JavaScript Overview

1. What is JavaScript and where is it mostly used?

JavaScript (JS) is a high-level, interpreted scripting language mainly used to make web pages interactive and dynamic.

It is mostly used in:

- Web development (form validation, dynamic content, animations)

- Backend development (Node.js)

- Mobile application development

- Desktop applications

- Games, real-time applications, and IoT

HTML provides structure, CSS provides style, and JavaScript provides behavior.

2. Is JavaScript standardized?

Yes, JavaScript is standardized.

It follows the ECMAScript standard maintained by ECMA International.

This ensures consistent behavior across different browsers and platforms.

3. Latest JavaScript specification name

The latest JavaScript specification is ECMAScript 2024 (ES15).

ECMAScript is updated every year with new features and improvements.

4. What is a JavaScript Engine?

A JavaScript engine is a program that executes JavaScript code.

It parses, compiles, and runs JavaScript programs on a device or browser.

5. Popular JavaScript Engines and their usage

- V8 Engine: Developed by Google, used in Google Chrome and Node.js

- SpiderMonkey: Developed by Mozilla, used in Mozilla Firefox

- JavaScriptCore: Developed by Apple, used in Safari browser

- Chakra: Developed by Microsoft, used in legacy versions of Microsoft Edge

Summary:

JavaScript is a standardized scripting language (ECMAScript) mainly used for web development and executed using JavaScript engines like V8 and SpiderMonkey.

JavaScript – Execution Context & Hoisting

6. What is an Execution Context in JavaScript?

An Execution Context is an abstract environment in which JavaScript code is evaluated and executed.

Whenever JavaScript code runs, it is executed inside an execution context.

Each execution context contains three main components:

1. Variable Environment – stores variables, function declarations, and arguments.

2. Scope Chain – provides access to variables of outer scopes.

3. This Binding – determines the value of the 'this' keyword.

Execution contexts are created in two phases:

• Creation Phase – memory allocation for variables and functions.

• Execution Phase – code execution line by line.

Thus, execution context controls how code is executed and how variables and functions are accessed.

7. How many types of Execution Context are there in JavaScript?

There are three types of execution contexts in JavaScript:

1. Global Execution Context (GEC)

   - Created when the JavaScript program starts.
   - Only one global execution context exists.
   - Global variables and functions are stored here.
   - 'this' refers to the global object.

2. Function Execution Context (FEC)

   - Created whenever a function is invoked.
   - Each function call creates a new execution context.
   - Stored in the call stack.

3. Eval Execution Context

   - Created when code is executed inside eval().
   - Rarely used and generally avoided.

Among these, the Global and Function Execution Contexts are most commonly used.

8. What is Hoisting?

Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top

of their scope during the compilation phase before code execution.

Important points:

• Only declarations are hoisted, not initializations.

• Hoisting happens during the creation phase of execution context.

Example:

console.log(x); // undefined

var x = 10; //Here, the declaration 'var x' is hoisted, but initialization happens later.

Hoisting allows functions and variables to be accessed before they are defined in the code.

9. What are the different types of Hoisting?

There are two main types of hoisting in JavaScript:

1. Variable Hoisting
   - Variables declared using 'var' are hoisted and initialized as undefined.
   - Variables declared using 'let' and 'const' are hoisted but not initialized, leading to the Temporal Dead Zone (TDZ).
2. Function Hoisting
   - Function declarations are fully hoisted.
   - Function expressions are hoisted only as variables.

Thus, hoisting behavior differs based on how variables and functions are declared.

10. Explain how functions are hoisted.

Function hoisting depends on how a function is defined:

1. Function Declarations:

Function declarations are completely hoisted along with their definitions.

They can be called before they are defined in the program.

Example:

```
sayHello();

function sayHello() {

  console.log("Hello");

}
```

This works because the entire function is hoisted.

2. Function Expressions:

Function expressions are hoisted like variables.

Only the variable name is hoisted, not the function body.

Example:

```
sayHi(); // Error

var sayHi = function () {

  console.log("Hi");

};
```

Here, 'sayHi' is undefined during hoisting, so calling it causes an error.

Thus, function declarations support full hoisting, while function expressions do not.

JavaScript – Call Stack, Event Loop & Concurrency

11. What is Call Stack?

The Call Stack is a data structure used by the JavaScript engine to keep track of function calls

during program execution. It follows the Last In First Out (LIFO) principle.

Whenever a function is called, it is pushed onto the call stack. When the function finishes

execution, it is popped out of the stack.

Key points:

• Manages execution order of functions

• Works on LIFO principle

• Each function call creates a stack frame

• Stack overflow occurs if the stack exceeds its limit

Thus, the call stack ensures that JavaScript executes functions in the correct order.

12. Explain how JavaScript uses a Call Stack.

JavaScript uses the call stack to track the execution of functions step by step.

Working:

1. When the program starts, the Global Execution Context is pushed onto the stack.
2. When a function is called, its Function Execution Context is pushed on top of the stack.
3. JavaScript executes the function at the top of the stack.
4. After execution, the function context is popped from the stack.
5. Control returns to the previous execution context.

Example:

```
function first() {
  second();
}
function second() {
  console.log("Hello");
}
first();
```

Execution order:

- Global Context
- first()
- second()

Thus, JavaScript ensures synchronous execution using the call stack.

13. Is JavaScript multi-threaded?

No, JavaScript is not multi-threaded. It is a single-threaded language.

This means:

• JavaScript executes one task at a time

• Only one call stack exists

• Tasks are executed sequentially

However, JavaScript appears asynchronous due to:

• Event loop

• Web APIs

• Callback queue

These features allow JavaScript to handle non-blocking operations efficiently

without using multiple threads.

14. What is an Event Loop?

The Event Loop is a mechanism in JavaScript that allows asynchronous operations to be executed

without blocking the main thread.

Its main job is to:

• Continuously monitor the call stack

• Check the callback queue

• Push waiting callbacks into the call stack when it becomes empty

The event loop enables JavaScript to handle:

• Timers (setTimeout)

• Promises

• DOM events

• API calls. Thus, the event loop makes JavaScript non-blocking and responsive.

15. What is Callback Queue?

The Callback Queue is a data structure that stores callback functions waiting to be executed after asynchronous operations are completed.

Examples of async operations:

• setTimeout()

• setInterval()

• DOM events

• Web API responses

Working:

• Async tasks are handled by Web APIs

• Once completed, their callbacks are placed in the callback queue

• The event loop moves callbacks to the call stack when it is empty

The callback queue works with the event loop to support asynchronous behavior in JavaScript.

JavaScript – Data Types & Primitives

16. What data types do JavaScript provide?

JavaScript provides two broad categories of data types:

1. Primitive Data Types

  - Number

  - String

  - Boolean

  - Undefined

  - Null

  - Symbol and   - BigInt

2. Non-Primitive (Reference) Data Types

- Object
- Array
- Function
- Date, Map, Set, etc.

Primitive data types store simple values and are immutable, whereas non-primitive data types

store complex data and are mutable.

JavaScript is a dynamically typed language, meaning the data type of a variable is determined at runtime.

17. How to find out the data type of value?

The data type of value in JavaScript can be determined using the typeof operator.

Syntax:

typeof value

Example:

typeof 10   → "number"

typeof "Hello"   → "string"

typeof true   → "boolean"

Special cases:

typeof null   → "object" (known JavaScript bug)

typeof undefined → "undefined"

typeof []   → "object"

Thus, typeof is the primary way to identify the data type of value in JavaScript.

18. Which operator is used to get the type of value?

The typeof operator is used to get the type of a value in JavaScript.

Key points:

• Returns the data type as a string

• Works with all primitive data types

• Commonly used for type checking and debugging

Example:

```
typeof 5         → "number"
typeof "JS"      → "string"
typeof function(){} → "function"
```

Hence, typeof is the standard operator to identify data types in JavaScript.

19. What is a primitive in JavaScript?

A primitive in JavaScript is a basic data type that represents a single, simple value.

Characteristics of primitives:

• Immutable (cannot be changed)

• Stored by value

• Do not have methods or properties (temporarily wrapped when used)

Example:

```
let a = 10;
let b = a;
b = 20;
```

Here, 'a' remains unchanged because primitives are copied by value.

Thus, primitives represent fundamental data values in JavaScript.

20. What are the different types of primitives?

JavaScript has seven primitive data types:

1. Number – represents integers and floating-point numbers
2. String – represents textual data
3. Boolean – represents true or false
4. Undefined – represents an uninitialized variable
5. Null – represents intentional absence of a value
6. Symbol – represents unique identifiers
7. BigInt – represents large integers beyond Number limits

These primitives form the foundation of data handling in JavaScript.

JavaScript – Primitives, Wrapper Objects & Type Coercion (8-Mark Answers)

21. What happens in the background when we try to look up methods on a primitive?

In JavaScript, primitive values such as numbers, strings, and booleans do not have methods or properties by themselves.

However, JavaScript allows method access on primitives due to a process called temporary object wrapping.

Background process:

1. When a method is accessed on a primitive, JavaScript temporarily converts the primitive into its corresponding wrapper object.
2. The method is executed on this temporary object.
3. After execution, the temporary object is destroyed.

Example:

```
let str = "Hello";

str.toUpperCase();
```

Here, "Hello" is temporarily wrapped into a String object, the toUpperCase() method is executed, and then the object is discarded.

This mechanism allows primitives to behave like objects without losing their immutability and performance benefits.

22. What are the various wrapper objects JavaScript provides out of the box?

JavaScript provides built-in wrapper objects for primitive data types.

These wrapper objects allow primitives to access methods and properties.

Common wrapper objects:

1. Number – wrapper for number primitive

2. String – wrapper for string primitive

3. Boolean – wrapper for boolean primitive

4. BigInt – wrapper for bigint primitive

5. Symbol – wrapper for symbol primitive

Example:

let num = 10;

num.toFixed(2);

Here, num is temporarily wrapped into a Number object to access the toFixed() method.

Wrapper objects are automatically created and destroyed by JavaScript and should not be created explicitly using the new keyword.

23. What is Type Coercion? Give some examples.

Type coercion is the automatic or implicit conversion of values from one data type to another by JavaScript.

JavaScript performs type coercion to evaluate expressions involving different data types.

Examples:

1. "5" + 2 → "52" (number converted to string)

2. "5" - 2 → 3 (string converted to number)

3. true + 1 → 2 (true converted to 1)

4. false == 0 → true (boolean converted to number)

Types of coercion:

• Implicit coercion – performed automatically by JavaScript

• Explicit coercion – done manually using functions like Number (), String(), Boolean()

Type coercion makes JavaScript flexible but can also lead to unexpected results if not handled carefully.

24. Any significant difference between 'undefined' and 'null'?

Yes, there are significant differences between undefined and null in JavaScript.

Undefined:

• Means a variable is declared but not assigned a value

• Default value of uninitialized variables

• Type is undefined

Null:

• Represents intentional absence of value

• Assigned explicitly by the programmer

• Type is object (due to a known JavaScript bug)

Comparison:

undefined == null → true

undefined === null → false

Thus, undefined indicates lack of assignment, while null indicates deliberate absence of a value.

JavaScript – Boolean, Number & Comparison Operators

BOOLEAN

25. What Boolean literals do JavaScript provide?

JavaScript provides two Boolean literals:

• true

• false

These literals represent logical truth values and are mainly used in conditions, loops, and decision-making statements.

26. What all in JavaScript are falsy?

The following values are considered falsy in JavaScript:

• false

• 0, -0

• "" (empty string)

• null

• undefined

• NaN

All other values are considered truthy.

27. What are different ways to convert any value to a boolean?

Values can be converted to boolean using:

• Boolean(value)

• Double NOT operator (!!value)

• Conditional statements (if, while)

These methods explicitly or implicitly convert values to true or false.

28. How does JavaScript store numbers in memory?

JavaScript stores numbers using the IEEE 754 double-precision 64-bit floating-point format.

This allows representation of integers and decimals but can cause precision issues.

Total Bits	64 bits
Sign Bit	1 Bits
Exponential bit	11 Bits
Mantissa bit	52 bits (+1 hidden)
Bias	1023
Range	$\approx \pm 2.2 \times 10^{-308}$ ≈±2.2×10−308 to $\pm 1.8 \times 10^{308}$ ±1.8×10308

29. What are the different ways to convert a value to a number?

• Number(value)

• parseInt(value)

• parseFloat(value)

• Unary plus (+value)

• Math functions

30. Which base of numbers does JavaScript support?

JavaScript supports:

• Binary (base 2)

• Octal (base 8)

• Decimal (base 10)

• Hexadecimal (base 16)

31. How to write binary numbers?

Binary numbers are written with prefix 0b.

Example: 0b1010

32. How to write octal numbers?

Octal numbers are written with prefix 0o.

Example: 0o755

33. How to write hexadecimal numbers?

Hexadecimal numbers are written with prefix 0x.

Example: 0xFF

34. How to convert a decimal into binary / octal / hexadecimal?

Use the toString() method with base.

Example: (10).toString(2)

35. How to convert a binary / octal / hexadecimal into decimal?

Use parseInt(value, base).

Example: parseInt("1010", 2)

36. How does the parseInt function work?

parseInt converts a string to an integer based on the specified radix.

It reads characters until an invalid character is encountered.

37. What is a numeric separator? Why is it used? Numeric separator (_) improves readability of large numbers. Example: 1_000_000 It does not affect value.

38. What is NaN?

NaN stands for Not-a-Number. It represents an invalid numeric result.

39. Which operations may result in NaN?

- 0 / 0
- Math.sqrt(-1)
- parseInt("abc")

40. How to check if a value is NaN?

- Number.isNaN(value)
- isNaN(value)

41. What does NaN === NaN return?

It returns false because NaN is not equal to anything, including itself.

42. How does JavaScript handle divide by zero?

- Positive number / 0 → Infinity
- Negative number / 0 → -Infinity
- 0 / 0 → NaN

43. What are MAX_SAFE_INTEGER and MIN_SAFE_INTEGER?

MAX_SAFE_INTEGER = 2^53 - 1

MIN_SAFE_INTEGER = -(2^53 - 1)

They represent the safe integer limits.

Infinity and -Infinity represent unbounded values.

44. What comparison operators does JavaScript have?

- ==, ===
- !=, !==
- <, <=, >, >=

45. What type of result do all comparison operators return?

All comparison operators return a boolean value (true or false).

46. How does the abstract equality operator (==) operate?

The == operator compares values after performing type coercion.

47. When will type coercion occur for ==?

Type coercion occurs when operands are of different types.

48. What is the difference between == and ===?

== compares values with type coercion.

=== compares both value and type (strict equality).

49. What are the expected data type of operands for <, <=, > and >=?

Operands are expected to be numbers or strings.

50. For string inputs, how do <, <=, > and >= operate?

Strings are compared lexicographically based on Unicode values.

51. When will type coercion occur for <, <=, > and >=?

Type coercion occurs when operands are not of the same type.