1. What is JavaScript?

JavaScript is a **programming language used for creating dynamic content on websites**. It is a **lightweight**, **cross-platform** and **single-threaded** programming language. JavaScript is an **interpreted** language that executes code line by line providing more flexibility.

---

2. Why Use JavaScript?

JavaScript is essential for the following reasons:

- Client-Side Interactivity : Enables user interactivity without requiring server round trips (e.g., form validation, animations).
- Cross-Platform : Works on all modern browsers and platforms.
- Asynchronous Operations : Allows asynchronous communication via APIs (AJAX, Fetch API).
- Large Ecosystem : Rich set of libraries and frameworks (e.g., React, Angular, Node.js) that simplify development.
- Versatility : Can be used on both client-side (browser) and server-side (Node.js).

----

3. What Are Its Features?

1. Works on Client-Side and Server-Side : JavaScript can be executed in the browser (client-side) or on the server (with Node.js).

2. Dynamically Typed : You don't need to declare variable types explicitly. Variables can hold any type of data and can change type at runtime.

3. Supports Functional and Object-Oriented Programming (OOP) : JavaScript allows you to write in both styles, enabling flexibility.

4. Makes HTML into DHTML (Dynamic HTML) : JavaScript can modify HTML and CSS on the fly, creating dynamic, real-time updates on web pages.

5. Synchronous : JavaScript executes code line by line, but can use asynchronous functions to manage non-blocking code (like API calls).

6. Single-Threaded : JavaScript runs on a single thread, meaning it processes one task at a time. Asynchronous programming allows handling multiple tasks efficiently.

---

4. How to Use JavaScript?

To use JavaScript:

- In a Web Browser : Include JavaScript in HTML by using `<script>` tags.

```
<script>
  console.log('Hello, JavaScript!');
</script>
```

- External File : Save JavaScript code in a `.js` file and link it to HTML.

```
<script src="script.js"></script>
```

- On the Server : Use Node.js to run JavaScript code server-side.

  ```bash

```
node app.js
```

  ```

---

5. Primitive Data Types in JavaScript

Primitive types in JavaScript are:

- String : Represents textual data (`'Hello'`).
- Number : Represents numeric data (`42`, `3.14`).
- BigInt : Represents large integers.
- Boolean : Represents `true` or `false`.
- undefined : Represents a variable that has been declared but not assigned a value.
- null : Represents an intentional absence of any object value.
- Symbol : Used to create unique identifiers.

---

6. Difference Between `null` and `undefined`

1. `undefined` :
   - Automatically assigned to a variable that is declared but not yet assigned a value.
   - Type of `undefined` is `undefined`.

2. `null` :
   - Represents the absence of any value or object.
   - Type of `null` is `object` (a known JavaScript quirk).

Key Differences :

- `undefined` is the default value for uninitialized variables, while `null` is an explicit assignment.
- `undefined` is more about a variable's state; `null` is about an intentional absence of value.

---

7. `typeof` Method

The `typeof` operator is used to check the type of a variable or value. It returns a string representing the type.

```javascript
typeof 42; // "number"
typeof "Hello"; // "string"
typeof true; // "boolean"
typeof undefined; // "undefined"
typeof null; // "object" (quirk)
typeof function() {}; // "function"
```

---

8. Scope Chaining in JavaScript

Scope chaining occurs when JavaScript searches for a variable in the current scope, and if not found, it moves up to the outer scope, continuing the search until it reaches the global scope.

Example:

```
let a = 10;
function outer() {
  let b = 20;
  function inner() {
    let c = 30;
    console.log(a, b, c); // a and b are found in outer scope, c in the current scope
  }
  inner();
}
outer();
```

---

9. Operators in JavaScript

Operators in JavaScript are symbols or keywords that perform operations on values:

- Arithmetic Operators : `+`, `-`, `*`, `/`, `%`, `++`, `--`
- Assignment Operators : `=`, `+=`, `-=`, `*=`, `/=`
- Comparison Operators : `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`
- Logical Operators : `&&`, `||`, `!`
- Ternary Operator : `condition ? true : false`
- Bitwise Operators : `&`, `|`, `^`, `<<`, `>>`

---

10. Array Object and Most Common Methods

Arrays in JavaScript are used to store multiple values in a single variable. They are objects with numeric indices.

Common array methods:

- `push()` : Adds an item to the end.

- `pop()` : Removes an item from the end.

- `shift()` : Removes an item from the start.

- `unshift()` : Adds an item to the start.

- `map()` : Creates a new array with the results of calling a function on every element.

- `filter()` : Creates a new array with elements that pass a test.

- `forEach()` : Iterates through an array for side effects (does not return a new array).

---

11. Difference Between `for...of` and `for...in` Loop

1. `for...of` :

  - Iterates over values of iterable objects like arrays, strings, or maps.

  - Example:

```
for (let item of [1, 2, 3]) {
  console.log(item); // Output: 1, 2, 3
}
```

  ```

2. `for...in` :

  - Iterates over keys/indexes of an object or array.

  - Example:

```
let obj = {a: 1, b: 2};
```
```

```
  for (let key in obj) {
    console.log(key); // Output: a, b
  }
```

  ```

---

**12. `filter` vs `map` Method**

- `filter()` : Returns a new array with elements that satisfy the provided condition.

```
let arr = [1, 2, 3, 4];
let result = arr.filter(x => x > 2); // [3, 4]
```

- `map()` : Returns a new array by applying a function to each element.

```
let arr = [1, 2, 3, 4];
let result = arr.map(x => x * 2); // [2, 4, 6, 8]
```

Difference :

- `map()` transforms elements of the array; `filter()` selects elements based on a condition.

---

13. What is an Object in JavaScript? Why and How Use It?

An object in JavaScript is a collection of key-value pairs. It is used to store and organize data, as well as represent entities in your application.

Example:

```
let person = {
  name: "Alice",
```
```

```
 age: 25,
 greet() {
   console.log("Hello " + this.name);
 }
};
```

---

14. Difference Between JavaScript Object and JSON Object

1. JavaScript Object :

  - A JavaScript object is a data structure with key-value pairs.

  - Can contain functions (methods).

2. JSON Object :

  - JSON (JavaScript Object Notation) is a text format for data exchange.

  - JSON is a string and cannot contain methods.

  - Example JSON:

```json
{
  "name": "Alice",
  "age": 25
}
```

---

15. What is JSON? Why and How Use It?

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is widely used to exchange data between a server and a client.

Use it for:

- Sending data via APIs.

- Storing data in a text-based format.

Example:

~~~
```json
{
  "name": "Alice",
  "age":


 25
}
```
~~~

---

16. `JSON.parse()` and `JSON.stringify()`

- `JSON.parse()` : Converts a JSON string into a JavaScript object.

```
  let jsonStr = '{"name": "Alice", "age": 25}';

  let obj = JSON.parse(jsonStr); // { name: "Alice", age: 25 }
```

- `JSON.stringify()` : Converts a JavaScript object into a JSON string.

```
  let obj = { name: "Alice", age: 25 };

  let jsonStr = JSON.stringify(obj); // '{"name": "Alice", "age": 25}'
```

---

17. Difference Between JSON Object and JSON Array

- JSON Object : An unordered collection of key-value pairs (like a JavaScript object).

```json
{ "name": "Alice", "age": 25 }
```

- JSON Array : An ordered list of values.

```json
[ "Alice", 25, "Engineer" ]
```

JSON objects are used for key-value mappings, while JSON arrays store an ordered list of items.

18. Difference between let, var, and const

var	let	const
The scope of a *var* variable is functional or global scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared in the same scope.	It can be updated but cannot be re-declared in the same scope.	It can neither be updated or re-declared in any scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.

var	let	const
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
These variables are hoisted.	These variables are hoisted but stay in the temporal dead zone untill the initialization.	These variables are hoisted but stays in the temporal dead zone until the initialization.

19. What is Hoisting?

Hoisting is a JavaScript mechanism where variables and function declarations are moved ("hoisted") to the top of their containing scope during the compile phase, before the code execution begins.

For var, only the declaration is hoisted, but not the initialization. This means the variable will exist, but its value will be undefined until the assignment line is reached.

```
console.log(a); // undefined

var a = 5;
```

For function declarations, the entire function definition is hoisted, so you can call the function before it's defined in the code.

```
myFunction(); // works, function is hoisted


function myFunction() {
  console.log("Hello!");
}
```

20. Hoisting in case of let and const

let and const are hoisted, but with a key difference: they are in a "temporal dead zone" from the start of the block until their actual declaration. This means you cannot access them before the declaration line; attempting to do so results in a ReferenceError.

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization

let a = 10;
```

In contrast, var is hoisted with an undefined value, so accessing it before the declaration won't throw an error, just return undefined.

```
console.log(a); // undefined

var a = 10;
```

21. What is a Function and Different Types of Functions

1. Function Declarations (Named Functions)

This is the most traditional way to define a function. A function declaration includes the function keyword, followed by a name, and the function body enclosed in curly braces.

```
function add(a, b) {

  return a + b;

}

console.log(add(3, 5)); // Output: 8
```

Hoisting: Function declarations are hoisted to the top of their scope, so you can call the function before it is defined in the code**.**

2. Function Expressions (Anonymous Functions)

A function expression is when a function is defined and assigned to a variable. These functions may or may not have a name. When no name is provided, the function is called an anonymous function.

```
const subtract = function(a, b) {

  return a - b;

};

console.log(subtract(5, 3)); // Output: 2
```

Not hoisted: Function expressions are not hoisted, so you can only call them after they have been defined.

3. Arrow Functions (ES6)

Arrow functions provide a more concise syntax for writing functions. They also have different behavior for the this keyword (they inherit this from the surrounding context).

```
const multiply = (a, b) => a * b;

console.log(multiply(4, 3)); // Output: 12
```

4. IIFE (Immediately Invoked Function Expression)

An IIFE is a function expression that is executed immediately after it is defined. It’s commonly used for creating a private scope.

```
(function() {

  console.log("This is an IIFE");

})();
```

5. Higher-Order Function

6. callback function

22. Hoisting in the Case of Function Statement and Function Expression

23. What is a Higher-Order Function?

A higher-order function is a function that either:

- Takes one or more functions as arguments.
- Returns a function as its result.

Examples of higher-order functions include map, filter, reduce, and setTimeout.

```
function greet(person) {

  return function(message) {

    console.log(`${message}, ${person}!`);

  };

}



const greetJohn = greet("John");

greetJohn("Hello"); // "Hello, John!"
```

In this example, greet is a higher-order function because it returns another function.

24. what is Callback Function

A callback function is a function that is passed as an argument to another function and is executed once that function completes its task. It's typically used to handle asynchronous operations like reading files, making HTTP requests, or dealing with user input.

```
<script>
  function operation(a,b,callback){
   if(typeof(a)!="number" || typeof(b)!="number")
    callback(new Error("invalid input[only numbers are allowed]"));
   else{
    let result = a + b;
    console.log("Addition : "+result);
    callback(null,result);
   }
  }
  operation(20,10,function(err,result){
    if(err)
     console.log(err);
    else
     result%2 ? console.log("Odd") : console.log("Even");
  });
</script>
```

 Error Callback (Error-First Callback): A callback function where the first argument is reserved for handling errors, allowing the calling function to handle both success and error conditions in a structured manner.

```
<script>
  function operation(a,b,callback){
    if(typeof(a)!="number" || typeof(b)!="number")
      callback(new Error("invalid input[only numbers are allowed]"));
    else{
      let result = a + b;
      console.log("Addition : "+result);
      callback(null,result);
    }
  }

  operation(20,10,function(err,result){
      if(err)
        console.log(err);
      else
        result%2 ? console.log("Odd") : console.log("Even");
  });
</script>
```

25 . What is **Synchronous and Asynchronous:**

Synchronous:

In synchronous programming, operations are executed one after another, in sequence. Each task must complete before the next task begins.

- **Blocking:** If a function is running, the program "waits" for it to finish before moving on to the next task.
- **Simple flow:** Easier to understand, but can cause delays if one task takes too long.

Example (Synchronous):

```
function firstTask() {
  console.log('First task completed');
}
function secondTask() {
  console.log('Second task completed');
}
firstTask();  // Runs first
secondTask(); // Runs second, only after firstTask is finished
```

Asynchronous:

Asynchronous programming allows tasks to run concurrently, meaning one task can start without waiting for the previous one to finish. Asynchronous code does not block the program’s execution.

- **Non-blocking:** The program can continue executing other tasks while waiting for the asynchronous task to complete.
- **Callbacks, Promises, and async/await** are common ways to handle asynchronous operations.

Example (Asynchronous):

```
function firstTask() {
  setTimeout(() => {
    console.log('First task completed');
  }, 2000);  // Delays execution by 2 seconds
}
function secondTask() {
  console.log('Second task completed');
}
firstTask();  // Starts first task, doesn't block the second task
secondTask(); // Runs immediately, even before first task completes
```

In this asynchronous example, the setTimeout simulates an asynchronous task (e.g., network request). The secondTask() runs without waiting for firstTask() to finish.

Key Differences:

- **Synchronous:** Executes one task at a time, blocking further operations until the current one is done.
- **Asynchronous:** Executes tasks independently of each other, allowing multiple tasks to be handled at once, making better use of system resources and reducing waiting times.