are listed inside the parentheses () in the function definition.

Function arguments are the values received by the function when it is invoked.

```
function myFunction(p1, p2) {
  return p1 * p2;
}

let result = myFunction(4, 3); //12
```

3. Function Return:

The return statement is used to stop a function and send a value back to where the function was called.

```
// Function is called, the return value will end up in x
let x = myFunction(4, 3); //Output: 12

function myFunction(a, b) {
// Function returns the product of a and b
  return a * b;
}
```

Functions with Default Arguments:

Default arguments allow you to assign default values to parameters in case no value is passed during the function call.

Example:

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

greet();          // Output: Hello, Guest!
greet("Sam");     // Output: Hello, Sam!
```

Functions with Variable Length Arguments:

Functions can accept any number of arguments using the rest parameter syntax (...), which gathers them into an array.

Example:

```
function sum(...numbers) {
  let total = 0;
  for (let num of numbers) {
    total += num;
  }
  return total;
}

console.log(sum(1, 2, 3));        // Output: 6
console.log(sum(4, 5, 6, 7, 8));  // Output: 30
```

Generator Function:

A generator function is a special type of function that can pause its execution using the yield keyword and resume later, allowing lazy evaluation.

Usage: Used for lazy iteration, handling asynchronous flows, or producing data on demand.

Syntax:

```
function* generatorFunction() {
  yield value1;
  yield value2;
  // ...
}
```

Example:

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

let counter = countUp();

console.log(counter.next().value); // Output: 1
console.log(counter.next().value); // Output: 2
console.log(counter.next().value); // Output: 3
console.log(counter.next().value); // Output: undefined
```

Explanation: Each call to .next() resumes the function from where it last paused at yield.

Advantage: Allows efficient memory usage by pausing and resuming execution.

Disadvantage: More complex to read, write, and debug compared to regular functions.

Function Expression:

A function expression defines a function and assigns it to a variable, allowing functions to be created anonymously and used like values.

Example:

```
const greet = function(name) {
  return "Hello, " + name;
};

console.log(greet("Sam")); // Output: Hello, Sam
```

Explanation: The function has no name and is stored in the variable greet.

Arrow Function:

Arrow functions were introduced in ES6 and it allow us to write shorter function syntax.

Example:

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

console.log(add(5, 3)); // Output: 8
```

Arrow Functions Return Value by Default:

```
const hello = () => "Hello World!";
console.log(hello()); //Output: Hello World!
```

Arrow Function Without Parentheses:

If you have only one parameter, you can skip the parentheses as well.

```
let hello = "";
hello = val => "Hello " + val;

console.log(hello("Universe!")); //Hello Universe!
```

Nested Function:

A nested function is a function defined inside another function, and it can access the variables of its outer function.

Example:

```
function outer() {
  let message = "Hello";

  function inner() {
    console.log(message);
  }

  inner();
}

outer(); // Output: Hello
```

Explanation: The inner function is nested inside outer and can use its variables.

Hoisting:

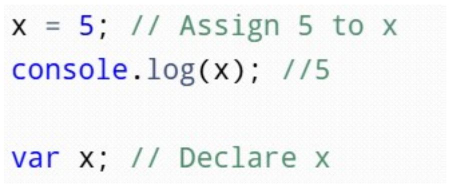
Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope.

Note:

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

Example:



```
x = 5; // Assign 5 to x
console.log(x); //5

var x; // Declare x
```

The var, let and const Keywords in Hosting:

- var is hoisted and given a value undefined.

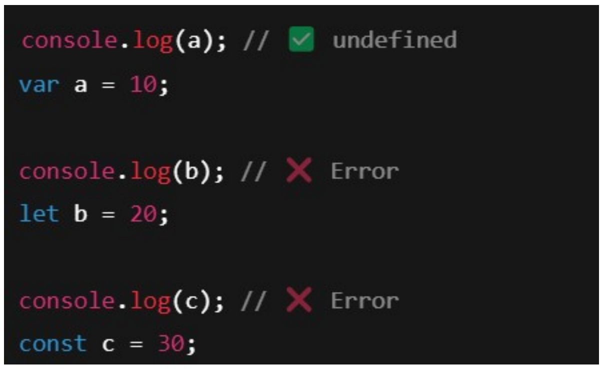
✅ You can use it before declaring (but value will be undefined).

- let and const are hoisted but not given any value.

❌ You cannot use them before declaring — it gives an error

Note: Hoisted means the declaration of the variable is moved to the top of the code before the code runs.

Example:



```
console.log(a); // ✅ undefined
var a = 10;

console.log(b); // ❌ Error
let b = 20;

console.log(c); // ❌ Error
const c = 30;
```

Advantage: Lets you use functions and variables before declaration.

Disadvantage: Can cause unexpected behavior and bugs if the code structure is not well understood.

<u>Closure in JavaScript:</u>

A closure is when a function remembers and uses variables from the outer function, even after the outer function has finished running.

Example:

```
function outer() {
  let name = "Sam";

  function inner() {
    console.log(name); // inner still remembers 'name'
  }

  return inner;
}

const callFunc = outer();
callFunc(); // Output: Sam
```

Advantage: Enables data privacy and state persistence.

Disadvantage: Can lead to memory leaks if not managed properly.

Higher Order Functions:

A higher order function is a function that takes another function as an argument or returns a function.

Example:

```
function greet(name) {
  return "Hello " + name;
}

function higherOrder(func, value) {
  return func(value); // using function as an argument
}

console.log(higherOrder(greet, "Sam")); // Output: Hello Sam
```

Usage: Used to take functions as arguments or return them.

Advantage: Promotes code reuse and abstraction.

Disadvantage: May be harder to understand for beginners.

Map():

Used to create a new array by changing each element.

Example:

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

Advantage: Doesn't modify the original array and makes code cleaner.

Disadvantage: Not ideal if you're not returning a value for every element.

Filter():

Used to create a new array with only the items that match a condition.

Example:

```
const numbers = [1, 2, 3, 4];
const evens = numbers.filter(num => num % 2 === 0);
console.log(evens); // [2, 4]
```

Advantage: Easily extracts matching data from arrays.

Disadvantage: Can be inefficient on large datasets if not optimized.

Reduce():

Used to take all items and reduce them to a single value.

Example:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // 10
```

Advantage: Great for aggregating values like sums or totals.

Disadvantage: Can be hard to read and understand for beginners.

Note:

total: This is the running total (also called the accumulator).

num: This is the current number from the array.

0: This is the starting value for total.

forEach():

forEach() is used to run a function once for each item in an array.

Syntax:

[1, 2, 3].forEach(x => console.log(x));

Example:

```
const fruits = ["apple", "banana", "cherry"];

fruits.forEach(function(fruit) {
  console.log(fruit);
});

//Output:
apple
banana
cherry
```

Advantage: Simple syntax for looping through arrays.

Disadvantage: Doesn’t return a new array or break out early.

some():

some() checks if at least one item in the array passes a condition.

Example:

```
const numbers = [1, 2, 3, 4];
const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // true
```

Advantage: Stops checking after the first match, improving performance.

Disadvantage: Doesn’t return matching elements, only a boolean.

every():

every() checks if all items in the array pass a condition.

Example:

```
const numbers = [2, 4, 6];
const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // true
```

Advantage: Ensures array consistency with minimal code.

Disadvantage: Fails fast and gives only a boolean result.

Regex:

A regex is a pattern used to search, match, or replace text in strings.

Example:

```
const text = "I love JavaScript!";
const pattern = /love/;
console.log(pattern.test(text)); // true
```

Note: test() checks if the word "love" is in the text.

Advantage: Powerful tool for string validation and manipulation.

Disadvantage: Syntax is complex and hard to debug.

DOM:

The DOM (Document object Model) is a tree-like structure of your web page that allows JavaScript to read and manipulate the content, elements, and styles dynamically.

Methods:

Method	Description
`getElementById(id)`	Finds an element by its `id`.
`getElementsByClassName(class)`	Returns a list of elements with the given class name.
`getElementsByTagName(tag)`	Returns a list of elements with the given tag name (e.g., `div`, `p`).
`querySelector(selector)`	Returns the **first element** that matches the CSS selector.
`querySelectorAll(selector)`	Returns **all elements** that match the CSS selector.

Example:

```
document.getElementById("demo").innerText = "You clicked the button!";
```

Advantage: Allows JavaScript to interact with and modify HTML/CSS dynamically.

Disadvantage: Complex structure can lead to performance issues with large pages.

Event Propagation:

Event propagation is how an event moves through the elements in the DOM — it can either go from parent to child (capturing) or from child to parent (bubbling) when an event is triggered.

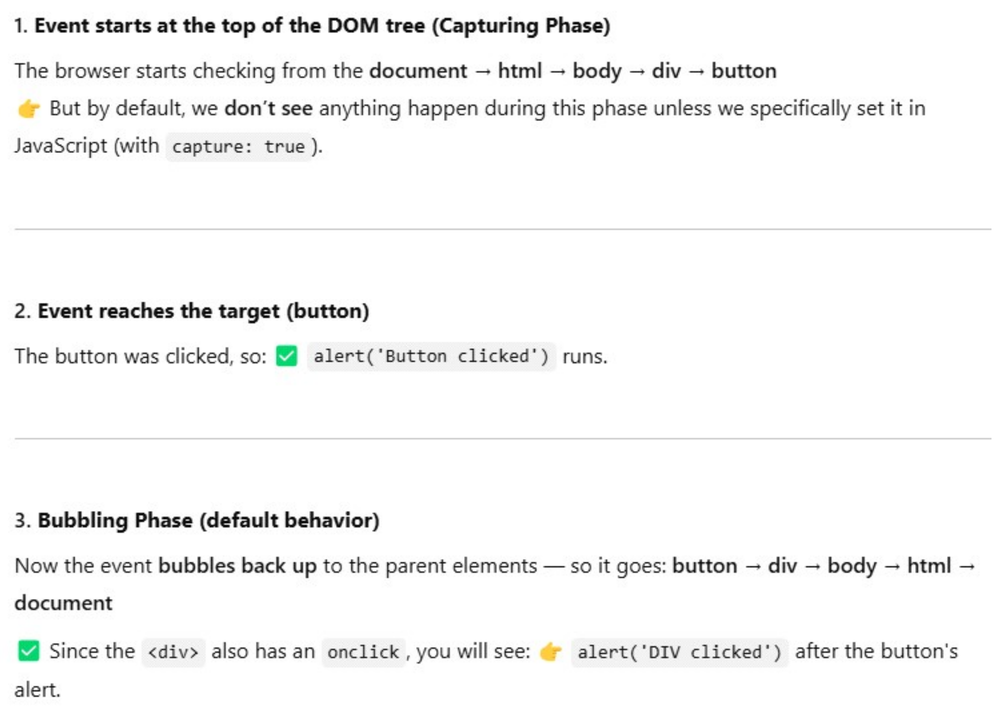
Two Main Phases:

- Capturing Phase – Event travels from the top (window) down to the target element.
- Bubbling Phase – Event bubbles back up from the target element to the top.

Example:

```html
<div onclick="alert('DIV clicked')">
  <button onclick="alert('Button clicked')">Click Me</button>
</div>
```

Explanation: What happens when you click the button?



1. **Event starts at the top of the DOM tree (Capturing Phase)**

The browser starts checking from the **document** → **html** → **body** → **div** → **button**
👉 But by default, we **don't see** anything happen during this phase unless we specifically set it in JavaScript (with `capture: true`).

---

2. **Event reaches the target (button)**

The button was clicked, so: ✅ `alert('Button clicked')` runs.

---

3. **Bubbling Phase (default behavior)**

Now the event **bubbles back up** to the parent elements — so it goes: **button** → **div** → **body** → **html** → **document**

✅ Since the `<div>` also has an `onclick`, you will see: 👉 `alert('DIV clicked')` after the button's alert.

Advantage: Gives control over how events are captured and handled (capture/bubble).

Disadvantage: Can lead to unexpected behavior if not properly managed.

Event Listeners:

Event Listeners are functions in JavaScript that wait for user actions (like clicks, key presses, or mouse movements) and respond when those actions happen.

Note:

addEventListener() is a method used to attach an event handler to an HTML element without overwriting existing events.

Syntax:

```
element.addEventListener("event", function, useCapture);
```

Example:

Html:

```
<button id="btn">Click Me</button>
```

Js:

```
document.getElementById("btn").addEventListener("click", function () {
  alert("Button clicked!");
});
```

Advantage: Enables dynamic interaction with web elements without altering HTML.

Disadvantage: Too many listeners can slow down the page or cause memory leaks.

Exception Handling:

Exception handling is a technique used to detect, catch, and handle errors during program execution without stopping the entire script.

Syntax:

```
try {
  // Code that may cause an error
} catch (error) {
  // Handle the error
} finally {
  // (Optional) Code that runs no matter what
}
```

Example:

```
try {
  let a = 10;
  console.log(a.toUpperCase()); // Error
} catch (err) {
  console.log("An error occurred:", err.message);
} finally {
  console.log("This runs no matter what.");
}
```

Advantage: Prevents app crashes by handling runtime errors gracefully.

Disadvantage: Overuse may hide bugs or make debugging harder.

Synchronous JavaScript :

Synchronous JavaScript executes line by line, one task at a time — it waits for each task to finish before moving to the next.

Example:

```
function syncFunc() {
  console.log("Start");
  console.log("Processing...");
  console.log("End");
}

syncFunc();

//Output:
Start
Processing...
End
```

Asynchronous JavaScript:

Asynchronous JavaScript allows multiple tasks to run in the background — it doesn't wait and continues running other code.

Example:

```
function asyncFunc() {
  console.log("Start");

  new Promise((resolve) => {
    setTimeout(() => {
      resolve("Processing...");
    }, 1000);
  }).then((message) => {
    console.log(message);
  });

  console.log("End");
}

asyncFunc();

//Output:
Start
End
Processing...  ← (after 1 second)
```

API Calling:

API calling using fetch or axios means sending a request from your JavaScript code to an external server (API) to get or send data, often used to load content dynamically in web apps.

- fetch is a built-in JavaScript function to make HTTP requests.
- axios is a popular third-party library that simplifies API calls and handles JSON and errors more easily.

i) Using fetch() — Native JavaScript method:

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => response.json())
  .then((data) => console.log("Fetch Result:", data))
  .catch((error) => console.error("Error:", error));
```

Explanation:

- fetch() sends a request to the URL.
- .then() handles the response and converts it to JSON.
- .catch() handles any errors.

ii) Using axios — External library (you need to install via npm or CDN):

```
axios.get("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => console.log("Axios Result:", response.data))
  .catch((error) => console.error("Error:", error));
```

Explanation:

- axios.get() sends a GET request.
- response.data gives you the actual data.
- Simpler and has automatic JSON parsing.

Optional Chaining (?.)

It safely accesses deeply nested properties without throwing an error if any part is null or undefined.

Example:

```
let user = { profile: { name: "Sam" } };

console.log(user.profile?.name); // "Sam"
console.log(user.address?.street); // undefined (no error)
```

Nullish Coalescing Operator (??)

It returns the right-hand value only if the left-hand value is null or undefined (not for 0, false, or "").

Example:

```
let name = null;
let userName = name ?? "Guest";

console.log(userName); // "Guest"
```