Week 7 - Javascript

Variable

JavaScript provides **three ways** to declare variables: `var`, `let`, and `const`.

Each behaves differently in terms of scope, hoisting, and reassignment.

1. `var` — Function-Scoped / Globally-Scoped

- `var` is **function-scoped** (or global if declared outside a function).
- Ignores block scope (`{}`).
- Can be **redeclared** and **reassigned**.
- Gets hoisted with default value `undefined`.

Example:

```
var name = "Alice";
console.log("Using var:", name); // alice

{
  var name = "Bob"; // Overwrites outer variable
  console.log("Using var inside block:", name); // bob
}

console.log("Using var outside block after reassignment:", name); // bob
```

2. `let` — Block-Scoped

- `let` is **block-scoped** (`{}`).
- Cannot be redeclared in the same scope.
- Can be **reassigned**.
- Hoisted but not initialized (Temporal Dead Zone).

Example:

```
let age = 25;
console.log("Using let:", age); // 25

{
  let age = 30; // Different variable (block-scoped)
```

```
  console.log("Using let inside block:", age); // 30
}

console.log("Using let outside block:", age); // 25
```

3. `const` — Block-Scoped & Read-Only

- Block-scoped like `let`.
- **Must be initialized** during declaration.
- **Cannot be reassigned**.
- The value inside a const object/array can still be mutable.

Example:

```
const country = "USA";
console.log("Using const:", country);
```

Hoisting & Global Scope

Hoisting of Variables

JavaScript “hoists” variable declarations, but **not** their initial values.

`var` Hoisting

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

- `var a` is hoisted to the top of the scope.
- Its value is set to `undefined` until the assignment happens.
- `var` becomes a **property of the global object** (`window.a` in browsers).

`let` and `const` Hoisting

```
// console.log(b); // ReferenceError: Cannot access 'b' before initialization
let b = 20;
```

- The declaration is hoisted but not initialized.
- They stay in the **Temporal Dead Zone (TDZ)** until execution reaches their line.
- Accessing them early results in **ReferenceError**, not `undefined`.

Undeclared Variables

```
c = 30; // Implicit global variable
console.log(c); // Accessible globally
```

- Assigning to a variable without `var`, `let`, or `const` creates a **global variable**.
- This applies even inside functions.

  Not recommended — leads to accidental globals.

Hoisting of Functions

JavaScript treats two kinds of functions differently:

Function Declaration

```
fun1(); // Works

function fun1() {
  console.log("Inside fun1");
}
```

- Fully hoisted with function body.
- Can be called **before** the declaration.

Function Expression

```
// fun2(); // ReferenceError: Cannot access 'fun2' before initialization

fun2 = function () {
  console.log("Inside fun2");
};
```

- Behaves like a variable assignment.
- Not hoisted with its function body.

Data Types

JavaScript has two main categories of data types:

- **Primitive (stack memory)**
- **Non-primitive / Reference types (heap memory)**

1. Primitive Data Types

Primitive values are **immutable** and stored directly in memory (stack).

1. Number

```
let a = 10;
let b = 10.2;
```

2. String

```
let c = "abc";
let d = "abc";
let e = `abc${a}`; // string interpolation
```

3. Boolean

```
let f = false;
```

4. Undefined

Declared but not assigned any value.

```
let g;
```

5. Null

Intentional empty value.

```
let h = null;
```

6. BigInt

Used for very large integers.

```
let i = 9007199254740991n;
```

7. Symbol

Unique and immutable identifier.

```
let j = Symbol("id");
```

2. Non-Primitive (Reference) Data Types

Stored in heap memory; variables hold a **reference**.

Object

```
let k = {
  name: "dakshil",
  college: "bvm",
};
```

Array

```
let l = [1, 2, 3];
```

Function

```
let m = function () {
  console.log("FUNCTION");
};
```

Memory Allocation Difference

Primitive — values are copied

```
let name = "a";
let anotherName = name;

anotherName = "b";

console.log(name); // a
console.log(anotherName); // b
```

Changing one doesn't affect the other.

Reference Types — reference (address) is copied

```
let ob = { name: "a" };
let anotherOb = ob;

anotherOb.name = "b";

console.log(ob);        // { name: "b" }
console.log(anotherOb); // { name: "b" }
```

Both variables point to the **same object** in memory.

Type Conversion

String → Number

```
let s = "5";
let num = Number(s); // 5
```

Undefined → Number

```
let a = undefined;
num = Number(a); // NaN
```

Type of NaN

```
console.log(typeof NaN); // "number"
```

Other Conversion

Value	Number() Output
`"123"`	123
`""`	0
`"abc"`	NaN
`null`	0
`undefined`	NaN
`true`	1
`false`	0

Boolean Conversion

```
let boolean = Boolean("");
console.log(boolean); // false
```

Falsy Values in JS

- false
- undefined
- null
- 0
- NaN
- "" (empty string)

String + Number Behavior

JavaScript evaluates left to right.

```
console.log("1" + 1 + 1); // "111"
// string → string → string

console.log(1 + 1 + "1"); // "21"
// number + number = 2 → "2" + "1" = "21"
```

Comparison: `==` vs `===`

`==` → loose comparison (type conversion allowed)

`===` → strict comparison (no type conversion)

```
console.log(2 == "2");  // true
```

```
console.log(2 === "2"); // false
```

Control Flow

`switch` statement

Use `switch` to branch on discrete values. Don't forget `break` to avoid fall-through.

```
const month = 3;

switch (month) {
  case 1:
    console.log("January");
    break;
  case 2:
    console.log("February");
    break;
  case 3:
    console.log("March");
    break;
  case 4:
    console.log("April");
    break;
  default:
    console.log("Default");
    break;
}
```

Single-line `if` and the comma

If you write an `if` without braces, **only the first statement** after it is controlled by the `if`.

```
const balance = 1000;

// WARNING: only the first statement is conditional
if (balance > 500) console.log("test"), console.log("test2");
// parsed as:
// if (balance > 500) console.log("test");
// console.log("test2");  // runs regardless of condition
```

Always use braces for clarity:

```
if (balance > 500) {
  console.log("test");
```

```
  console.log("test2");
}
```

Truthy / Falsy values

Falsy values (evaluate to `false` in boolean context):

- `false`
- `0`
- `0n` (BigInt zero)
- `""` (empty string)
- `null`
- `undefined`
- `NaN`

Truthy examples:

- `"0"`, `"false"`, `" "` (non-empty strings)
- `[]` (empty array)
- `{}` (empty object)
- `function(){}` (functions)

> Anything non-empty or non-zero is generally truthy.

Check empty array / object

```
const userEmail = [];
if (userEmail.length === 0) {
  console.log("Array is empty");
}

const emptyObj = {};
if (Object.keys(emptyObj).length === 0) {
  console.log("Object is empty");
}
```

Nullish coalescing operator `??`

`??` returns the first operand that is **not** `null` or `undefined`.

```
let val1;

val1 = 5 ?? 10;         // 5
val1 = null ?? 10;      // 10
```

```
val1 = undefined ?? 15; // 15

val1 = null ?? 10 ?? 20;        // 10
val1 = null ?? undefined ?? 11; // 11
```

Use `??` when you want to treat `0`, `""`, or `false` as valid values (unlike `||` which treats them as falsy).

Logical OR Operator `||`

`||` returns the first non `falsy` value

```
val1 = 5 || 10; // 5
val1 = 0 || 10; // 10
val1 = "" || 15; // 15
val1 = null || 10 || 20; // 10
val1 = undefined || 0 || 11; // 11
```

Ternary operator

Short inline `if`:

```
// condition ? valueIfTrue : valueIfFalse
const iceTeaPrice = 100;
iceTeaPrice <= 80 ? console.log("Less than 80") : console.log("More than 80");
```

Loop

Array Iteration

1) Classic `for` loop

Useful when you need full control over index and step.

```
for (let i = 0; i < arr.length; i++) {
  console.log(`Index: ${i}, Value: ${arr[i]}`);
}
```

2) `for...of` loop

Best for iterating **values** of arrays.

```
for (let value of arr) {
  console.log(value);
}
```

- Gives **values**

- Cleaner than the classic loop

3) `for...in` **loop** (Not ideal for arrays)

Iterates **keys (indexes)**.

```
for (let index in arr) {
  console.log(arr[index]);
}
```

4) `forEach()`

Clean and functional-style iteration.

```
arr.forEach((value, index) => {
  console.log(`Index: ${index}, Value: ${value}`);
});
```

Object Iteration

1) `for...in` **loop**

Iterates over **keys**.

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

2) Using `Object.keys()` **+** `forEach()`

Gives an array of keys.

```
Object.keys(obj).forEach((key) => {
  console.log(`Key: ${key}, Value: ${obj[key]}`);
});
```

3) Using `Object.entries()` **+** `for...of`

Gives `[key, value]` pairs.

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

While & Do...While Loops

While Loop

Runs when condition is **true**.

```
while (count < 5) {
  console.log(count);
```

```
  count++;
}
```

Do...While Loop

Executes **at least once**, even if condition is false initially.

```
let num = 0;
do {
  console.log(num);
  num++;
} while (num < 5);
```

`break` and `continue`

`continue` → Skip current iteration

`break` → Exit loop immediately

```
for (let i = 0; i < 10; i++) {
  if (i % 2 === 0) continue; // skip even numbers

  console.log(`Odd Number: ${i}`);

  if (i === 7) break; // stop loop
}
```

Objects

Using Symbols as Object Keys

Symbols create **unique, non-enumerable** keys.

```
const mySyn = Symbol("key1");

const user = {
  name: "dakshil",
  "full name": "gorasiya dakshil r",
  [mySyn]: "mykey1",  // symbol key (needs brackets)
};
```

Why brackets?

- `mySyn: "mykey1"` would create a **string key** `"mySyn"`
- `[mySyn]: "mykey1"` correctly uses the **Symbol** itself as the key

Accessing Object Properties

```
console.log(user.name);
console.log(user["full name"]);
console.log(user[mySyn]); // using symbol
console.log(user.lastLoginDays[0]);
```

Three access methods:

- Dot notation → `user.name`
- Bracket notation for multi-word keys → `user["full name"]`
- Bracket notation for symbol keys → `user[mySyn]`

Object.freeze() — Make an Object Immutable

```
Object.freeze(user);
user.name = "abc"; // will NOT work
```

- After freezing, no properties can be **added, changed, or deleted**.

Adding Methods to Objects

```
user.greeting = function () {
  console.log("hello user");
};
```

Creating Object Using `new Object()`

```
const ob = new Object();
ob.fun = function () {
  console.log("hello from ob");
};
ob.fun();
```

- Same as `{}`, but more explicit.

Merging Objects

1) Using `Object.assign()`

```
const obj3 = Object.assign({}, obj1, obj2);
```

2) Using Spread Operator `...`

```
const obj4 = { ...obj1, ...obj2 };
```

- Cleaner, modern syntax.

Object Utility Methods

```
Object.keys(obj3);    // ['1', '2', '3', '4']
Object.values(obj3);  // ['a', 'b', 'a', 'b']
Object.entries(obj3); // [['1','a'],['2','b'],...]
obj3.hasOwnProperty("1"); // true
```

- `keys` → returns array of property names
- `values` → returns array of values
- `entries` → returns key-value pairs
- `hasOwnProperty` → checks if a key exists

Object Destructuring

Extract values directly into variables.

```
const obj1 = {
  1: "a",
  2: "b",
};

const { 1: firstProp, 2: secondProp } = obj1;

console.log(firstProp);   // a
console.log(secondProp);  // b
```

- Works even with number-like keys.
- Syntax: `{ keyName: newVariableName }`

Functions

Function Hoisting

JavaScript hoists function declarations **fully**, but not function expressions or arrow functions.

```
console.log(add(2, 3)); // ✅ Works
// console.log(sub(5, 2)); // ❌ Error
// console.log(mul(2, 3)); // ❌ Error
```

Three Ways to Declare Functions

1) Function Declaration

Fully hoisted.

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

2) Function Expression

Stored in a variable.

```
let sub = function (a, b) {
  return a - b;
};
```

Not callable before definition.

3) Arrow Function

Shorter syntax.

```
let mul = (a, b) => {
  return a * b;
};
```

IIFE — Immediately Invoked Function Expression

Runs as soon as it is created.

```
(function () {
  console.log("IIFE function executed");
})();
```

IIFEs are used to:

- Avoid polluting global namespace
- Run setup code immediately

Scope & Closures

A closure is created when an inner function remembers variables from its outer function even after the outer function has returned.

```
function outer() {
  let outerVar = "I am from outer function";

  function inner() {
    console.log(outerVar); // outerVar is still accessible
  }

  return inner;
}
```

```
let innerFunc = outer();
innerFunc(); // "I am from outer function"
```

Closure Key Points:

- Inner functions can access outer function variables
- Those variables stay “alive” even after outer function execution
- Useful for data privacy, function factories, and more

DOM Manipulation

Selecting Elements

`querySelector()`

Returns the **first element** that matches a CSS selector.

```
h1tag = document.querySelector(".box h1");
```

`getElementById()`

Selects an element using its **id**.

```
button = document.getElementById("myButton");
```

`getElementsByClassName()`

Returns an **HTMLCollection** (array-like) of elements.

```
optionList = document.getElementsByClassName("optionsList")[0];
```

Changing Text Content

`innerText`

Sets/gets visible text only.

```
mainContent = document.getElementById("mainContent");
mainContent.innerText = "Lorem ipsum dolor sit amet...";
```

`innerHTML`

Replaces HTML inside the element.

```
mainContent.innerHTML = "<strong>Lorem ipsum dolor sit amet</strong>...";
```

> ⚠️ Use innerHTML carefully to avoid XSS.

Changing CSS with JavaScript

You can modify styles directly through `.style`:

```
button.style.padding = "10px 20px";
button.style.fontSize = "16px";
button.style.cursor = "pointer";
button.style.borderRadius = "5px";
```

Adding Event Listeners

Use `.addEventListener()` to react to user actions:

```
button.addEventListener("click", function () {
  console.log("Button was clicked!");
});
```

Advantages:

- Multiple listeners allowed
- Keeps HTML clean

Creating New Elements

`createElement()`

Creates a new element dynamically.

```
newLi = document.createElement("li");
newLi.innerText = "Option 4";
```

Appending a Child

```
optionList.appendChild(newLi);
```

Adds the new `<li>` at the end of the `ul`.

Removing Elements

Remove a specific child:

```
optionList.removeChild(optionList.children[1]); // Removes second item
```

Creating & Adding a Link

```
googleLink = document.createElement("a");
googleLink.setAttribute("href", "https://www.google.com");
googleLink.setAttribute("target", "_blank");
googleLink.innerText = "Go to Google";
```

```
document.body.appendChild(googleLink);
```

Key notes:

- `setAttribute()` is used to set attributes like `href`, `target`, `id`, etc.
- `_blank` opens the link in a new tab.

Events

Event Bubbling

- Default behavior of events in JavaScript.
- Event flow:

  Child → Parent → Document
- When an event occurs on a child element, it automatically propagates to its parent elements.
- Commonly used in event delegation.

Example Flow:

Button → Div → Body → Document

Event Capturing

- Opposite of event bubbling.
- Event flow:

  Parent → Child
- Must be enabled manually using `{ capture: true }`.

Order of execution

If `div>button` and both have both bubble and capture listeners then order of execution will be

1. div capture
2. button capture
3. button bubble
4. div bubble

`event.target` vs `event.currentTarget`

`event.target`

- Refers to the **actual element that triggered the event**.
- Changes depending on where the user clicks.

`event.currentTarget`

- Refers to the **element on which the event listener is attached**.
- Always remains constant for that handler.

Event Delegation

- Technique where a **single parent element** handles events of multiple child elements.
- Uses the concept of **event bubbling**.

Instead of:

- Adding many event listeners to child elements

We add One listener to the parent

Advantages:

- Better performance
- Less memory usage
- Works with dynamically created elements

`addEventListener()`

- Modern method for attaching events.
- Allows:
  - Multiple event handlers on the same element
  - Capturing or bubbling control
  - Better separation of logic and UI

Syntax supports:

- Bubbling (default)
- Capturing (`{ capture: true }`)

`stopPropagation()`

It prevents the event from reaching parent elements but allows other listeners on the same element to execute.

`stopImmediatePropagation()`

It stops the event completely — no parent handlers and no remaining handlers on the same element will run.

Important events

Mouse events

Event	When it Fires
`click`	Mouse click
`dblclick`	Double click
`mouseover`	Mouse enters element
`mouseout`	Mouse leaves element
`mousemove`	Mouse moves
`mousedown`	Mouse button pressed
`mouseup`	Mouse button released

Keyboard events

Event	When it Fires
`keydown`	Key is pressed
`keyup`	Key is released
`keypress`	Key is pressed (old, less used)

Form events

Event	When it Fires
`submit`	Form is submitted
`reset`	Form is reset
`change`	Value is changed
`input`	User types input
`focus`	Element gets focus
`blur`	Element loses focus

Window events

Event	When it Fires
`load`	Page fully loaded
`unload`	Page is closed
`resize`	Window size changes
`scroll`	Page is scrolled

Media events

Event	When it Fires
`play`	Media starts
`pause`	Media pauses
`ended`	Media ends
`volumechange`	Volume changes