# Summer Homework – Computer Science JavaScript Notes

1. Comments

• // is used for inline comments.  
Example: var pixels = 5; // inline comment  
  
• /\* ... \*/ is used for multi-line comments.  
Example:  
/\*  
 This can be used for  
 multiple lines because multi-line  
 comments work like this.  
\*/  
  
Tip: Use comments to explain what your code is doing.

## 2. Data Types

JavaScript has 7 data types (6 primitive + objects):  
• undefined – a declared variable with no value yet.  
• null – an intentional 'empty' value (nothing).  
• boolean – true or false.  
• string – text inside quotes.  
• symbol – a unique and immutable primitive value.  
• number – integers and decimals (NaN and Infinity are special number values).  
• object – collections of key–value pairs (arrays, functions, etc. are objects).  
  
Examples: var myName = "Laith"; let ourCollege = "BHASVIC"; var age = 10;

## 3. Variables: var, let, const

• var – function‑scoped; available throughout the function it is defined in.  
• let – block‑scoped; only available inside the nearest { } block.  
• const – block‑scoped and cannot be reassigned (use for constants).  
  
Example: const PI = 3.14159;

## 4. Storing Values with Assignment Operators

Semicolons are optional in JavaScript, but using them is good practice.  
Example:  
var a;  
var b = 2;  
a = 7;  
b = a;  
console.log(a); // prints 7

## 5. Initialising Variables & Case Sensitivity

You can declare and assign in one line: var x = 5;  
Practice:  
var A = 15;  
var D = 20;  
var C = 25;  
  
JavaScript is case‑sensitive: A and a are different variable names.  
Use camelCase for variable names, e.g., totalScore, lastName.

## 6. Increment and Decrement

• Increment adds 1: myVar++; (same as myVar = myVar + 1)  
• Decrement subtracts 1: myVar--;  
Example: var myVar = 11; myVar++; myVar--;

## 7. Decimal Numbers & Remainders

• Decimal example: var decimal = 5.7;  
• Remainder operator % returns the remainder after division:  
Example: 11 % 3; // 2

## 8. Escaping Quotes & Escape Sequences

To include quotes inside a string, either escape them with a backslash or use different outer quotes.  
  
Examples:  
var s1 = "I am a "double‑quoted" string";  
var s2 = 'I am a "double‑quoted" string';  
Backticks (`) allow both single and double quotes without escaping:  
var s3 = `He said, "It's fine."`;  
  
Common escape sequences:  
• \' single quote  
• \" double quote  
• \\ backslash  
• \n new line  
• \r carriage return  
• \t tab  
• \b backspace  
• \f form feed

## 9. Strings: Length, Indexing, Immutability

• str.length returns the number of characters in a string.  
• Indexing starts at 0 (zero‑based). str[0] is the first character.  
• Strings are immutable: you cannot change a single character in place; create a new string instead.  
  
Example:  
var firstName = "Adnan";  
var firstLetter = firstName[0]; // 'A'  
var lastLetter = firstName[firstName.length - 1];

10. String Concatenation

• Use + to join strings:  
var myStr = "This is the start " + "and this is the end.";  
  
• Use += to append:  
var greeting = "Hello ";  
greeting += "Adnan!";  
console.log(greeting); // Hello Adnan!

## 11. Constructing Strings with Variables & Bracket Notation

Example:  
var myName = "Devu";  
var sentence = "My name is " + myName + "!";  
  
Bracket notation recap:  
• Position 0 is always the first character.  
• Example: var firstInitial = myName[0];

12. Function Practice – Word Blanks

Set‑up a function that builds a sentence from words passed in:  
function wordBlanks(myNoun, myAdj, myVerb, myAdverb) {  
 var result = "";  
 result += "The " + myAdj + " " + myNoun + " " + myVerb + " " + myAdverb + ".";  
 return result;  
}  
console.log(wordBlanks("dog", "big", "ran", "quickly"));

## 13. Arrays

• Arrays always start with a square bracket [ ]. Each element is separated by a comma.  
• Every element in the array is accessed by its index (starting from 0).  
  
Example: var myArray = ["Smith", 17];  
  
Nested Arrays  
Example: var myArray = [["Bulls", 23], ["Cod", 32]];  
  
Accessing Arrays  
You can access array elements using their index number.  
Example: var ourArray = [50, 60, 70]; var ourData = ourArray[0]; // 50  
  
Modifying Array Data  
You can change array values directly by assigning new values using the index.  
Example: myArray[0] = 2;  
  
Accessing Multi-Dimensional Arrays  
You can access nested arrays using multiple index values.  
Example: var myArray = [[1,2,3], [4,5,6], [7,8,9]]; var myData = myArray[0][0]; // 1  
  
Manipulating Arrays with push()  
push() adds an element to the end of an array.  
Example: myArray.push(["Hope", 36]);  
  
Manipulating Arrays with pop()  
pop() removes the last element from an array.  
Example: var removedFromArray = myArray.pop();  
  
Manipulating Arrays with shift()  
shift() removes the first element from an array.  
Example: var removedFromArray = myArray.shift();  
  
Manipulating Arrays with unshift()  
unshift() adds an element to the beginning of an array.  
Example: myArray.unshift(["Liam", 18]);

## 14. Functions

Functions allow you to write reusable blocks of code.  
Example:  
function myReusableCode() {  
 console.log("Heyya, what are you doing");  
}  
myReusableCode();  
  
Function Arguments  
Functions can take inputs (parameters) called arguments.  
Example:  
function myCodeWithArgs(a, b) { console.log(a + b); }  
myCodeWithArgs(10, 5); // 15

## 15. Scope (Global vs Local)

• Global Scope: Variables declared outside of a function are global. If var is used outside a function, it is global.  
• Local Scope: Variables declared inside a function are local to that function only.  
• Undefined Value: A function that doesn’t return anything will return undefined by default.

## 16. Return a Value from a Function with return

function minusSeven(num) {  
 return num - 7;  
}  
  
console.log(minusSeven(10));  
// Output: 3  
  
function timesFive(num) {  
 return num \* 5;  
}  
  
console.log(timesFive(3));  
// Output: 15

17. Undefined Value Returned

function addSix() {  
 sum += 6;  
}  
// No return value → Answer would be undefined.

## 18. Assignment with a Returned Value

var changed = 0;  
  
function change(num) {  
 return (num + 5) / 3;  
}  
  
changed = change(10);  
// changed = (10 + 5) / 3 = 5

19. Stand in Line (Queue Example)

function nextInLine(arr, item) {  
 arr.push(item);  
 return arr.shift();  
}  
  
let testArr = [1, 2, 3, 4, 5];  
  
console.log("Before: " + JSON.stringify(testArr));  
console.log(nextInLine(testArr, 6));  
console.log("After: " + JSON.stringify(testArr));  
  
// Before: [1, 2, 3, 4, 5]  
// Returned: 1  
// After: [2, 3, 4, 5, 6]

## 20. Boolean Values

function welcomeToBooleans() {  
 return true;  
}

21. Use Conditional Logic with if Statements

function trueOrFalse(wasThatTrue) {  
 if (wasThatTrue) {  
 return "Yes, that was true";  
 }  
 return "No, that was not true";  
}  
  
console.log(trueOrFalse(true));  
// Output: "Yes, that was true"

## 22. Comparison with the Equality Operator

function testEqual(val) {  
 if (val == 12) {  
 return "Equal";  
 }  
 return "Not Equal";  
}  
  
console.log(testEqual(12));  
// Output: "Equal"

## 23. Comparison with the Strict Equality Operator

// Strict equal sign is ===  
// Strict equality does not do type conversion.  
  
12 === 12 // true  
12 === "12" // false  
12 == "12" // true

## 24. Comparison with the Inequality Operator

function testNotEqual(val) {  
 if (val != 12) {  
 return "Not Equal";  
 }  
 return "Equal";  
}

## 25. Comparison with Greater Than Operator

function testGreaterThan(val) {  
 if (val > 7) {  
 return "Greater than 7";  
 }  
 if (val > 2) {  
 return "Greater than 2";  
 }  
 return "2 or less";  
}

## 26. Comparison with Equality

function testEqual(val) {  
 if (val == 12) { // == checks equality of values (with type conversion)  
 return "Equal";  
 }  
 return "Not Equal";  
}  
console.log(testEqual(10)); // "Not Equal" because 10 != 12

The == operator compares values, converting types if needed[[1]](https://toolsqa.com/javascript/equality-operator-javascript/#:~:text=It%20is%20type%20coercion%2C%20which,If%C2%A0both%20operands%20are).

## 27. Strict Equality

function testStrict(val) {  
 if (val === 7) { // === checks value \*and\* type (no conversion)  
 return "Equal";  
 }  
 return "Not Equal";  
}  
console.log(testStrict(10)); // "Not Equal" (10 is not strictly equal to 7)

The === operator is a strict (identity) equality: both value and type must match.

## 28. Comparison with Inequality

function testNotEqual(val) {  
 if (val != 99) { // != checks inequality of values (with type conversion)  
 return "Not Equal";  
 }  
 return "Equal";  
}  
console.log(testNotEqual(99)); // "Equal" because 99 == 99

The != operator is the loose inequality (opposite of ==) with type conversion.

## 29. Strict Inequality

function testStrictNotEqual(val) {  
 if (val !== 17) { // !== checks inequality without type conversion  
 return "Not Equal";  
 }  
 return "Equal";  
}  
console.log(testStrictNotEqual(17)); // "Equal" because 17 === 17

The !== operator is the strict (identity) inequality: no conversion, must differ in value or type0.

## 30. Greater Than & Less Than

function compareSize(a, b) {  
 if (a > b) { // greater than  
 return "a > b";  
 }  
 return "a <= b";  
}  
console.log(compareSize(5, 3)); // "a > b"

function compareRange(val) {  
 if (val >= 20) { // greater than or equal to  
 return "20 or Over";  
 }  
 if (val <= 19) { // less than or equal to  
 return "Under 20";  
 }  
}  
console.log(compareRange(10)); // "Under 20"

Standard numeric comparisons (>, >=, <, <=) work normally.

## 31. Logical AND (&&)

function logicalAnd(val) {  
 if (val >= 25 && val <= 50) { // both conditions must be true  
 return "Yes";  
 }  
 return "No";  
}  
console.log(logicalAnd(30)); // "Yes"

&& returns true if both sides are true.

## 32. Logical OR (||)

function logicalOr(val) {  
 if (val < 10 || val > 20) { // one or both conditions true  
 return "Outside";  
 }  
 return "Inside";  
}  
console.log(logicalOr(25)); // "Outside"

|| returns true if at least one side is true.

## 33. If Statements

if (true) {  
 console.log("Condition is true");  
}

An if executes its block when the condition is truthy.

## 34. If...Else

function testElse(val) {  
 if (val > 5) {  
 return "Over 5";  
 } else {  
 return "5 or Under";  
 }  
}  
console.log(testElse(4)); // "5 or Under"

else runs when the if condition is false.

## 35. Else If

function testElseIf(val) {  
 if (val > 10) {  
 return "Greater than 10";  
 } else if (val < 5) {  
 return "Less than 5";  
 } else {  
 return "Between 5 and 10";  
 }  
}  
console.log(testElseIf(7)); // "Between 5 and 10"

Chaining if, else if, and else handles multiple cases.

## 36. Golf Code (chained if/else)

function golfScore(par, strokes) {  
 if (strokes == 1) {  
 return "Hole-in-one!";  
 } else if (strokes <= par - 2) {  
 return "Eagle";  
 } else if (strokes == par - 1) {  
 return "Birdie";  
 } else if (strokes == par) {  
 return "Par";  
 } else if (strokes == par + 1) {  
 return "Bogey";  
 } else {  
 return "Double Bogey or worse";  
 }  
}  
console.log(golfScore(5, 4)); // "Birdie"

Multiple branches to handle scoring rules.

## 37. Switch Statements

function caseInSwitch(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(caseInSwitch(1)); // "alpha"

switch compares the expression to each case value. Don’t forget break.

## 38. Multiple Identical Options

function sequentialSizes(val) {  
 var answer = "";  
 switch(val) {  
 case 1:  
 case 2:  
 case 3:  
 answer = "Low";  
 break;  
 case 4:  
 case 5:  
 case 6:  
 answer = "Mid";  
 break;  
 case 7:  
 case 8:  
 case 9:  
 answer = "High";  
 break;  
 }  
 return answer;  
}  
console.log(sequentialSizes(2)); // "Low"

Cases can stack without code to group outcomes.

## 39. Replacing If/Else with Switch

function chainToSwitch(val) {  
 var answer = "";  
 switch(val) {  
 case "bob":  
 answer = "Marley";  
 break;  
 case 42:  
 answer = "The Answer";  
 break;  
 case 1:  
 answer = "There is no #1";  
 break;  
 case 99:  
 answer = "Missed me by this much!";  
 break;  
 case 7:  
 answer = "Ate Nine";  
 }  
 return answer;  
}

Functionally equivalent to many if/else statements.

## 40. Returning Boolean Values from Functions

function isLess(a, b) {  
 return a < b;  
}  
console.log(isLess(10, 15)); // true

Directly returning a comparison result.

## 41. Return Early Pattern

function abTest(a, b) {  
 if (a < 0 || b < 0) {  
 return undefined; // exit early on invalid input  
 }  
 // Rest of function...  
}

Early return avoids nested ifs.

## 42. Counting Cards (Switch Example)

var count = 0;  
function cc(card) {  
 switch(card) {  
 case 2:  
 case 3:  
 case 4:  
 case 5:  
 case 6:  
 count++;  
 break;  
 case 10:  
 case "J":  
 case "Q":  
 case "K":  
 case "A":  
 count--;  
 break;  
 }  
 var holdbet = 'Hold'; // default decision  
 if (count > 0) {  
 holdbet = 'Bet'; // bet if count positive  
 }  
 return count + " " + holdbet;  
}  
console.log(cc(2)); // "1 Bet"  
console.log(cc(10)); // "0 Hold"

Updates a running count and returns a string like "1 Bet". No ternary used here.

## 43. Build JavaScript Objects

var ourDog = {  
 "name": "Camper",  
 "legs": 4,  
 "tails": 1,  
 "friends": ["everything"]  
};  
var myDog = {  
 name: "Quincy",  
 legs: 3,  
 tails: 2,  
 friends: []  
}

Objects store key:value pairs. Keys can be quoted or unquoted strings.

## 44. Accessing Object Properties with Dot Notation

var testObj = {  
 "hat": "ballcap",  
 "shirt": "jersey",  
 "shoes": "cleats"  
};  
var hatValue = testObj.hat; // "ballcap"  
var shirtValue = testObj.shirt; // "jersey"

Use object.property to get values.

## 45. Accessing Object Properties with Bracket Notation

var testObj = {  
 "an entree": "hamburger",  
 "my side": "veggies",  
 "the drink": "water"  
};  
var entreeValue = testObj["an entree"]; // "hamburger"  
var drinkValue = testObj["the drink"]; // "water"

Use brackets and a string key. Needed when key has spaces.

## 46. Accessing Object Properties with Variables

var testObj = {  
 12: "Montana",  
 16: "Alaska",  
 19: "California"  
};  
var playerNumber = 16;  
var player = testObj[playerNumber]; // "Alaska"

Bracket notation can take a variable key name.

## 47. Updating Object Properties

var myDog = {  
 "name": "Coder",  
 "legs": 4,  
 "tails": 1,  
 "friends": []  
};  
myDog.name = "Happy Coder";

Just assign with dot (or bracket) to change a value.

## 48. Adding New Properties to an Object

var ourDog = {  
 "name": "Camper",  
 "legs": 4,  
 "tails": 1,  
 "friends": ["everything"]  
};  
ourDog.bark = "bow-wow";  
myDog["bark"] = "woof!";

Dot or bracket notation to create new keys.

## 49. Deleting Properties from an Object

var ourDog = {  
 "name": "Camper",  
 "bark": "bow-wow"  
};  
delete ourDog.bark; // removes the property  
delete myDog.tails;

Use delete object.property.

50. Using Objects for Lookups

var result = "";  
function phoneticLookup(val) {  
 var lookup = {  
 "alpha": "Adams",  
 "bravo": "Boston",  
 "charlie": "Chicago",  
 "delta": "Denver",  
 "echo": "Easy",  
 "foxtrot": "Frank"  
 };  
 result = lookup[val];  
 return result;  
}

Replace a switch with an object map. Lookup properties by key.

## 51. Testing Objects for Properties

function checkObj(checkProp) {  
 if (myObj.hasOwnProperty(checkProp)) {  
 return myObj[checkProp];  
 } else {  
 return "Not Found";  
 }  
}

Use object.hasOwnProperty(prop) to test for a key[[4]](https://ytscribe.com/th/v/PkZNo7MFNFg#:~:text=to%20create%20var%20lookup,And%20let%E2%80%99s).

52. Manipulating Complex Objects

var myMusic = [  
 {  
 "artist": "Billy Joel",  
 "title": "Piano Man",  
 "release\_year": 1973,  
 "formats": ["CD", "8T", "LP"],  
 "gold": true  
 },  
 {  
 "artist": "Beau Carnes",  
 "title": "Cereal Man",  
 "release\_year": 2003,  
 "formats": ["YouTube video"]  
 }  
];

An array of objects. Each object can hold varied data types (strings, numbers, arrays).

## 53. Accessing Nested Objects

var myStorage = {  
 "car": {  
 "inside": {  
 "glove box": "maps",  
 "passenger seat": "crumbs"  
 },  
 "outside": {  
 "trunk": "jack"  
 }  
 }  
};  
var gloveBoxContents = myStorage.car.inside["glove box"]; // "maps"

Chain dot and bracket for nested props.

## 54. Accessing Nested Arrays

var myPlants = [  
 {  
 type: "flowers",  
 list: ["rose", "tulip", "dandelion"]  
 },  
 {  
 type: "trees",  
 list: ["fir", "pine", "birch"]  
 }  
];  
var secondTree = myPlants[1].list[1]; // "pine"

Use multiple indexes and property access.

55. Record Collection (Challenge)

function updateRecords(id, prop, value) {  
 if (value === "") {  
 delete collection[id][prop];  
 } else if (prop === "tracks") {  
 // create array if it doesn't exist  
 collection[id][prop] = collection[id][prop] || [];  
 collection[id][prop].push(value);  
 } else {  
 collection[id][prop] = value;  
 }  
 return collection;  
}

Modify or delete nested object entries based on conditions.

## 56. Iterate with While Loops

var myArray = [];  
var i = 0;  
while (i < 5) {  
 myArray.push(i);  
 i++;  
}  
console.log(myArray); // [0, 1, 2, 3, 4]

Runs a block while the condition is true.

57. Iterate with For Loops

var myArray = [];  
for (var i = 0; i < 5; i++) {  
 myArray.push(i);  
}  
console.log(myArray); // [0, 1, 2, 3, 4]

Classic for loop with init, condition, final-expression.

## 58. Iterate Odd Numbers with a For Loop

var myArray = [];  
for (var i = 1; i < 10; i += 2) {  
 myArray.push(i);  
}  
console.log(myArray); // [1, 3, 5, 7, 9]

Change the increment (here i += 2) to skip numbers.

59. Count Backwards with a For Loop

var myArray = [];  
for (var i = 10; i > 0; i -= 2) {  
 myArray.push(i);  
}  
console.log(myArray); // [10, 8, 6, 4, 2]

Initialize at a higher number and decrement.

## 60. Iterate Through an Array with a For Loop

var ourArr = [9, 10, 11, 12];  
var total = 0;  
for (var i = 0; i < ourArr.length; i++) {  
 total += ourArr[i];  
}  
console.log(total); // 42

Use array.length to loop over all elements.

## 61. Nesting For Loops

function multiplyAll(arr) {  
 var product = 1;  
 for (var i = 0; i < arr.length; i++) {  
 for (var j = 0; j < arr[i].length; j++) {  
 product \*= arr[i][j];  
 }  
 }  
 return product;  
}  
console.log(multiplyAll([[1,2],[3,4],[5,6,7]])); // 5040

## Double loop for 2D arrays.

62. Iterate with Do...While Loops

var myArray = [];  
var i = 10;  
do {  
 myArray.push(i);  
 i++;  
} while (i < 5);  
console.log(i, myArray); // 11 [10]

do...while runs the block once before checking the condition.

## 63. Profile Lookup (Challenge)

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

Loop through array of objects to match a name and return a property, or an error if missing.

## 64. Generate Random Fractions

function randomFraction() {  
 return Math.random();  
}  
console.log(randomFraction()); // e.g. 0.745

Math.random() returns a pseudo-random float in [0,1).

## 65. Generate Random Whole Numbers

function randomWholeNum() {  
 return Math.floor(Math.random() \* 10);  
}  
console.log(randomWholeNum()); // 0-9 inclusive

Multiply by range and use Math.floor to get a whole number.

## 66. Generate Random Whole Numbers within a Range

function randomRange(myMin, myMax) {  
 return Math.floor(Math.random() \* (myMax - myMin + 1)) + myMin;  
}  
console.log(randomRange(5, 15)); // between 5 and 15 inclusive

Formula for random integer between myMin and myMax, inclusive.

## 67. Use the parseInt Function

function convertToInteger(str) {  
 return parseInt(str);  
}  
console.log(convertToInteger("56")); // 56

parseInt() converts a string to an integer.

## 68. Use parseInt with a Radix

function convertToInteger(str, radix) {  
 return parseInt(str, radix);  
}  
console.log(convertToInteger("10111", 2)); // 23 (binary to decimal)

Second argument radix specifies base (e.g., 2 for binary).

69. Use the Conditional (Ternary) Operator

function checkEqual(a, b) {  
 return a === b ? true : false;  
}

function checkSign(num) {  
 return num > 0 ? "positive" : num < 0 ? "negative" : "zero";  
}

The ternary condition ? exprIfTrue : exprIfFalse is a concise if...else.

## 70. Differences Between var and let

"use strict";  
var catName = "Quincy";  
var catName = "Beau"; // no error  
let dogName = "Spot";  
let dogName = "Hunter"; // Error: cannot redeclare

let is block-scoped and cannot be re-declared in the same scope.

## 71. Compare Scopes of var and let

function checkScope() {  
 let i = "function scope";  
 if (true) {  
 let i = "block scope";  
 console.log("Block scope i is: " + i);  
 }  
 console.log("Function scope i is: " + i);  
 return i;  
}

var is function-scoped; let/const are block-scoped.

## 72. Declare a Read-Only Variable with const

const SENTENCE = "freeCodeCamp is cool!";  
let sentence = SENTENCE;  
console.log(sentence);

const declares a constant that cannot be reassigned.

## 73. Mutate an Array Declared with const

const s = [5, 7, 2];  
s[0] = 2;  
s[1] = 5;  
s[2] = 7;  
console.log(s); // [2, 5, 7]

You cannot reassign the array itself, but you can change its elements.

## 74. Prevent Object Mutation (Object.freeze)

function freezeObj() {  
 const MATH\_CONSTANTS = {  
 PI: 3.14  
 };  
 Object.freeze(MATH\_CONSTANTS);  
 try {  
 MATH\_CONSTANTS.PI = 99;  
 } catch(err) {  
 return err;  
 }  
 return MATH\_CONSTANTS.PI;  
}  
const PI = freezeObj();

Object.freeze(obj) makes an object’s properties immutable.

## 75. Write Concise Anonymous Functions with Arrow Syntax

// Traditional  
var magic = function() {  
 return new Date();  
};  
// Arrow function  
const magic = () => new Date();

Arrow functions are a compact alternative to function expressions[[14]](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#:~:text=An%20arrow%20function%20expression%20is,and%20deliberate%20limitations%20in%20usage).

## 76. Write Arrow Functions with Parameters

// Traditional  
var myConcat = function(arr1, arr2) {  
 return arr1.concat(arr2);  
};  
// Arrow function  
const myConcat = (arr1, arr2) => arr1.concat(arr2);

Omit function keyword and, for single expressions, braces and return (implied).

77. Write Higher-Order Arrow Functions (Filter & Map)

const squareList = arr => {  
 return arr  
 .filter(num => Number.isInteger(num) && num > 0)  
 .map(x => x \* x);  
};  
console.log(squareList([-3, 4.8, 5, 3, -3])); // [16, 25, 9]

Pass arrow functions to filter and map for concise operations.

## 78. Use Default Parameters

function increment(number, value = 1) {  
 return number + value;  
}  
console.log(increment(4, 5)); // 9  
console.log(increment(4)); // 5 (uses default value 1)

value=1 sets default when not provided.

## 79. Use the Rest Operator with Function Parameters

function sum(...args) {  
 return args.reduce((a, b) => a + b, 0);  
}  
console.log(sum(1, 2, 3, 4)); // 10

...args collects any number of arguments into an array.

## 80. Use the Spread Operator to Evaluate Arrays In-Place

const arr1 = ['JAN', 'FEB', 'MAR'];  
let arr2;  
(function() {  
 arr2 = [...arr1]; // spread into new array  
 arr1[0] = 'potato';  
})();  
console.log(arr2[0]); // 'JAN'

... in an array literal expands another array’s elements.

## 81. Destructuring Assignment from Objects

const voxel = {x: 3.6, y: 7.4, z: 6.54};  
const {x: a, y: b, z: c} = voxel;

Unpack object properties into variables. Here x→a, etc.

82. Destructuring Assignment with Nested Objects

const LOCAL\_FORECAST = {  
 today: { min: 72, max: 83 },  
 tomorrow: { min: 73.3, max: 84.6 }  
};  
const { tomorrow: { max: maxOfTomorrow } } = LOCAL\_FORECAST;  
console.log(maxOfTomorrow); // 84.6

Use matching object structure on left-hand side to unpack nested values.

## 83. Destructuring Assignment from Arrays

const [z, x, , y] = [1, 2, 3, 4, 5, 6];  
console.log(z, x, y); // 1 2 4

Assign array elements to variables. Extra commas skip elements.

## 84. Use Destructuring Assignment with the Rest Operator

function removeFirstTwo(list) {  
 const [ , , ...rest] = list;  
 return rest;  
}  
const source = [1,2,3,4,5,6,7,8,9,10];  
console.log(removeFirstTwo(source)); // [3,4,5,6,7,8,9,10]

Combine array destructuring with ... to gather remaining elements into an array.

## 85. Use Destructuring to Pass an Object as a Function’s Parameters

const stats = {  
 max: 56.78,  
 min: -0.75,  
 // ...  
};  
const half = ({ max, min }) => (max + min) / 2.0;  
console.log(half(stats)); // 28.015

In function signature, { max, min } unpacks only needed props.

## 86. Create Strings using Template Literals

const person = { name: "Zodiac Hasbro", age: 56 };  
const greeting = `Hello, my name is ${person.name}!   
I am ${person.age} years old.`;

Enclose string in backticks and embed variables with ${...}[[15]](https://ytscribe.com/th/v/PkZNo7MFNFg#:~:text=destructured%20into%20just%20the%20max,max).

## 87. Write Concise Object Literal Declarations Using Simple Fields

const createPerson = (name, age, gender) => ({ name, age, gender });  
console.log(createPerson("Zodiac Hasbro", 56, "male"));

If property name and variable name are same, just list name once.

## 88. Write Concise Declarative Functions in Objects

const bicycle = {  
 gear: 2,  
 setGear(newGear) { // shorthand method  
 this.gear = newGear;  
 }  
};  
bicycle.setGear(3);  
console.log(bicycle.gear); // 3

ES6 allows omitting function: in method definitions.

## 89. Use Class Syntax to Define a Constructor Function

class SpaceShuttle {  
 constructor(targetPlanet) {  
 this.targetPlanet = targetPlanet;  
 }  
}  
const zeus = new SpaceShuttle('Jupiter');  
console.log(zeus.targetPlanet); // "Jupiter"

class and constructor replace the old function + new pattern.

## 90. Class Example (Vegetable)

class Vegetable {  
 constructor(name) {  
 this.name = name;  
 }  
}  
const carrot = new Vegetable('carrot');  
console.log(carrot.name); // "carrot"

Define class properties in the constructor; use new ClassName() to instantiate.

## 91. Use Getters and Setters to Control Access to an Object

class Thermostat {  
 constructor(fahrenheit) {  
 this.\_temp = 5/9 \* (fahrenheit - 32);  
 }  
 get temperature() {  
 return this.\_temp;  
 }  
 set temperature(updatedTemp) {  
 this.\_temp = updatedTemp;  
 }  
}  
const thermos = new Thermostat(76);  
let temp = thermos.temperature; // getter called  
thermos.temperature = 26; // setter called  
temp = thermos.temperature;  
console.log(temp); // 26

Getters/setters allow controlled access. Access without calling a function (no () after temperature).

## 92. Differences Between import and require

Modern JS uses import/export modules instead of require.

// In capitalize\_strings.js (module file)  
export const capitalizeString = str => str.toUpperCase();  
  
  
import { capitalizeString } from "./capitalize\_strings";  
console.log(capitalizeString("hello")); // "HELLO!"

export makes functions/vars available, import brings them into another file.

## 93. Use export to Reuse a Code Block

// capitalize\_strings.js  
export function capitalizeString(str) {  
 return str.toUpperCase();  
}  
export const foo = "bar";  
export const bar = "foo";  
  
  
import { capitalizeString, foo, bar } from "./capitalize\_strings";

Export multiple named items to import them elsewhere[[16]](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#:~:text=The%20,visible%20in%20its%20imported%20value).

## 94. Use \* to Import Everything from a File

import \* as capitalizeStrings from "./capitalize\_strings";  
// Now capitalizeStrings.capitalizeString, .foo, etc. are available.

Imports all exports as properties of an object.

95. Create an Export Fallback with export default

// math\_functions.js  
export default function subtract(x, y) {  
 return x - y;  
}  
  
  
import subtract from "./math\_functions";  
console.log(subtract(7, 4)); // 3

export default is a module’s default export. Imported without braces.