Day :15 JavaScript Notes.

Understanding JavaScript Variable Declarations

What Are Variables?

- **Variables are containers** used to store data values in memory.
- In JavaScript, variables can be declared using:
  - var (older way)
  - let (modern, preferred for mutable values)
  - const (modern, preferred for immutable references)

---

var

- **Used before ES6** (older standard).
- **Function-scoped** — accessible within the function it's declared in.
- Can be **re-declared** and **updated**.
- **Hoisted** — moved to the top of its scope, but initialized as undefined.

Example:

```
var x = 10;
var x = 20; // ✅ Allowed
```

---

let

- **Introduced in ES6 (2015)**.
- **Block-scoped** — limited to the {} block where it is declared.
- Can be **updated**, but **not re-declared** in the same scope.
- Also **hoisted**, but not initialized — accessing it before declaration causes an error.

Example:

```
let y = 5;
```

y = 15; // ✅ Allowed

let y = 20; // ❌ Error: Identifier 'y' has already been declared

---

const

- Also introduced in **ES6**.
- **Block-scoped**, like let.
- **Cannot be updated or re-declared**.
- Must be **initialized at the time of declaration**.
- You **can modify contents** of objects or arrays, but **cannot reassign** the variable.

Example:

const z = 100;

z = 200; // ❌ Error

const arr = [1, 2, 3];

arr.push(4); // ✅ Allowed — modifying contents

Type Conversions in JavaScript

JavaScript is loosely typed, meaning values can be **converted between types** automatically (implicit conversion) or manually (explicit conversion).

---

1. Converting to Number

You can convert other types to numbers using:

- Number(value)
- parseInt(value)
- parseFloat(value)
- Unary + operator

Examples:

```
console.log(Number(" "));       // 0
console.log(Number("99 88"));   // NaN (space makes it invalid)
console.log(Number("hads"));    // NaN


let x = "5";
let y = +x;
console.log(typeof y);          // "number"
```

parseFloat() vs parseInt()

- parseFloat("3.14") returns 3.14
- parseInt("100px") returns 100 (parses until non-digit)

---

2. Converting to String

You can convert other types to strings using:

- String(value)
- value.toString()

```
let num = 123;
console.log(String(num));       // "123"
console.log((true).toString()); // "true"
```

---

3. Converting Dates

JavaScript's Date object can be converted to strings and numbers:

```
let d = new Date();
console.log(d);             // Full date object
console.log(d.getDate());   // Day of the month (1–31)
console.log(d.toString());  // Date as readable string
```

JavaScript Numbers and BigInt

JavaScript Number (Standard Numeric Type)

In JavaScript, all regular numbers are stored using the **IEEE-754 double-precision floating-point format**. This means JavaScript can accurately represent integers up to **15 digits**. Beyond this, precision issues may occur, and numbers may be **rounded or lose accuracy**.

Example:

```
let n = 12345678901234568;
console.log(n);  // Output may be inaccurate due to precision limits
```

To ensure accurate handling of integers, JavaScript defines limits:

- **Maximum safe integer**: 2^53 - 1
- **Minimum safe integer**: -(2^53 - 1)

You can access them using:

```
console.log(Number.MAX_SAFE_INTEGER);  // 9007199254740991
console.log(Number.MIN_SAFE_INTEGER);  // -9007199254740991
```

JavaScript BigInt

BigInt is a special numeric type introduced to represent **very large integers** beyond the safe range of regular numbers.

Key Points:

- BigInt values are created by appending **n** to the end of the number, or by using the **BigInt()** constructor.
- BigInt supports arbitrary-length integers and maintains **full precision**, no matter how large the number is.
- **BigInt cannot be mixed** with regular numbers in arithmetic operations.

Examples:

```
let y = 999999999999999999999;        // May lose precision
let z = 999999999999999999999999999n;  // BigInt
```

```
let z1 = BigInt("11111111111111111111111111");
let y1 = BigInt(12336314143211234221532523453 46);


console.log(y);     // Inaccurate value
console.log(z, z1); // Accurate BigInt values
console.log(typeof z1); // Output: bigint
```

BigInt Arithmetic Rules

BigInt values must not be directly combined with regular numbers in operations. Doing so will throw a TypeError.

Invalid Example:

```
// let result = 5n / 2; // ❌ Error
```

To perform such operations, you must **convert BigInt to a Number** first:

```
let n3 = 5n;
let n4 = Number(n3) / 2;
console.log(n4);  // Output: 2.5
```

Note: Converting a BigInt to a Number may **lose precision** if the BigInt is very large.

Checking Numbers in JavaScript

JavaScript provides methods to verify number types and ranges:

- Number.isInteger(x) → Checks if x is an integer.
- Number.isSafeInteger(x) → Checks if x is an integer within the safe range.

```
console.log(Number.isInteger(12));     // true
console.log(Number.isInteger(12.5));   // false


console.log(Number.isSafeInteger(12)); // true
```

```
console.log(Number.isSafeInteger(11122222222222222222222222222222222222));
// false
```

Limitations of BigInt

- **BigInt is not compatible with JSON.stringify()**. Trying to stringify an object containing BigInt will throw a TypeError.

```
let big = BigInt(12345678912345678912345678);

// JSON.stringify(big); // ✕ Throws TypeError
```

- **BigInt does not support Number methods** such as .toFixed() or .toPrecision().

```
let big = 123456789n;

// console.log(big.toFixed(2)); // ✕ Error: toFixed is not a function
```

When to Use BigInt

Use BigInt in situations where you need to work with **very large whole numbers** that exceed JavaScript's safe integer limit. Common use cases include:

- Cryptographic computations
- High-precision financial systems
- Scientific simulations that deal with extremely large integers

JavaScript Hoisting, Strict Mode

---

1. JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving **declarations** to the **top of their scope** before code execution. This allows you to reference certain variables and functions **before they are declared**, but only in specific cases.

Key Rules:

- Only **declarations** are hoisted — **initializations are not**.
- Hoisting applies to:
  - var declarations (not values)
  - **Function declarations** (but not function expressions)
- It does **not apply** the same way to let and const.

Example (Error with let):

```
x = 5;
console.log(x); // ReferenceError
let x;
```

Here, although x is declared with let, it's in the **Temporal Dead Zone (TDZ)** — a phase where the variable exists but cannot be accessed until the actual declaration line is reached.

Example of TDZ:

```
try {
  carName = "Saab";
  let carName = "Volvo"; // Error: Cannot access before initialization
} catch(err) {
  console.log(err); // ReferenceError
}
```

---

2. "use strict" Directive

JavaScript's "use strict" is a **directive** that enables **strict mode**, which was introduced in **ECMAScript 5**. It helps developers write **cleaner, safer code** by turning previously silent errors into exceptions.

Global Strict Mode:

```
"use strict";
x = 3.14; // ReferenceError: x is not defined
```

Function-Level Strict Mode:

```
x = 3.14; // This will work normally
function myFunction() {
  "use strict";
  y = 3.14; // ReferenceError: y is not defined
}
```

Advantages of Strict Mode:

- Disallows usage of undeclared variables.
- Prevents accidental creation of global variables.
- Eliminates this coercion: this remains undefined in functions if not explicitly set.
- Disallows duplicate parameter names in functions.
- Prevents use of future JavaScript **reserved keywords** as variable names (e.g., implements, interface, let, package, etc.).

Strict mode is particularly useful in helping developers avoid common mistakes and write more secure and robust code.

JavaScript Bitwise Operators

Bitwise operators in JavaScript allow you to manipulate individual bits of binary numbers. They are mainly used in low-level programming, performance optimization, cryptography, and image processing.

Each number is internally represented in **binary (base-2)** form, and these operators work directly on those binary values.

List of Bitwise Operators

& (AND)

Compares each bit of two numbers and returns 1 **only if both bits are 1**.

```
5 & 1;  // 0101 & 0001 → 0001 → 1
```

| (OR)

Returns 1 if **at least one** of the bits is 1.

```
5 | 1;  // 0101 | 0001 → 0101 → 5
```

^ (XOR – Exclusive OR)

Returns 1 if **only one** of the bits is 1, not both.

```
5 ^ 1;  // 0101 ^ 0001 → 0100 → 4
```

~ (NOT – Bitwise Inversion)

Inverts every bit. In JavaScript, ~n returns -(n + 1) due to the way negative numbers are stored in two's complement format.

```
~5;     // ~00000000 00000000 00000000 00000101 → 11111111 11111111
11111111 11111010
        // Result: -6
```

<< (Left Shift)

Shifts bits to the left by the specified number of positions. Zeros are filled in from the right.

```
5 << 1;  // 0101 << 1 → 1010 → 10
```

>> (Right Shift)

Shifts bits to the right. The leftmost bit is filled based on the sign (0 for positive, 1 for negative).

```
5 >> 1;  // 0101 >> 1 → 0010 → 2
```

>>> (Unsigned Right Shift)

Like >>, but fills 0 from the left **regardless of the sign**. Always returns a non-negative result.

```
5 >>> 1;  // 0101 >>> 1 → 0010 → 2
```

Binary and Decimal Conversion

You can convert between decimal and binary using built-in methods in JavaScript:

Convert Decimal to Binary

Use (number >>> 0).toString(2) to convert a decimal to a binary string.

```
function decToBin(dec) {
  return (dec >>> 0).toString(2);
}
console.log(decToBin(5));  // Output: "101"
```

Convert Binary to Decimal

Use parseInt(binaryString, 2) to convert a binary string to a decimal number.

```
function binToDec(bin) {
  return parseInt(bin, 2).toString(10);
}
console.log(binToDec("101"));  // Output: "5"
```

When to Use Bitwise Operators

- Working with permissions and flags (e.g., bit masks)
- Performance-critical math or logic
- Compression and encryption algorithms
- Manipulating binary data formats or hardware-level values

JavaScript Regular Expressions (Regex)

Regular expressions are powerful tools used for **pattern matching** within strings. They are most commonly used for **searching**, **matching**, and **replacing** parts of text based on specific patterns.

In JavaScript, regular expressions are often used with string methods such as:

- .search()
- .match()
- .replace()

Basic Syntax

Regular expressions can be written between forward slashes /pattern/ or by using the RegExp() constructor.

Example:

```
let text = "Visit W3Schools!";
let index = text.search(/w3schools/i);  // Output: 6
```

Here, the pattern /w3schools/i searches for "w3schools" regardless of case (because of the i modifier).

Using .replace() with Regex

The .replace() method can substitute matched patterns with new content.

Example:

```
let text2 = "Visit Microsoft!";
let result = text2.replace(/microsoft/i, "W3Schools");
console.log(result);  // Output: "Visit W3Schools!"
```

Regex Modifiers

Modifiers change the behavior of the pattern matching:

- **i** – Case-insensitive matching
- **g** – Global matching (finds all matches, not just the first)

Example:

```
let sentence = "Is this all there is?";
let result = sentence.match(/is/g);  // Output: ['is', 'is']
```

Common Regex Patterns

Here are some useful patterns and what they do:

- [abc] – Matches any one character from a, b, or c
- [0-9] – Matches any single digit from 0 to 9
- (x|y) – Matches either x or y
- \d – Matches any digit (equivalent to [0-9])
- \s – Matches any whitespace (space, tab, newline)

Examples:

```
"123456789".match(/[1-4]/g);     // Output: ['1', '2', '3', '4']
"Give 100%!".match(/\d/g);       // Output: ['1', '0', '0']
```

Quantifiers in Regex

Quantifiers allow you to match patterns based on how many times a character or group appears:

- n+ – Matches **one or more** occurrences of n
- n* – Matches **zero or more** occurrences of n
- n? – Matches **zero or one** occurrence of n

Examples:

```
"Hellooo".match(/o+/g);        // Output: ['ooo']
"1, 100 or 1000?".match(/10?/g); // Output: ['1', '10', '10']
```

Best Practices and Notes

- Always use "use strict"; at the top of your JavaScript files to enforce better coding practices, including mandatory variable declarations.
- Avoid using undeclared variables — prefer let and const which are **block-scoped**.
- Regular expressions are very useful for:
    - **Form input validation**
    - **Data extraction**
    - **String parsing and transformation**
- When using dynamic or complex regex patterns, wrap your code in try...catch blocks to safely handle possible errors:

```
try {
  let regex = new RegExp(userInputPattern);
  // test or match something
} catch (error) {
  console.error("Invalid regular expression:", error);
}
```

---

3. JavaScript Modules

JavaScript **modules** allow you to split your code across multiple files, improving **maintainability**, **reusability**, and **readability**. Modules help organize related logic into separate files and prevent variable conflicts by keeping scope isolated.

Using Modules in HTML

To use JavaScript modules in the browser, you must set the <script> tag's type to "module":

```
<script type="module">
```

```
  import message from "./message.js";
</script>
```

This tells the browser to treat the script as a module, which enables features like import and export.

Exporting from a Module

You can **export** values from a JavaScript file to be used in another. There are two main types of exports:

Named Exports

Named exports allow you to export multiple variables or functions. You can do it either inline or at the bottom of the file.

Inline Named Export:

```
// person.js
export const name = "Jesse";
export const age = 40;
```

Grouped Export at the Bottom:

```
// person.js
const name = "Jesse";
const age = 40;

export { name, age };
```

You can import these by using the exact variable names inside curly braces:

```
import { name, age } from "./person.js";
console.log(`My name is ${name}, I am ${age}.`);
```

Default Export

A module can have **only one default export**, which is useful when the module is exporting a single main functionality.

```
// message.js
const message = () => {
  const name = "Jesse";
  const age = 40;
  return `${name} is ${age} years old.`;
};

export default message;
```

To import a default export, you can name it anything:

```
import message from "./message.js";
console.log(message());
```

Important Notes on Modules

- You **cannot mix named and default exports in a single import statement** unless you use the proper syntax:
- `import message, { name, age } from "./module.js";`
- Scripts loaded using type="module" are **deferred by default**, meaning they wait until the HTML is fully parsed before executing — just like using defer in regular scripts.
- Every module has its **own scope**, so variables declared in one module will not interfere with those in another, even if they use the same names.

Why Use Modules?

- Cleaner code organization
- Easier debugging and maintenance
- Better scalability in large projects

- Prevents global namespace pollution
- Encourages code reuse through shared logic

JavaScript Iterables, Sets, and Data Types

JavaScript Iterables

In JavaScript, an **iterable** is any object that can be looped over using the for...of loop. Common built-in iterables include **Strings**, **Arrays**, **Sets**, and **Maps**.

Iterating Over a String

A string is a sequence of characters and is iterable character-by-character:

```
let str = "hello";
let text = "";
for (let x of str) {
  text += x;
}
console.log(text); // Output: "hello"
```

Iterating Over an Array

Arrays are ordered collections of items. Using for...of makes it easy to loop through all elements:

```
let ary = [1, 23, 45];
let text1 = "";
for (let x of ary) {
  text1 += x;
}
console.log(text1); // Output: "12345"
```

JavaScript Sets

A **Set** is a built-in object that stores **unique values** — no duplicates allowed. A Set can hold both primitive types and object references.

Creating and Using a Set

```
let num1 = new Set(['a', 'b', 'c', 'a']); // Duplicate 'a' is ignored
num1.add(23);
console.log(num1); // Output: Set(4) { 'a', 'b', 'c', 23 }
```

Common Set Methods

- add(value): Adds a new value
- has(value): Checks if a value exists
- delete(value): Removes a value
- clear(): Removes all values
- size: Returns the number of unique elements

```
console.log(num1.has('a'));  // true
console.log(num1.size);      // 4
```

Iterating Over a Set Using for...of

```
let letters = new Set(['a', 'b', 'c']);
let result = "";
for (let letter of letters) {
  result += letter;
}
console.log(result); // Output: "abc"
```

Iterating Over a Set Using forEach()

You can also iterate using forEach() just like with arrays:

```
let values = new Set(['a', 'b']);
let output = "";
values.forEach(function (item) {
  output += item;
```

```
});
console.log(output); // Output: "ab"
```

JavaScript Maps

A **Map** in JavaScript is a collection of **key-value pairs**, similar to objects, but with some important differences. In a Map:

- Keys can be of **any data type** (not just strings).
- The **order of entries** is preserved (insertion order is remembered).
- Maps provide **better performance** for frequent additions/removals compared to regular objects.

Creating and Modifying a Map

You can create a Map by passing an array of key-value pairs. You can also add or update entries using the .set() method:

```
let fruits = new Map([
  ["mango", 800],
  ["bananas", 400],
  ["apple", 300]
]);

fruits.set("kiwi", 150);       // Adds new entry
fruits.set("bananas", 600);    // Updates existing entry
```

Useful Map Methods

- .get(key): Retrieves the value associated with the key.
- .set(key, value): Adds or updates the entry.
- .delete(key): Removes the entry with the specified key.

- .has(key): Returns true if the key exists.
- .size: Returns the number of entries in the Map.

```
console.log(fruits.get("apple"));     // 300
console.log(fruits.has("apple"));     // true
fruits.delete("bananas");             // Removes "bananas"
console.log(fruits.size);             // Current size of the Map
```

Iterating Through a Map

Maps can be iterated using both forEach() and for...of loops.

Using forEach()

The forEach() method takes a callback function with parameters (value, key):

```
let result = "";
fruits.forEach(function(value, key) {
  result += `${key}: ${value} `;
});
console.log(result); // Example: "mango: 800 apple: 300 kiwi: 150 "
```

Using for...of with keys(), values(), and entries()

You can loop over just the keys, just the values, or both:

```
// Keys
for (let key of fruits.keys()) {
  console.log(key); // mango, apple, kiwi
}

// Values
for (let value of fruits.values()) {
  console.log(value); // 800, 300, 150
```

```
}

// Entries (both key and value)
for (let [key, value] of fruits.entries()) {
  console.log(`${key}: ${value}`);
}
```

JavaScript Math Object (with Examples)

JavaScript provides a built-in Math object that contains properties and methods for performing mathematical operations. It's a static object, meaning you don't create a Math instance—just use it directly. This object is especially useful for handling rounding, trigonometry, random number generation, logarithmic calculations, and more.

Mathematical Constants

The Math object includes several useful constants:

- Math.PI returns the value of π (approximately 3.14159).
  Example: console.log(Math.PI); // 3.141592653589793
- Math.E returns Euler's number (≈ 2.71828).
  Example: console.log(Math.E);
- Math.LN10 and Math.LN2 return the natural log of 10 and 2.
  Example: console.log(Math.LN10);
- Math.LOG10E and Math.LOG2E give the base 10 and base 2 logarithms of Euler's number.
  Example: console.log(Math.LOG10E);
- Math.SQRT1_2 returns the square root of ½ and Math.SQRT2 returns the square root of 2.
  Example: console.log(Math.SQRT2);

Rounding Methods

JavaScript provides multiple methods to handle rounding:

- Math.round(4.5) rounds to the nearest whole number → 5.
- Math.ceil(4.1) rounds **up** to 5.
- Math.floor(4.9) rounds **down** to 4.
- Math.trunc(4.7) removes the decimal and returns 4.
  Example:
- let x = 4.5;
- console.log(Math.round(x)); // 5
- console.log(Math.ceil(x)); // 5
- console.log(Math.floor(x)); // 4
- console.log(Math.trunc(x)); // 4

Sign Function

The Math.sign() function tells whether a number is negative, positive, or zero:

- Math.sign(-4) → -1
- Math.sign(0) → 0
- Math.sign(4) → 1
  Example: console.log(Math.sign(-25));

Power and Roots

- Math.pow(4, 3) computes 4 to the power of 3 → 64.
- Math.sqrt(81) returns the square root → 9.
  Example:
- console.log(Math.pow(4, 3)); // 64
- console.log(Math.sqrt(81)); // 9

Absolute Value

- Math.abs(-4.2) returns 4.2, converting negative to positive.
  Example: console.log(Math.abs(-7.9));

Trigonometric Functions

Trigonometric functions in JavaScript use **radians**, not degrees. Convert degrees to radians using (degrees * Math.PI / 180).

- Math.sin(90 * Math.PI / 180) returns 1.
- Math.cos(0 * Math.PI / 180) returns 1.
  Example:
- console.log(Math.sin(90 * Math.PI / 180)); // 1
- console.log(Math.cos(0 * Math.PI / 180)); // 1

Min, Max, and Random Values

- Math.min(2, 3, 67, 7, 9) returns the smallest → 2.
- Math.max(1, 3, 67, 1, 0) returns the largest → 67.

To generate random numbers:

- Math.random() returns a decimal between 0 (inclusive) and 1 (exclusive).
- Math.floor(Math.random() * 10) + 1 gives a random number from 1 to 10.
  Example:
- console.log(Math.random()); // Random between 0–1
- console.log(Math.floor(Math.random() * 10) + 1); // 1–10

For **custom ranges**, use:

```
// Random number between min (inclusive) and max (exclusive)
function randomBetween(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

```
// Random number between min and max (inclusive)
function randomInclusive(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Always ensure min < max before calling these functions.

Logarithmic Functions

- Math.log(x) returns the natural logarithm (base e).
  Example: `console.log(Math.log(3));`
- Math.log10(x) returns the base-10 logarithm.
  Example: `console.log(Math.log10(100)); // 2`

Best Practices & Notes

- Math is not a constructor—do not use new Math().
- Math.random() is **not cryptographically secure**; for secure random values, use the Web Crypto API.
- Always round or truncate floating-point numbers to avoid precision issues.
- Convert angles to radians when using trigonometric functions.
