JavaScript Code — VS Code Dark Theme Export

```
# JavaScript Complete Course - Personal Documentation & Notes

## ■ Course Structure & Table of Contents

### Part 1: Basics & Fundamentals
1. [Comments & Data Types](#1-comments--data-types)
2. [Variables (var, let, const)](#2-variables-declaration--assignment)
3. [Operators & Math](#3-operators--math-operations)
4. [Strings & Manipulation](#4-strings--manipulation)
5. [Arrays Basics](#5-arrays-basics)
6. [Array Methods](#6-array-methods)

### Part 2: Functions & Scope
7. [Functions - Reusable Code](#7-functions---reusable-code)
8. [Scope - Global vs Local](#8-scope---global-vs-local)
9. [Return Values](#9-return-values)
10. [Understanding Undefined](#10-understanding-undefined)

### Part 3: Control Flow & Logic
11. [Boolean Values & Conditions](#11-boolean-values--conditions)
12. [Comparison Operators](#12-comparison-operators)
13. [If/Else Statements](#13-ifelse-statements)
14. [Switch Statements](#14-switch-statements)
15. [Ternary Operator](#15-ternary-operator)

### Part 4: Objects & Data Structures
16. [Building JavaScript Objects](#16-building-javascript-objects)
17. [Accessing Object Properties](#17-accessing-object-properties)
18. [Nested Objects & Arrays](#18-nested-objects--arrays)
19. [Record Collection Project](#19-record-collection-project)

### Part 5: Loops & Iteration
20. [While Loops](#20-while-loops)
21. [For Loops](#21-for-loops)
22. [Iterating Through Arrays](#22-iterating-through-arrays)
23. [Nesting For Loops](#23-nesting-for-loops)
24. [Do-While Loops](#24-do-while-loops)
25. [Profile Lookup Example](#25-profile-lookup-example)

### Part 6: Utility Functions
26. [Random Numbers](#26-random-numbers)
27. [parseInt Function](#27-parseint-function)

### Part 7: Modern JavaScript (ES6+)
28. [Var vs Let - Scope Differences](#28-var-vs-let---scope-differences)
29. [Const - Read-Only Variables](#29-const---read-only-variables)
30. [Object.freeze()](#30-objectfreeze)
31. [Arrow Functions](#31-arrow-functions)
32. [Spread Operator](#32-spread-operator)
33. [Destructuring Assignment](#33-destructuring-assignment)
34. [Destructuring with Nested Objects](#34-destructuring-with-nested-objects)
35. [Template Literals](#35-template-literals)
36. [Concise Object Literals](#36-concise-object-literals)
37. [Classes & Constructors](#37-classes--constructors)
38. [Getters and Setters](#38-getters-and-setters)
39. [Modules - Import vs Require](#39-modules---import-vs-require)
40. [Export - Reuse Code Blocks](#40-export---reuse-code-blocks)
41. [Import Everything (*)](#41-import-everything-)
42. [Default Export & Fallback](#42-default-export--fallback)

---

## Part 1: Basics & Fundamentals

### 1. Comments & Data Types

```javascript
// *- to add a comment in java script i need to use this a symbols like this up and down
// *- data types used in java script :
// *undifined - null - bolean ( true or false ) - string  - symbol - number  and object
```
```

---

2. Variables: Declaration & Assignment

```javascript
var myName = "Abdellah " // var is used to the whole programme
//* myName = 23
//* let myName = "Abdellah"
const pi = 3.14 // the const cant be changed if we try to change const we will have an errorr

//* the difference between  declaring variables and  assigning variables :
var a ; // declring a variable
var b = 23 ; // assign a variable and and add  semicolon
console.log(a)
a = 7 ; //assign 7 to a and  a is already declared
b = a  ; //assign the content of a to b
console.log(a) // i download  node js Node.js lets you run JavaScript code directly on your computer instead of
```

---

3. Operators & Math Operations

```javascript
var c ; // declaration
var c = " I'm a " ;
c = c + "string !"; // assignment
console.log(c)

var sum = 23 + 2 ;
var diffrence = 23 - 2 ;
console.log(sum)
console.log(diffrence)

var product = 23 * 10 ;
console.log(product)

var quotion = 40 / 2 ;
console.log(quotion)

var hamid = 999;
hamid ++ ; // INCREMENT hamid from 999 to 1000 by adding 1
// hamid -- ; decrement hamid fromm 999 to 998 by removing 1
console.log(hamid)

var DecimalNumber = 0.44 ;// floating point number
var product = 2.4 * 2.3 ; // multipliying 2 float numbers
var quotion = 2.4 * 2.3 ; // devide decimales

//finding a remainder :
// ==> WE CAN DO X = X + 23 ; or X += 23 ;
// ==> WE CAN DO X = X - 23 ; or X =- 23 ;
// ==> WE CAN DO X = X  * 23 ; or X *= 23 ;
// ==> WE CAN DO X = X / 23 ; or X /= 23 ;
```

---

4. Strings & Manipulation

```javascript
var Name = "Abdou " ; //declare string variables
var myStr = "I'm an \"Automation ENG\" i m 23 years old \"i'm the best "
console.log(myStr);

var mysttr = `i'm hamid "from fès" `;
console.log(mysttr);

var string = "i'm Abdellah " + "I'm from Meknès"; //concatenating  strings  with plus operator
console.log(string);

var strr = "I'm first . ";//concatenating  strings  with plus equals operator
strr += "I'm seconde .";
console.log(strr)
```

```javascript
//concatenating  strings  with variables :
var hamid = " hamidou " ;
var said = "hello i'm  said and this is my frend " ;
said += hamid;
console.log(said);

//find lenth of string we use .lenth
var lastNameLength = 0 ;
var lastName = "Abdellah" ;
lastNameLength = lastName.length ;
console.log(lastNameLength) ;

//Braket Notation  to find  first  chracter in  string  for same exemple
var FirstLetterOfFirstName ="";
var firstName = "Abdellah";
FirstLetterOfFirstName = firstName[0];
console.log(FirstLetterOfFirstName);

var lastLetterofFirstName = firstName[firstName.length-1]; //firnd the laste letter of firstNamae without knowir
console.log(lastLetterofFirstName);
```

---

5. Arrays Basics

```javascript
//Store Multiple Values  with  Arrays for this exemple we have a string and a number
var ourArray = ["Abdellah", 23];
console.log(ourArray);

//Nested Array
var myarray = [["Abdella", 23], ["Hamid", 999]];
console.log(myarray);

//access Array data with  indexes
var ourARRAY = [23, 3334, 999];
var ourData = ourARRAY[1];
console.log(ourData);

//modify Array Data With Indexes
ourARRAY[1] = 444;
console.log(ourARRAY);

//Access Multi-Dimentionnal Arrays With Indexes
var ARRAy = [[1,2,3], [4,5,6], [7,8,9]];
var myDATA = ARRAy[1][2];
console.log(myDATA);
```

---

6. Array Methods

```javascript
//push something into an array
var smiya = ["Abdellah", 23];
smiya.push(["Hamid", 38]);
console.log(smiya);

//manipulate array with pop function ()
var removedFroMyArray = smiya.pop();
console.log(smiya);

//manipulate array with shift function
var removedfromArray = smiya.shift();
console.log(smiya);

//seconde exemple of using shift and unshift
var myaray = [["Abdellah", 23], ["Mahmoud", 2030]];
myaray.shift() //so the first array ["Abdellah", 23] is deleated
myaray.unshift(["joebonamassa", 44]); //  the first array deleated is replased by a new array
console.log(myaray);
```

Part 2: Functions & Scope

7. Functions - Reusable Code

```javascript
//==>Write reusable  code  with functions
//the function is set up like this " function +function name + () + { the code inside the 2 curly brackets is ru
//we call the function like this ==> "functionn name();"
//exemple of function :
function Abdellah(){
    console.log(" Heya , Helllo to the best Abdellah fl3alam");
}
Abdellah();
Abdellah();

//passing values to functions with arguments
//inside the function we can have some parameters there like inputs for the funcction for exemple
function ourFunctioWithArgs(a, b){
    console.log(a - b);
}
ourFunctioWithArgs(10, 2);
```

---

8. Scope - Global vs Local

```javascript
//✔■Global Scope and functions
//✔■scope refers to the visibility of the variables
//✔■variables outside the function have a globale scope
//✔■exemple of this globale scope
var DJ = 145;
function djhmida(){
    console.log(DJ);
}
djhmida();
//✔■if the variable is in decaled in the function and we call it outside the function the variable is not found

//✔■global vs local scope in function
//✔■we can have sometimes local and global variables declared wwith the same name !!
//==> the local variable took precedence over the global variable  (exemple 1)
var variable = "Messi";
function footballer(){
    variable = "Ronaldo";
    return variable; //You just returned a value. Nothing was printed to the console because return only sends a
}
console.log(footballer()); //It returns "Ronaldo" because the function changes the global variable variable from
console.log(variable);
```

---

9. Return Values

```javascript
//✔■Return a value from  a function  with return
function minustwo(num){
    return num - 2;
}
console.log(minustwo(100));
```

---

10. Understanding Undefined

```javascript
//✔■understanding undefined value  returned from a function
processed = 2;
function addfive(summ){
    summ = summ + 3 ;
```

```
    return summ //if we dont rutrn the varable we will have "undefined "
}
console.log("addfive = "+ addfive(3));
processed = addfive(9);
console.log("processed = "+ processed);

//■ JSON.stringify converts a JavaScript object, array, or value into a JSON-formatted string.
//■ We use it to send data to a server, store it in localStorage, or log it in a readable format.
function nextinline(arr, item){
    arr.push(item)
    //return item;
    return arr.shift(); //after: [2,3,4,5,6]
}
var testarray = [1, 2, 3, 4, 5];
console.log("before: " + JSON.stringify(testarray));
console.log(nextinline(testarray, 6));
console.log("after: " + JSON.stringify(testarray));
// ■nextinline(arr, item) adds 'item' to the end of the array.
// ■Using 'return item;' just returns the added item.
// ■Using 'return arr.shift();' removes and returns the first element (like a queue).
```

Part 3: Control Flow & Logic

11. Boolean Values & Conditions

```javascript
//TODO: Boolean Values and condition exemple 1:
function trueOrFalse1(wasThatTrue){
    if (wasThatTrue){
        return "Yes , that was true ";
    }
    return"No, that was false" ;
}
console.log(trueOrFalse1(true));

//TODO: Boolean Values and condition exemple 2: with duble condition  using and statment
function trueOrFalse2(wasThatTrue, wasThatTrue2){
    if (wasThatTrue && wasThatTrue2){
        return "Yes , that was true ";
    } else {
        return"No, that was false" ;
    }
}
console.log(trueOrFalse2(true, false));
console.log(trueOrFalse2(true, true));
```

---

12. Comparison Operators

```javascript
//Comparaison with the equality operator
function testEqual(age) {
    if (age == 23){
        return "Equal !";
    }
    return "Not equal !";
}
console.log(testEqual('23'));

//comparaison with the strict Equality Operator
function testEqual2(age) {
    if (age === 23){
        return "Equal !";
    }
    return "Not equal !"; //!: Difference between '==' and '===' compare line 195 and line 203
}
console.log(testEqual2("23"));

function testEqual3(age) {
    if (age != 23){
```

```
        return "Equal !";
    }
    return "Not equal !"; //!: Difference between '==' and '===' compare line 195 and line 203
}
console.log(testEqual3("23"));

// !: if (age !== 23){
// !: if (val> 10)
// !: if (val>= 10)
// !: if (val>= 10 || val > 2) ==>"||" means " the or statment"
```

---

13. If/Else Statements

```javascript
//TODO: exemple of if and else if statment
function testSize(num){
    if (num < 5){
        return "Tiny";
    } else if (num < 10){
        return "Small";
    } else if (num < 15){
        return "Medium";
    } else if (num < 20){
        return "Large";
    } else {
        return "Huge"
    }
}
console.log(testSize(4));
console.log(testSize(9));
console.log(testSize(14));
console.log(testSize(19));
console.log(testSize(1000));

// TODO: the par is the average number of strokes
var names = ["Hole-in-one!", "Eagle", "Birdie", "Par", "Bogey", "Double Bogey", "Go Home!!"]
function GolfScores(par, Stockes){
    if (Stockes == 1){
        return names[0];
    } else if (Stockes <= par - 2){
        return names[1];
    } else if (Stockes == par - 1){
        return names[2];
    } else if (Stockes == par){
        return names[3];
    } else if (Stockes == par + 1){
        return names[4];
    } else if (Stockes == par + 2){
        return names[5];
    } else if (Stockes >= par + 3){
        return names[6];
    }
}
console.log(GolfScores(5, 4));
```

---

14. Switch Statements

```javascript
//* Switch Statments usings "case and break"
//? what we call case and break in java script porogramming
//? when we use case and brak and when we use if can we use them both in some cases
// TODO : a function where we use case and break
function caseInSwithch(val){
    var answer = "";
    switch(val){
        case 1 :
            answer = "Alpha";
            break;
        case 2 :
```

```
            answer = "beta";
            break;
        case 3 :
            answer = "Gamma";
            break;
        case 4  :
            answer = "Delta";
            break;
    }
    return answer;
}
console.log(caseInSwithch(1));
console.log(caseInSwithch(2));
console.log(caseInSwithch(3));
console.log(caseInSwithch(4));
//!: switch checks one value (val) and matches it against different cases
//!: case = a possible value to compare with
//!: break = stop after finding the match (avoid running next cases)
//!: without break, it continues to next cases (fall-through)

//*Dedault  Option  in Switch Statements
//!==> A statement is a single instruction in JavaScript
//!==> Examples: assigning a value, calling a function, if, switch, return, loop
//!==> It tells the computer to do one specific action
```

---

15. Ternary Operator

```javascript
//todo => Use the conditionnal (Ternary) Operator
function checkEqual(a, b){
    return a === b ? true : false;
    return a === b;
}
checkEqual(1, 2);

//todo multiple ternary operator
function checkSign(num){
    return num > 0 ? "positive " : num < 0 ? "negative" : "zero"
}
console.log(checkSign(0));
```

Part 4: Objects & Data Structures

16. Building JavaScript Objects

```javascript
//todo => Build java script objects
    //? instead of using indexes  to access data we use  properties
    //* ==> name and  legs  and tails  and friends are  all properties  and after the colons we have the values
    //! ==> the propeties can be string or numbers or arrays or any data type in java script
var my_profile = { //* this is an exemple f an object
    "name": "Rahmouni Abdellah",
    "job" : "automation engineer",
    "programing langueges":"java script , python , html , css",
    "number of programming languages": 4,
};
```

---

17. Accessing Object Properties

```javascript
//if we want to access a property on an object :2 methdes
    //first methode : 1- Dot notation
        var job_name = my_profile.job; //!==> to find the value of the property inside the object
        //*-- after the dot we put the name of the property that we want to see the value
        console.log(job_name); // we will recieve the value of the property
```

```
    //seconde methode : 2- Bracket notation
        var job_name = my_profile["job"]; //!==> to find the value of the property inside the object
        console.log(job_name);

    // 3- accessing object properties with variables
    var test_numbers ={
        23 : "Abdelllah",
        38 : "Hamid",
    };
    var anonyme = 23;
    var programer_name = test_numbers[anonyme]
    console.log(programer_name)//* i used up the variable "programmer_name " to look up the pbject property

    //4- update project properties
    my_profile.job = "Software engineer";

    //5- add a new properties to an object
    my_profile.age = "23";
    my_profile['frend'] = "Hamid";

    //6 -delete properties from an object
    delete my_profile.number_of_programming_languages;

//! ==> we can use the objectiinsted of using the swich and case statment
function caseInSwithch(val){ //same swich case statment exemple !!!
    var answer = "";
    var lookup = {
        1 : "Alpha",
        2 : "beta",
        3: "Gamma" ,
        4: "Delta" ,
    };
    answer = lookup[val];
    return answer;
}
console.log(caseInSwithch(1));

// testing objects for propeties
var lookup = {
    1 : "Alpha",
    2 : "beta",
    3: "Gamma" ,
    4: "Delta" ,
};
function chkobj(checkProp){
    if (lookup.hasOwnProperty(checkProp)){
        return lookup[checkProp];
    } else {
        return"Not Found";
    }
}
console.log(chkobj(6));// 6 is not a property in the object thats why i will recieve not found
```

---

18. Nested Objects & Arrays

```javascript
//! a java script object is a whey to store flexible data
var MyMusic = [   //todo exemple 1
    {
        "artist":"PinkFloy",
        "title": "the wall",
        "release_year": 1970,
        "formats":[
            "CD",
            "vidéo",
        ],
    },
    {
        "artist":"joeBonamassa",
        "title": "midnight blues",
        "release_year": 2000,
        "formats":[
```

```
            "CD",
            "vidéo",
        ],
    },
]

//todo exxemple 2
var my_storage = {
    "car": {
        "inside": {
            "glove box":"maps",
            "pasenger seat":"crumbs",
        },
        "outside": {
            "trunk":"jack",
        },
    },
};
var gloveBoxContents = my_storage.car.inside["glove box"];
console.log(gloveBoxContents);

//?  explanation of my questions using an exemple
var x = 5;  // semicolon ends the statement

obj = {
    name: "Ahmed",  // comma separates properties
    age: 25         // no comma on last property
}; // semicolon ends the whole declaration

//! Objects organize related data - used in every project!

//todo exemple where have an array with multipale objects
var MyMusic = [
    {
        "artist":"PinkFloy",
        "title": "the wall",
        "release_year": 1970,
        "formats":[
            "CD",
            "vidéo",
        ],
    },
    {
        "artist":"joeBonamassa",
        "title": "midnight blues",
        "release_year": 2000,
        "formats":[
            "CD",
            "vidéo",
        ],
    },
]
var todays_music = MyMusic[1].formats[1];// to foundsomething inside the object exactly inside the array
console.log(todays_music);
```

---

19. Record Collection Project

```javascript
//todo ==> Record Collection the goal is to creat an updateRecords function
// MUSIC COLLECTION - Object containing multiple albums
var collection = {
    "4122": {
        album: "The Thrill Is Gone",
        artist: "B.B. King",
        tracks: [
            "The Thrill Is Gone",
            "Sweet Little Angel"
        ]
    },
    "5289": {
        album: "Born Under a Bad Sign",
        artist: "Albert King",
```

```
        tracks: [ ]
    },
    "6344": {
        album: "At Last!",
    },
    "7455": {
        album: "Texas Flood",
        artist: "Stevie Ray Vaughan",
        tracks: [
            "Pride and Joy",
            "Texas Flood"
        ]
    }
};

// JSON.stringify() - Converts object to text string
// JSON.parse() - Converts text string back to object
// Together they create a deep copy (backup) of the collection
var collectionCopy = JSON.parse(JSON.stringify(collection));

function updateRecords(id, prop, value) {
    if (value === "") {
        delete collection[id][prop];  // Empty value = delete property
    } else if (prop === "tracks") {
        collection[id][prop] = collection[id][prop] || [];  // Create array if missing
        collection[id][prop].push(value);  // Add track to array
    } else {
        collection[id][prop] = value;  // Set any other property
    }
    return collection;
}

updateRecords(7455,"tracks","blues goat");
console.log(updateRecords(6344,"artist","Abdellah"));
```

Part 5: Loops & Iteration

20. While Loops

```javascript
//todo ==> Iterate with while loops
//LOOPS ALLOW YOU TO RUN THE SAME CODE Multiple times
//while loop it runs when its true and it stopes when it's not true
var myArray = [];
var i = 0;
while (i <= 10){
    myArray.push(i);
    i++;
}
console.log(myArray);
```

---

21. For Loops

```javascript
//the "for loop"
var myforloop = [];
for(var i = 1; i < 10; i += 2){ // we start by initianlisation of the variable i then seconde the condition ther
    myforloop.push(i);
}
console.log(myforloop);

var myForloop = [];
for(var i = 10; i > 0; i -= 2){ //decriment by 2 using for loop
    myForloop.push(i);
}
console.log(myForloop);
```

---

22. Iterating Through Arrays

```javascript
//todo iterate through an array with  a for loop
var ourAR = [10, 13, 15, 23];  // Array of numbers
var ourTotal = 0;              // Variable to store sum

// .length = number of items in array (here: 4)
// Loop runs while i < 4 (indexes: 0, 1, 2, 3)
for (var i = 0; i < ourAR.length; i++) {
    ourTotal += ourAR[i];  // Add current number to total
}

console.log(ourTotal);  // Result: 61 (10+13+15+23)
```

---

23. Nesting For Loops

```javascript
//todo ==> Netsing for loops
function multiplyAll(arr){
    var product = 1;
    for (var i = 0; i < arr.length; i++){ // i is outer array
        for (var j = 0; j < arr[i].length; j++){ // J is inter array
            product *= arr[i][j]
        }
    }
    return product;
}
var product = multiplyAll([[1,2],[3,4],[5,6,7]]);
console.log(product);

//Array: [[1,2], [3,4], [5,6,7]]
//Starting product = 1

//OUTER LOOP 1: i = 0 → arr[0] = [1, 2]
 // INNER LOOP 1: j = 0 → arr[0][0] = 1
 //   product = 1 * 1 = 1
 // INNER LOOP 2: j = 1 → arr[0][1] = 2
 //   product = 1 * 2 = 2

//OUTER LOOP 2: i = 1 → arr[1] = [3, 4]
 // INNER LOOP 1: j = 0 → arr[1][0] = 3
 //   product = 2 * 3 = 6
 // INNER LOOP 2: j = 1 → arr[1][1] = 4
   // product = 6 * 4 = 24

//OUTER LOOP 3: i = 2 → arr[2] = [5, 6, 7]
  //INNER LOOP 1: j = 0 → arr[2][0] = 5
  //  product = 24 * 5 = 120
  //INNER LOOP 2: j = 1 → arr[2][1] = 6
 //   product = 120 * 6 = 720
  //INNER LOOP 3: j = 2 → arr[2][2] = 7
  //  product = 720 * 7 = 5040 ■

//Final Result: 5040
```

---

24. Do-While Loops

```javascript
//todo ==> do while loops
// while loop check a condition before runs any code withing the loop
//the do while will run  once before cheking condition
var myArray = [];
var i = 0;
do{
    myArray.push(i);
    i++;
}while (i <= 10)
```

```
console.log(myArray);

//*                    ===== SUMMARY =====
//*    WHILE: Check first, then run → Might not run at all
//*    DO-WHILE: Run first, then check → Always runs at least once
```

---

25. Profile Lookup Example

```javascript
//todo exemple profile lookup
var contacts = [
    {
        firstName: "John",
        lastName: "Doe",
        number: "555-1234",
        likes: ["pizza", "gaming", "movies"]
    },
    {
        firstName: "Sarah",
        lastName: "Smith",
        number: "555-5678",
        likes: ["tacos", "music", "reading"]
    },
    {
        firstName: "Mike",
        lastName: "Johnson",
        number: "555-9012",
        likes: ["football", "coding", "coffee"]
    },
    {
        firstName: "Emma",
        lastName: "Williams",
        number: "unknown",
        likes: ["gaming", "tacos", "traveling"]
    }
];

function lookUpProfile(name, prop){
    for(var i = 0; i < contacts.length; i++){
        if(contacts[i].firstName === name){
            return contacts[i][prop] || "no such property";
        }
    }
    return "No Such contact";
}

var data = lookUpProfile("Mike","likes");
console.log(data);
```

Part 6: Utility Functions

26. Random Numbers

```javascript
//todo random Fraction
function randomFraction(){
    return Math.random();
}
console.log(randomFraction());

//todo generate random whole number
var randomNumberBetween0and19 = Math.floor(Math.random() * 20);

//exemple 2
function randomWholeNumber(){
    return Math.floor(Math.random() * 10);
}

//todo generate random whole numbers within a range
```

```
function ourRandomRange(ourMin, ourMax){
    return Math.floor(Math.random() * (ourMax - ourMin + 1)) + ourMin;
}
ourRandomRange(1, 9);
```

---

27. parseInt Function

```javascript
//todo use The parseInt Function
function convertToInteger(str){
    return parseInt(str);
}
convertToInteger("56");
convertToInteger("10011");  //function convertToInteger(str,2)
//retrn binary code to integer
```

Part 7: Modern JavaScript (ES6+)

28. Var vs Let - Scope Differences

```javascript
//* difference between var and let
//* var is globaly
//* let is
//? block scope and function scope

/* ========================================
   VAR vs LET - Key Differences
   ======================================== */

// 1. SCOPE DIFFERENCE
// -------------------
// var = Function Scope (escapes blocks)
// let = Block Scope (trapped in blocks)

if (true) {
    var x = 10;  // escapes the if block
    let y = 20;  // trapped inside the if block
}
console.log(x);  // 10 - works!
console.log(y);  // ERROR - y is not defined


// 2. REDECLARING
// --------------
// var = CAN redeclare (dangerous!)
// let = CANNOT redeclare (safe!)

var name = "Alice";
var name = "Bob";    // No error - name is now "Bob"

//let age = 25;
let age = 30;        // ERROR - already declared


// 3. LOOP PROBLEM
// ---------------
// var = shares same variable across loop
// let = creates new variable for each iteration

for (var i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 1000);
}
// Prints: 3, 3, 3

for (let j = 0; j < 3; j++) {
    setTimeout(() => console.log(j), 1000);
}
// Prints: 0, 1, 2
```

```
// RULE: Always use 'let' (or 'const') in modern JavaScript!
// Only use 'var' when working with old code
```

---

29. Const - Read-Only Variables

```javascript
//TODO: Declare a read -only variable with the const Keyword
/* ========================================
   LET vs CONST - Complete Comparison
   ======================================== */

/*
   LET (Modern Way)
   ----------------
   - Block Scope (trapped in blocks)
   - CAN be reassigned
   - CANNOT be redeclared
   - Does NOT need initialization
*/

// let example - block scope
// if (true) {
//     let y = 20;
// }
// console.log(y); // ERROR - trapped inside!

// let example - can reassign
// let age = 25;
// age = 30; // Works - we can change the value

// let example - cannot redeclare
// let score = 100;
// let score = 200; // ERROR - already declared

// let example - no need to initialize
// let name;
// name = "Alice"; // Works


/*
   CONST (Modern Way - Use Most Often!)
   ------------------------------------
   - Block Scope (trapped in blocks)
   - CANNOT be reassigned
   - CANNOT be redeclared
   - MUST be initialized immediately
*/

// const example - cannot reassign
// const PI = 3.14;
// PI = 3.15; // ERROR - cannot change

// const example - must initialize
// const city; // ERROR - must give value immediately
// const city = "Paris"; // Correct

// const with objects - can change inside
// const person = { name: "John" };
// person.name = "Alice"; // Works - changing property
// person = { name: "Bob" }; // ERROR - cannot reassign whole object

// const with arrays - can change inside
// const numbers = [1, 2, 3];
// numbers.push(4); // Works - modifying array
// numbers = [5, 6]; // ERROR - cannot reassign whole array


/* ========================================
   COMPARISON TABLE
   ========================================
```

```
   Feature          | let      | const
   -----------------|----------|----------
   Scope            | Block    | Block
   Can Reassign?    | YES      | NO
   Can Redeclare?   | NO       | NO
   Must Initialize? | NO       | YES
   Use When?        | Value changes | Value stays same

*/


/* ========================================
   WHEN TO USE EACH?
   ======================================== */

// Use CONST by default (most common)
// const userName = "Alice";
// const MAX_USERS = 100;
// const colors = ["red", "blue"];

// Use LET only when you need to reassign
// let counter = 0;
// counter++;
//
// let total = 100;
// total = total + 50;


/* ========================================
   MEMORY TRICK
   ======================================== */

// let = "I might change this value later"
// const = "This value stays the same"


/* ========================================
   BEST PRACTICE RULE
   ======================================== */

// 1. Start with CONST
// 2. Change to LET only if you need to reassign
// 3. Both are block-scoped and safe to use
```

---

30. Object.freeze()

```javascript
// If we have for exemple :
function freezeObj(){
    "use strict"
    const MATH_CONSTANTS = {
        PI: 3.14
    }; //if we want to freeze this variable we use object.freeze
    Object.freeze(MATH_CONSTANTS);
}
const PI = freezeObj()
```

---

31. Arrow Functions

```javascript
//TODO use arrow Functions  to write consise anonymous Functions
var magic = function(){   //! ==> it's an anoonymous function we can converted to an errow function
    return new Date();
};

//*it beomes :
const magicc = () => new Date();
```

```
//todo write arrow functions with parameters
//? what are arrow functions were and why use them
```

---

32. Spread Operator

```javascript
//*use spred operator
// arr2=[...arr1]; why using the 3 points before arr1
```

---

33. Destructuring Assignment

```javascript
// Destructuring Assignment

/* ========================================
   DESTRUCTURING ASSIGNMENT
   ======================================== */

/*
   What is Destructuring?
   ----------------------
   A way to "unpack" values from arrays or objects
   into separate variables in one line
   Like opening a box and taking out specific items
*/


/* ========================================
   ARRAY DESTRUCTURING
   ======================================== */

// WITHOUT Destructuring (old way)
// const colors = ["red", "green", "blue"];
// const color1 = colors[0]; // "red"
// const color2 = colors[1]; // "green"
// const color3 = colors[2]; // "blue"

// WITH Destructuring (modern way - much cleaner!)
// const colors = ["red", "green", "blue"];
// const [color1, color2, color3] = colors;
// console.log(color1); // "red"
// console.log(color2); // "green"
// console.log(color3); // "blue"


// Skipping elements
// const numbers = [1, 2, 3, 4, 5];
// const [first, , third] = numbers; // skip second element
// console.log(first);  // 1
// console.log(third);  // 3

// Getting rest of elements with spread operator
// const numbers = [1, 2, 3, 4, 5];
// const [first, second, ...rest] = numbers;
// console.log(first);  // 1
// console.log(second); // 2
// console.log(rest);   // [3, 4, 5]

// Default values
// const colors = ["red"];
// const [first, second = "blue"] = colors;
// console.log(first);  // "red"
// console.log(second); // "blue" (default value)

// Swapping variables (cool trick!)
// let a = 1;
// let b = 2;
// [a, b] = [b, a]; // swap values
// console.log(a); // 2
```

```
// console.log(b); // 1


/* ========================================
   OBJECT DESTRUCTURING
   ======================================== */

// WITHOUT Destructuring (old way)
// const person = { name: "Alice", age: 25, city: "Paris" };
// const name = person.name;
// const age = person.age;
// const city = person.city;

// WITH Destructuring (modern way - cleaner!)
// const person = { name: "Alice", age: 25, city: "Paris" };
// const { name, age, city } = person;
// console.log(name); // "Alice"
// console.log(age);  // 25
// console.log(city); // "Paris"

/*
   IMPORTANT: Variable names must match property names!
*/
// const person = { name: "Alice", age: 25 };
// const { name, age } = person; // Works - names match
// const { firstName, age } = person; // firstName = undefined (no match!)


// Renaming variables while destructuring
// const person = { name: "Alice", age: 25 };
// const { name: firstName, age: userAge } = person;
// console.log(firstName); // "Alice"
// console.log(userAge);   // 25
// console.log(name);      // ERROR - name doesn't exist

// Default values
// const person = { name: "Alice" };
// const { name, age = 30 } = person;
// console.log(name); // "Alice"
// console.log(age);  // 30 (default value)

// Nested destructuring
// const person = {
//     name: "Alice",
//     address: {
//         city: "Paris",
//         country: "France"
//     }
// };
// const { name, address: { city, country } } = person;
// console.log(city);    // "Paris"
// console.log(country); // "France"

// Getting rest of properties
// const person = { name: "Alice", age: 25, city: "Paris", job: "Developer" };
// const { name, age, ...others } = person;
// console.log(name);   // "Alice"
// console.log(age);    // 25
// console.log(others); // { city: "Paris", job: "Developer" }


/* ========================================
   WHY USE DESTRUCTURING?
   ======================================== */

/*
   Benefits:
   ---------
   1. Less code to write
   2. Cleaner and easier to read
   3. No need to repeat object/array names
   4. Can set default values easily
   5. Common in modern JavaScript (React, Node.js, etc.)
*/
```

```
/* ========================================
   COMMON USE CASES
   ======================================== */

// 1. Function parameters (very common!)
// Instead of this:
// function greet(person) {
//     console.log(`Hello ${person.name}, you are ${person.age} years old`);
// }

// Do this (cleaner!):
// function greet({ name, age }) {
//     console.log(`Hello ${name}, you are ${age} years old`);
// }
// greet({ name: "Alice", age: 25 });


// 2. Working with function returns
// function getUser() {
//     return { name: "Alice", age: 25, city: "Paris" };
// }
// const { name, age } = getUser(); // Extract only what you need


// 3. Working with API responses
// fetch('api/user')
//     .then(response => response.json())
//     .then(({ name, email, id }) => {
//         // Only extract what you need from the response
//         console.log(name, email, id);
//     });


// 4. Importing specific functions from modules
// import { useState, useEffect } from 'react'; // destructuring!


/* ========================================
   COMPARISON: Before vs After
   ======================================== */

// BEFORE Destructuring (repetitive and long)
// const user = { name: "Alice", age: 25, email: "alice@email.com" };
// const name = user.name;
// const age = user.age;
// const email = user.email;
// console.log(name, age, email);

// AFTER Destructuring (clean and short)
// const user = { name: "Alice", age: 25, email: "alice@email.com" };
// const { name, age, email } = user;
// console.log(name, age, email);


/* ========================================
   KEY POINTS TO REMEMBER
   ======================================== */

// Array Destructuring:
// - Uses square brackets []
// - Order matters (position-based)
// - const [a, b, c] = array;

// Object Destructuring:
// - Uses curly braces {}
// - Variable names must match property names
// - Order doesn't matter
// - const { name, age } = object;

// Both can have:
// - Default values
// - Rest operator (...)
// - Nested destructuring
```

```
/* ========================================
   MEMORY TRICK
   ======================================== */

// Destructuring = Unpacking a suitcase
// - Array destructuring: Take items in order [1, 2, 3]
// - Object destructuring: Take specific labeled items {name, age}
// - You choose what to unpack, leave the rest packed
```

---

34. Destructuring with Nested Objects

```javascript
/* ========================================
   DESTRUCTURING WITH NESTED OBJECTS
   ======================================== */

/*
   What is a Nested Object?
   ------------------------
   An object that contains another object inside it
   Like a box inside a box
*/

// Example of nested object:
// const person = {
//     name: "Alice",
//     age: 25,
//     address: {              // nested object!
//         city: "Paris",
//         country: "France",
//         zipCode: "75001"
//     }
// };


/* ========================================
   METHOD 1: Basic Nested Destructuring
   ======================================== */

// WITHOUT Destructuring (old way - long!)
// const person = {
//     name: "Alice",
//     address: {
//         city: "Paris",
//         country: "France"
//     }
// };
// const name = person.name;
// const city = person.address.city;
// const country = person.address.country;


// WITH Nested Destructuring (modern way - clean!)
// const person = {
//     name: "Alice",
//     address: {
//         city: "Paris",
//         country: "France"
//     }
// };
// const { name, address: { city, country } } = person;
// console.log(name);    // "Alice"
// console.log(city);    // "Paris"
// console.log(country); // "France"

/*
   IMPORTANT: After destructuring like this,
   'address' variable does NOT exist!
   Only 'city' and 'country' exist.
*/
```

```
/* ========================================
   METHOD 2: Get Both Parent and Children
   ======================================== */

// If you want BOTH the address object AND its properties:
// const person = {
//     name: "Alice",
//     address: {
//         city: "Paris",
//         country: "France"
//     }
// };

// Destructure in TWO steps:
// const { name, address } = person;
// const { city, country } = address;
// console.log(name);    // "Alice"
// console.log(address); // { city: "Paris", country: "France" }
// console.log(city);    // "Paris"
// console.log(country); // "France"

// OR do it all in one line:
// const {
//     name,
//     address,
//     address: { city, country }
// } = person;
// Now you have: name, address, city, country all available


/* ========================================
   METHOD 3: Deep Nesting (3+ Levels)
   ======================================== */

// const user = {
//     id: 1,
//     name: "Alice",
//     location: {
//         address: {
//             street: "Main St",
//             city: "Paris",
//             coordinates: {
//                 lat: 48.8566,
//                 lng: 2.3522
//             }
//         }
//     }
// };

// Destructuring deep nested objects:
// const {
//     name,
//     location: {
//         address: {
//             city,
//             coordinates: { lat, lng }
//         }
//     }
// } = user;

// console.log(name); // "Alice"
// console.log(city); // "Paris"
// console.log(lat);  // 48.8566
// console.log(lng);  // 2.3522


/* ========================================
   PRACTICE EXERCISE
   ======================================== */

// Try destructuring this yourself:
// const company = {
//     name: "TechCorp",
//     ceo: {
```

```
//         firstName: "John",
//         lastName: "Doe",
//         contact: {
//             email: "john@techcorp.com",
//             phone: "555-0100"
//         }
//     },
//     location: {
//         headquarters: {
//             city: "San Francisco",
//             state: "CA"
//         }
//     }
// };

// Goal: Extract companyName, ceoFirstName, ceoEmail, and headquartersCity
```

---

35. Template Literals

```javascript
/* ========================================
   TEMPLATE LITERALS (TEMPLATE STRINGS)
   ======================================== */

/*
   What are Template Literals?
   ---------------------------
   A modern way to create strings using backticks ``
   Can include variables and expressions directly
   Can span multiple lines easily
*/

// OLD WAY: String concatenation (ugly!)
// const name = "Alice";
// const age = 25;
// const message = "Hello, my name is " + name + " and I am " + age + " years old.";
// console.log(message);

// NEW WAY: Template Literals (beautiful!)
// const name = "Alice";
// const age = 25;
// const message = `Hello, my name is ${name} and I am ${age} years old.`;
// console.log(message);

/*
   Syntax: Use backticks `` and ${} for variables/expressions
*/

// Example 1: Simple variables
// const firstName = "Alice";
// const lastName = "Smith";
// const fullName = `${firstName} ${lastName}`;
// console.log(fullName); // "Alice Smith"

// Example 2: Expressions inside ${}
// const a = 5;
// const b = 10;
// console.log(`The sum is: ${a + b}`); // "The sum is: 15"
// console.log(`Double of a is: ${a * 2}`); // "Double of a is: 10"

// Example 3: Multi-line strings (no \n needed!)
// const message = `
//     Hello,
//     This is a multi-line
//     string using template literals.
//     Very easy!
// `;
// console.log(message);

// Example 4: Calling functions inside ${}
// function getDiscount(price) {
//     return price * 0.1;
```

```
// }
// const price = 100;
// console.log(`Price: ${price}, Discount: ${getDiscount(price)}`);
// Output: "Price: $100, Discount: $10"
```

---

36. Concise Object Literals

```javascript
/* ========================================
   CONCISE OBJECT LITERAL DECLARATIONS
   ======================================== */

/*
   What is Concise Object Literal?
   -------------------------------
   A shorter way to write object properties and methods
   When variable name = property name, you can write it once!
*/

// OLD WAY (repetitive):
// const name = "Alice";
// const age = 25;
// const city = "Paris";
// const person = {
//     name: name,      // repetitive!
//     age: age,        // repetitive!
//     city: city       // repetitive!
// };

// NEW WAY (concise - ES6):
// const name = "Alice";
// const age = 25;
// const city = "Paris";
// const person = { name, age, city }; // Much shorter!
// console.log(person); // { name: "Alice", age: 25, city: "Paris" }


/*
   Concise Method Syntax
   ---------------------
*/

// OLD WAY:
// const person = {
//     name: "Alice",
//     greet: function() {
//         console.log(`Hello, I'm ${this.name}`);
//     }
// };

// NEW WAY (concise methods):
// const person = {
//     name: "Alice",
//     greet() {  // No 'function' keyword needed!
//         console.log(`Hello, I'm ${this.name}`);
//     }
// };
// person.greet(); // "Hello, I'm Alice"


/*
   Computed Property Names
   -----------------------
*/

// Dynamic property names using []
// const propName = "age";
// const person = {
//     name: "Alice",
//     [propName]: 25,  // property name from variable
//     [`user_${propName}`]: 25  // can use expressions!
// };
```

```
// console.log(person); // { name: "Alice", age: 25, user_age: 25 }
```

---

37. Classes & Constructors

```javascript
/* ========================================
   CLASS SYNTAX - CONSTRUCTOR FUNCTION
   ======================================== */

/*
   What is a Class?
   ----------------
   A blueprint for creating objects
   Modern way to create objects with constructor (ES6)
*/

// Basic Class Structure
// class Person {
//     // Constructor - runs when you create new object
//     constructor(name, age) {
//         this.name = name;  // this = the new object being created
//         this.age = age;
//     }

//     // Method (function inside class)
//     greet() {
//         return `Hello, my name is ${this.name}`;
//     }

//     // Another method
//     getAge() {
//         return this.age;
//     }
// }

// Creating objects from class (instances)
// const person1 = new Person("Alice", 25);
// const person2 = new Person("Bob", 30);

// console.log(person1.name);      // "Alice"
// console.log(person1.greet());   // "Hello, my name is Alice"
// console.log(person2.greet());   // "Hello, my name is Bob"


// Example 2: Class Inheritance (extending classes)
// class Animal {
//     constructor(name) {
//         this.name = name;
//     }

//     speak() {
//         return `${this.name} makes a sound`;
//     }
// }

// class Dog extends Animal {  // Dog inherits from Animal
//     constructor(name, breed) {
//         super(name);  // Call parent constructor
//         this.breed = breed;
//     }

//     speak() {  // Override parent method
//         return `${this.name} barks!`;
//     }

//     getBreed() {
//         return `${this.name} is a ${this.breed}`;
//     }
// }

// const dog = new Dog("Max", "Golden Retriever");
// console.log(dog.speak());     // "Max barks!"
```

```
// console.log(dog.getBreed());  // "Max is a Golden Retriever"
```

---

38. Getters and Setters

```javascript
/* ========================================
   GETTERS AND SETTERS
   ======================================== */

/*
   What are Getters and Setters?
   -----------------------------
   Special methods to control how you access/modify object properties
   - Getter: Read a property (get)
   - Setter: Write/change a property (set)
   - Used for validation, computed properties, encapsulation
*/

// Basic Getter and Setter
// class Person {
//     constructor(firstName, lastName) {
//         this._firstName = firstName;  // _ means "private by convention"
//         this._lastName = lastName;
//     }

//     // GETTER - access like a property, not a function
//     get fullName() {
//         return `${this._firstName} ${this._lastName}`;
//     }

//     // SETTER - set like a property, not a function
//     set fullName(name) {
//         const parts = name.split(' ');
//         this._firstName = parts[0];
//         this._lastName = parts[1];
//     }

//     // Another getter
//     get firstName() {
//         return this._firstName;
//     }

//     // Another setter with validation
//     set firstName(name) {
//         if (name.length < 2) {
//             console.log("Name too short!");
//             return;
//         }
//         this._firstName = name;
//     }
// }

// const person = new Person("Alice", "Smith");

// Using GETTER (no parentheses!)
// console.log(person.fullName);  // "Alice Smith" - like a property!

// Using SETTER (no parentheses!)
// person.fullName = "Bob Johnson";  // like assigning to a property!
// console.log(person.firstName);    // "Bob"

/*
   Why use Getters and Setters?
   -----------------------------
   1. Validation - check values before setting
   2. Computed properties - calculate values on the fly
   3. Encapsulation - hide internal implementation
   4. Read-only properties - getter without setter
   5. Logging - track when properties are accessed/changed
*/
```

---

39. Modules - Import vs Require

```javascript
/* ========================================
   IMPORT vs REQUIRE (ES6 MODULES vs COMMONJS)
   ======================================== */

/*
   Two Module Systems in JavaScript
   --------------------------------
   1. CommonJS (require/module.exports) - Old way, used in Node.js
   2. ES6 Modules (import/export) - Modern way, standard JavaScript
*/

/*
   COMMONJS (require/module.exports)
   ---------------------------------
   - Used in older Node.js code
   - Synchronous loading
   - Dynamic imports possible
*/

// Exporting with CommonJS (in math.js file):
// function add(a, b) {
//     return a + b;
// }
// function subtract(a, b) {
//     return a - b;
// }
// module.exports = { add, subtract };  // CommonJS export

// Importing with CommonJS (in app.js file):
// const math = require('./math');  // CommonJS import
// console.log(math.add(5, 3));     // 8


/*
   ES6 MODULES (import/export)
   ---------------------------
   - Modern standard way
   - Better for browsers
   - Asynchronous loading
   - Static imports (analyzed before code runs)
   - Must use .mjs extension or "type": "module" in package.json
*/

// Exporting with ES6 (in math.js file):
// export function add(a, b) {
//     return a + b;
// }
// export function subtract(a, b) {
//     return a - b;
// }

// Importing with ES6 (in app.js file):
// import { add, subtract } from './math.js';  // ES6 import
// console.log(add(5, 3));  // 8


/*
   KEY DIFFERENCES
   ---------------

   Feature          | require (CommonJS)    | import (ES6)
   -----------------|-----------------------|-----------------
   Syntax           | const x = require()   | import x from
   Loading          | Synchronous           | Asynchronous
   When loaded      | Runtime (dynamic)     | Before runtime (static)
   Browser support  | No (needs bundler)    | Yes (modern browsers)
   Where to use     | Node.js (old)         | Modern JS, React, etc
   File extension   | .js                   | .js or .mjs
*/
```

---

40. Export - Reuse Code Blocks

```javascript
/* ========================================
   EXPORT - Reuse Code Blocks
   ======================================== */

/*
   Named Exports (can export multiple things)
   ------------------------------------------
*/

// Method 1: Export individually (in utils.js)
// export const PI = 3.14159;
// export function square(x) {
//     return x * x;
// }
// export function cube(x) {
//     return x * x * x;
// }

// Method 2: Export at the end (in utils.js)
// const PI = 3.14159;
// function square(x) {
//     return x * x;
// }
// function cube(x) {
//     return x * x * x;
// }
// export { PI, square, cube };

// Method 3: Export with rename (in utils.js)
// function square(x) {
//     return x * x;
// }
// export { square as sq };  // export with different name


/*
   Importing Named Exports
   -----------------------
*/

// Import specific items (in app.js)
// import { PI, square, cube } from './utils.js';
// console.log(PI);        // 3.14159
// console.log(square(5)); // 25

// Import with rename
// import { square as sq } from './utils.js';
// console.log(sq(5));  // 25
```

---

41. Import Everything (*)

```javascript
/* ========================================
   IMPORT EVERYTHING FROM A FILE (*)
   ======================================== */

/*
   Import Everything with * (Namespace Import)
   -------------------------------------------
   Gets ALL exports from a file as one object
*/

// File: math.js
// export const PI = 3.14159;
// export function add(a, b) { return a + b; }
// export function subtract(a, b) { return a - b; }
```

```
// export function multiply(a, b) { return a * b; }

// File: app.js
// import * as math from './math.js';  // Import EVERYTHING

// Now use with namespace:
// console.log(math.PI);              // 3.14159
// console.log(math.add(5, 3));       // 8
// console.log(math.subtract(10, 4)); // 6
// console.log(math.multiply(3, 7));  // 21

/*
   When to use import *?
   - When you need many exports from a file
   - Keeps code organized with namespace
   - Avoids naming conflicts
*/
```

---

42. Default Export & Fallback

```javascript
/* ========================================
   DEFAULT EXPORT & IMPORT
   ======================================== */

/*
   Default Export
   --------------
   One "main" export per file (can only have ONE default export)
*/

// File: User.js (exporting a class as default)
// export default class User {
//     constructor(name) {
//         this.name = name;
//     }
//     greet() {
//         return `Hello, I'm ${this.name}`;
//     }
// }

// File: app.js (importing default export)
// import User from './User.js';  // No curly braces for default!
// const user = new User("Alice");
// console.log(user.greet());  // "Hello, I'm Alice"


/*
   Mixing Default and Named Exports
   --------------------------------
   You can have ONE default + multiple named exports
*/

// File: utils.js
// export default function mainFunction() {
//     return "Main function";
// }
// export const helper1 = () => "Helper 1";
// export const helper2 = () => "Helper 2";

// File: app.js
// import mainFunction, { helper1, helper2 } from './utils.js';
// console.log(mainFunction());  // "Main function"
// console.log(helper1());       // "Helper 1"


/*
   Export Fallback with Default Export
   -----------------------------------
   "Fallback"
```