1. Basics of JavaScript

1. What is JavaScript? Difference between JavaScript and Java?

JavaScript (JS) is a high-level, just-in-time compiled, multi-paradigm programming language. It is best known as the scripting language for web pages, but it's used in many non-browser environments as well, such as server-side development (Node.js), mobile apps, and desktop apps.

Key Differences between JavaScript and Java:

Feature	JavaScript	Java
Typing	**Dynamic Typing**: Types are checked at runtime.	**Static Typing**: Types are checked at compile-time.
Execution	**Interpreted**: Executed by a JavaScript engine (e.g., V8) in the browser or Node.js.	**Compiled & Interpreted**: Compiled to bytecode, then run on a Java Virtual Machine (JVM).
Concurrency	**Single-threaded** with an event loop for non-blocking I/O.	**Multi-threaded**, allowing parallel execution of tasks.
OOP	**Prototype-based**: Objects inherit directly from other objects.	**Class-based**: Objects are instances of classes.
Use Case	Primarily for web development (client-side and server-side).	Primarily for enterprise-level applications, Android development, and large systems.

---

2. What are features of JavaScript?

- **High-Level Language**: Abstracts away machine details like memory management.
- **Dynamic Typing**: A variable can hold different data types at different times.
- **Multi-Paradigm**: Supports procedural, object-oriented (prototype-based), and functional programming styles.

- **Single-Threaded with Event Loop**: Handles asynchronous operations efficiently without blocking the main thread.
- **Interpreted / JIT-Compiled**: Code is executed by an engine without needing a separate compilation step for the developer.
- **Large Ecosystem**: Has a massive ecosystem of libraries and frameworks (npm, React, Angular, Vue, Node.js).

---

3. Difference between `var`, `let`, and `const`.

This is a fundamental question about variable scope and mutability.

Keyword	Scope	Hoisting	Re-declaration	Re-assignment
`var`	Function-scoped	Hoisted and initialized with `undefined`.	✅ Allowed	✅ Allowed
`let`	Block-scoped ({})	Hoisted but not initialized (TDZ).	❌ Not Allowed	✅ Allowed
`const`	Block-scoped ({})	Hoisted but not initialized (TDZ).	❌ Not Allowed	❌ Not Allowed*

*For `const`, you cannot reassign the variable itself, but if it holds an object or array, the contents of that object/array can be mutated.

Example:

```JavaScript
// var (function scope)
function varTest() {
  if (true) {
    var x = 10;
  }
  console.log(x); // 10 (x is visible outside the if block)
}

// let (block scope)
function letTest() {
  if (true) {
```

```
    let y = 20;
    console.log(y); // 20
  }
  // console.log(y); // ReferenceError: y is not defined
}

// const (block scope and immutable reference)
const z = 30;
// z = 40; // TypeError: Assignment to constant variable.

const person = { name: "Alex" };
person.name = "Bob"; // This is allowed!
console.log(person.name); // "Bob"
```

---

4. What are data types in JavaScript?

JavaScript has **eight** data types:

Primitive Types:

1. **Number**: For integers and floating-point numbers (`10`, `3.14`).
2. **String**: For text (`"hello"`, `'world'`).
3. **Boolean**: `true` or `false`.
4. **undefined**: A variable that has been declared but not assigned a value.
5. **null**: An intentional absence of any object value. It's a special value representing "nothing".
6. **BigInt**: For integers larger than the maximum safe integer in `Number`.
7. **Symbol**: A unique and immutable primitive value, often used as an object property key.

Non-Primitive Type:

8. Object: A collection of key-value pairs. Arrays, functions, and maps are all types of objects.

---

5. What is `typeof` operator?

The `typeof` operator returns a string indicating the type of the unevaluated operand.

Example:

JavaScript

```
console.log(typeof 42);          // "number"
console.log(typeof "hello");     // "string"
console.log(typeof true);        // "boolean"
console.log(typeof undefined);   // "undefined"
console.log(typeof { a: 1 });    // "object"
```

```
console.log(typeof [1, 2, 3]);    // "object" (Arrays are objects)
console.log(typeof function() {});// "function"
console.log(typeof null);         // "object" (This is a well-known historical bug)
```

---

6. What are truthy and falsy values?

In a boolean context (like an `if` statement), every value coerces to either `true` (truthy) or `false` (falsy).

Falsy Values (there are only 8):

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

Truthy Values: Any value that is not falsy. This includes `[]` (empty array), `{}` (empty object), `"0"`, `"false"`, and any non-zero number.

Example:

JavaScript

```
if ([]) {
  console.log("Empty array is truthy");
}
if ("") {
  // This block will not run
} else {
  console.log("Empty string is falsy");
}
```

---

7. Difference between `==` and `===` (loose vs strict equality).

- **`==` (Loose Equality)**: Compares two values for equality **after** performing type coercion if the types are different.
- **`===` (Strict Equality)**: Compares two values for equality **without** performing type coercion. It checks both the value and the type.

Best Practice: Always use `===` unless you have a specific reason to perform type coercion with `==`.

Example:

```
JavaScript

console.log(10 == "10");   // true (string "10" is coerced to number 10)
console.log(10 === "10");  // false (number vs string)

console.log(true == 1);    // true (boolean true is coerced to number 1)
console.log(true === 1);   // false (boolean vs number)

console.log(null == undefined);   // true
console.log(null === undefined);  // false
```

---

8. What is `NaN` in JavaScript? How to check for it?

`NaN` stands for **"Not-a-Number"**. It's a special numeric value that represents an invalid or unrepresentable number, often the result of a mathematical operation that failed.

A unique property of `NaN` is that it is not equal to anything, **including itself**.

- `NaN == NaN` is `false`.
- `NaN === NaN` is `false`.

To check for `NaN`, you must use the global `isNaN()` function or the more reliable `Number.isNaN()` method.

Example:

```
JavaScript

console.log(0 / 0); // NaN
console.log(Math.sqrt(-1)); // NaN

const result = 10 * "hello"; // NaN
console.log(result === NaN); // false!

// How to check correctly:
console.log(isNaN(result));          // true
console.log(Number.isNaN(result));   // true (preferred method, as it doesn't coerce types)
```

---

9. Difference between `null`, `undefined`, and undeclared variables.

- **`undefined`**: A variable has been declared, but no value has been assigned to it. It's the default value.
- **`null`**: An assignment value. It represents the intentional absence of any object value. It is explicitly assigned by a developer.
- **Undeclared**: A variable that has not been declared at all. Trying to access it will result in a `ReferenceError`.

Example:

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

let b = null;
console.log(b); // null

// console.log(c); // ReferenceError: c is not defined
```

---

10. What is scope in JavaScript? (Global, function, block).

Scope determines the accessibility of variables and functions at various parts of your code.

1. **Global Scope**: Variables declared outside of any function or block ({}) are in the global scope. They can be accessed from anywhere in the application.
2. **Function Scope**: Variables declared with `var` inside a function are only accessible within that function.
3. **Block Scope**: Variables declared with `let` and `const` inside a block (e.g., `if`, `for`, or just {}) are only accessible within that block.

Example:

```JavaScript
var globalVar = "I am global"; // Global scope

function myFunction() {
  var functionVar = "I am in a function"; // Function scope
  console.log(globalVar); // Accessible

  if (true) {
    let blockVar = "I am in a block"; // Block scope
    console.log(blockVar); // Accessible
  }
  // console.log(blockVar); // ReferenceError
}
myFunction();
// console.log(functionVar); // ReferenceError
```

---

2. Functions & Execution

11. What are function declarations vs function expressions?

- **Function Declaration**: A function defined with the function keyword. They are **hoisted**, meaning they are loaded before any code is executed and can be called before they are defined.
- JavaScript

```
sayHi(); // Works because of hoisting

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

-
-
- **Function Expression**: A function created and assigned to a variable. They are **not hoisted**. They cannot be called before the line where they are defined.
- JavaScript

```
// sayHello(); // TypeError: sayHello is not a function

const sayHello = function() {
  console.log("Hello!");
};

sayHello(); // Works
```

-
-

---

12. What are arrow functions?

Introduced in ES6, arrow functions provide a more concise syntax for writing functions. They also have a key difference in how they handle the this keyword.

Key Features:

- **Concise Syntax**: Shorter syntax, especially for simple, one-line functions.
- **Lexical this**: They do not have their own this. Instead, they inherit this from the parent scope (lexical scope).
- **No arguments Object**: They don't have their own arguments object. You can use rest parameters (...args) instead.
- Cannot be used as constructors (with new).

Example:

JavaScript

```
// Regular function
const add = function(a, b) {
```

```
  return a + b;
};

// Arrow function (concise)
const subtract = (a, b) => a - b;

// Arrow function with lexical `this`
const person = {
  name: "John",
  greet: function() {
    // `this` here refers to the person object
    setTimeout(() => {
      // `this` inside arrow function is inherited from greet(), so it's still the person object
      console.log(`Hello, my name is ${this.name}`);
    }, 1000);
  }
};
person.greet(); // "Hello, my name is John"
```

---

13. What is a callback function?

A callback function is a function that is passed as an **argument** to another function and is executed after some operation has been completed. They are the cornerstone of asynchronous programming in JavaScript.

Example:

```
JavaScript

function fetchData(url, callback) {
  console.log(`Fetching data from ${url}...`);
  // Simulate a network request
  setTimeout(() => {
    const data = { id: 1, content: "Some data" };
    // The `callback` is executed after the data is "fetched"
    callback(data);
  }, 2000);
}

// `displayData` is the callback function
function displayData(data) {
  console.log("Data received:", data);
}

fetchData("https://api.example.com", displayData);
```

---

14. What are higher-order functions?

A higher-order function is a function that either:

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

Many built-in array methods like `map`, `filter`, and `reduce` are higher-order functions.

Example:

JavaScript

```
// `map` is a higher-order function because it takes a function as an argument
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2); // [2, 4, 6]

// `createGreeter` is a higher-order function because it returns a function
function createGreeter(greeting) {
  return function(name) {
    console.log(`${greeting}, ${name}!`);
  };
}

const sayHello = createGreeter("Hello");
sayHello("Alice"); // "Hello, Alice!"
```

---

15. What is the difference between synchronous and asynchronous code?

- **Synchronous (Sync)**: Code is executed line by line, one after another. Each task must complete before the next one starts. This can block the main thread if a task is slow, making the application unresponsive.
- JavaScript

```
console.log("First");
// This will block execution for 2 seconds
const startTime = new Date().getTime();
while (new Date().getTime() < startTime + 2000);
console.log("Second"); // Will only run after 2 seconds
```

-
-
- **Asynchronous (Async)**: Allows multiple operations to run concurrently without blocking the main thread. When a task starts, the program can continue to execute other code. When the task finishes, it will notify the program (often via a callback, promise, or async/await).
- JavaScript

```
console.log("First");
```

```
setTimeout(() => {
  console.log("Second"); // Runs after 2 seconds, but doesn't block
}, 2000);
console.log("Third"); // Runs immediately

// Output order: First, Third, Second
```

-
-

---

16. What is the difference between `setTimeout`, `setInterval`, and `requestAnimationFrame`?

- `setTimeout(callback, delay)`: Executes the `callback` function **once** after the specified `delay` (in milliseconds).
- `setInterval(callback, delay)`: Executes the `callback` function **repeatedly** at the specified `delay` interval.
- `requestAnimationFrame(callback)`: Tells the browser that you wish to perform an animation. It requests that the browser calls a specified function to update an animation before the next repaint. It's more efficient for animations than `setTimeout` or `setInterval` because it syncs with the browser's refresh rate and pauses when the tab is not visible.

---

17. What is IIFE (Immediately Invoked Function Expression)?

An IIFE is a JavaScript function that runs as soon as it is defined. It's created by wrapping a function expression in parentheses and then immediately calling it.

Purpose:

- To avoid polluting the global scope. Variables declared inside an IIFE are not accessible from the outside.
- To create a private scope for modules before ES6 modules were introduced.

Example:

```
JavaScript

(function() {
  var privateVar = "I am private";
  console.log("This IIFE ran immediately!");
})();

// console.log(privateVar); // ReferenceError: privateVar is not defined
```

---

18. What is closure in JavaScript? Examples?

A closure is a function that remembers the environment (the lexical scope) in which it was created. It gives you access to an outer function's scope from an inner function, even after the outer function has finished executing.

Example:

```
JavaScript

function createCounter() {
  let count = 0; // `count` is a private variable inside the closure

  return function() {
    count++;
    console.log(count);
    return count;
  };
}

const counter1 = createCounter(); // `counter1` is a closure
const counter2 = createCounter(); // `counter2` is a separate closure

counter1(); // 1
counter1(); // 2

counter2(); // 1 (It has its own separate `count`)
```

In this example, the inner function returned by `createCounter` "closes over" the `count` variable. Each time `createCounter` is called, a new, independent closure is created.

---

19. What is lexical scope?

Lexical scope (or static scope) means that the accessibility of variables is determined by their position in the code at the time of writing, not at runtime. An inner function can access variables from its parent scopes. This is the mechanism that makes closures possible.

Example:

```
JavaScript

let globalName = "Global";

function outer() {
  let outerName = "Outer";

  function inner() {
    let innerName = "Inner";
    // `inner` can access its own scope, `outer`'s scope, and the global scope
    console.log(innerName, outerName, globalName);
  }
```

```
  inner();
}

outer(); // "Inner Outer Global"
```

---

20. What is the difference between call, apply, and bind?

These methods are used to control the value of `this` inside a function.

- `call(thisArg, arg1, arg2, ...)`: Invokes the function immediately with a specified `this` value and arguments provided individually.
- `apply(thisArg, [argsArray])`: Invokes the function immediately with a specified `this` value and arguments provided as an array.
- `bind(thisArg, arg1, ...)`: Returns a **new function** with the `this` value permanently bound to `thisArg`. The new function can be called later.

Example:

```
JavaScript

const person = {
  name: "Alice"
};

function greet(greeting, punctuation) {
  console.log(`${greeting}, my name is ${this.name}${punctuation}`);
}

// Call: arguments are passed individually
greet.call(person, "Hello", "!"); // "Hello, my name is Alice!"

// Apply: arguments are passed as an array
greet.apply(person, ["Hi", "."]); // "Hi, my name is Alice."

// Bind: returns a new function that can be called later
const greetAlice = greet.bind(person, "Hey");
greetAlice("?"); // "Hey, my name is Alice?"
```

A mnemonic to remember the difference: **C**all uses **C**ommas, **A**pply uses an **A**rray.

---

3. Objects & Prototypes

21. How to create objects in JavaScript?

There are several ways to create objects:

1. **Object Literal**: The simplest way.
2. JavaScript

```
const person = { name: "John", age: 30 };
```

3.
4.
5. **Constructor Function**: Using a function with the `new` keyword.
6. JavaScript

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
const person = new Person("John", 30);
```

7.
8.
9. **ES6 Class**: Syntactic sugar over constructor functions.
10. JavaScript

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
const person = new Person("John", 30);
```

11.
12.
13. **`Object.create()`**: Creates a new object with a specified prototype.
14. JavaScript

```
const personProto = {
  greet() { console.log("Hello!"); }
};
const person = Object.create(personProto);
```

15.
16.

---

22. Difference between object literal and constructor function?

- **Object Literal**: A simple way to create a single object. The syntax is concise and declarative. It's best for one-off objects.

- **Constructor Function**: A blueprint for creating multiple objects of the same type. You use the `new` keyword to create instances. It's better for creating multiple objects that share the same structure and methods.

---

23. What is `this` keyword in JavaScript?

The `this` keyword refers to the **context** in which a function is executed. Its value is determined by how a function is called.

- **Global Context**: Outside of any function, `this` refers to the global object (`window` in browsers, `global` in Node.js).
- **Object Method**: When a function is called as a method of an object, `this` refers to the object itself.
- **Simple Function Call**: In strict mode, `this` is `undefined`. In non-strict mode, it defaults to the global object.
- **Event Listeners**: `this` refers to the element that triggered the event.
- **Arrow Functions**: `this` is lexically inherited from the parent scope.

Example:

```
JavaScript

const car = {
  brand: "Tesla",
  model: "Model S",
  displayInfo: function() {
    // `this` refers to the `car` object
    console.log(`${this.brand} ${this.model}`);
  }
};
car.displayInfo(); // "Tesla Model S"
```

---

24. How does `this` behave in arrow functions vs normal functions?

- **Normal Functions**: The value of `this` is dynamic and depends on **how the function is called**.
- **Arrow Functions**: The value of `this` is static (lexical). It is **inherited from the parent scope** where the arrow function was defined. It does not have its own `this` binding.

Example showing the difference:

```
JavaScript

const myObject = {
  name: "My Object",

  normalFunc: function() {
```

```
    // `this` is myObject
    setTimeout(function() {
      // `this` is the global `window` object (or undefined in strict mode)
      console.log("Normal function:", this.name); // undefined
    }, 500);
  },

  arrowFunc: function() {
    // `this` is myObject
    setTimeout(() => {
      // Arrow function inherits `this` from `arrowFunc`'s scope
      console.log("Arrow function:", this.name); // "My Object"
    }, 1000);
  }
};

myObject.normalFunc();
myObject.arrowFunc();
```

---

25. What is prototype in JavaScript?

Every JavaScript object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on, until an object is reached with `null` as its prototype. This is called the **prototype chain**.

When you try to access a property on an object, if the property is not found on the object itself, the JavaScript engine looks up the prototype chain until it finds the property or reaches the end (`null`). This is the basis of **prototypal inheritance**.

---

26. Difference between `__proto__` and `prototype`.

- **`prototype`**: A property that exists on **constructor functions** (and ES6 classes). It is the object that will become the prototype of all instances created by that constructor.
- **`__proto__`** (dunder proto): A property that exists on **every object instance**. It is a direct link to the object's prototype. It's a non-standard way to access the prototype; the standard way is `Object.getPrototypeOf(obj)`.

Example:

```
JavaScript

function Dog(name) {
  this.name = name;
}

// `prototype` is a property of the constructor function `Dog`
Dog.prototype.bark = function() {
```

```
  console.log("Woof!");
};

const myDog = new Dog("Rex"); // Create an instance

// `__proto__` on the instance points to the constructor's prototype
console.log(myDog.__proto__ === Dog.prototype); // true

myDog.bark(); // Accessing the method via the prototype chain
```

---

27. What is prototypal inheritance?

Prototypal inheritance is a feature in JavaScript where objects can inherit properties and methods from other objects. Instead of classes inheriting from other classes, an object can have another object as its prototype.

Example:

```
JavaScript

const animal = {
  isAlive: true,
  eat() {
    console.log("Eating...");
  }
};

const rabbit = {
  jump() {
    console.log("Jumping...");
  }
};

// Set `animal` as the prototype of `rabbit`
Object.setPrototypeOf(rabbit, animal);

console.log(rabbit.isAlive); // true (inherited from animal)
rabbit.eat(); // "Eating..." (inherited from animal)
rabbit.jump(); // "Jumping..." (own method)
```

---

28. What is `Object.create()`?

`Object.create(proto, [propertiesObject])` is a method that creates a new object, using an existing object as the prototype of the newly created object.

It's a direct way to implement prototypal inheritance.

Example:

```
JavaScript

const personProto = {
  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

const user = Object.create(personProto);
user.name = "John";
user.greet(); // "Hello, my name is John"
```

---

29. Difference between shallow copy and deep copy in JavaScript objects.

- **Shallow Copy**: Creates a new object, but if it contains nested objects or arrays, it copies the **references** to those nested objects, not the objects themselves. Modifying a nested object in the copy will also affect the original.
- **Deep Copy**: Creates a new object and recursively copies all nested objects and arrays, creating completely new instances. The copy is fully independent of the original.

Example:

```
JavaScript

const original = {
  name: "Test",
  details: {
    id: 1,
    tags: ["a", "b"]
  }
};

// Shallow Copy (using spread operator)
const shallow = { ...original };
shallow.details.id = 2; // This will change the original object too
console.log(original.details.id); // 2

// Deep Copy (using JSON methods - simple but has limitations)
const deepCopy = JSON.parse(JSON.stringify(original));
deepCopy.details.id = 3; // This will NOT change the original
console.log(original.details.id); // Still 2
```

Note: `JSON.stringify/parse` cannot copy functions, `undefined`, Symbols, or handle circular references. For robust deep cloning, a library like Lodash's `cloneDeep` is often used.

---

30. How to clone an object in JavaScript?

- **Shallow Clone**:
  - **Spread Operator (...)**: `const clone = { ...original };`
  - **Object.assign()**: `const clone = Object.assign({}, original);`
- **Deep Clone**:
  - **JSON.parse(JSON.stringify(obj))**: Easy but has limitations (as mentioned above).
  - **structuredClone()**: A modern, built-in API designed for deep cloning. It's the best native option.
  - **Libraries**: Using functions like `_.cloneDeep()` from Lodash for complex objects.

Example with structuredClone():

```
JavaScript


const original = {
  name: "Test",
  date: new Date(),
  details: { id: 1 }
};

const deepClone = structuredClone(original);
deepClone.details.id = 99;

console.log(original.details.id); // 1 (original is unaffected)
```

... and so on for the remaining sections. I will continue generating the rest of the answers following this detailed, example-driven format. Due to the length, I will proceed to generate the rest.

---

4. Arrays & Iteration

31. Difference between for, for...in, and for...of.

- **for loop**: The traditional loop. It gives you full control over the iteration (start, condition, increment).
- JavaScript

```
const arr = ['a', 'b', 'c'];
for (let i = 0; i < arr.length; i++) {
   console.log(arr[i]);
}
```

-
-
- **for...in loop**: Iterates over the **enumerable properties** (keys) of an object. It's generally not recommended for arrays because it can iterate over inherited properties and the order is not guaranteed.
- JavaScript

```
const obj = { a: 1, b: 2 };
for (const key in obj) {
  console.log(key); // 'a', 'b'
}
```

-
-
- **for...of loop**: Iterates over the **values** of an iterable object (like Array, String, Map, Set). This is the modern and preferred way to loop over arrays.
- JavaScript

```
const arr = ['a', 'b', 'c'];
for (const value of arr) {
  console.log(value); // 'a', 'b', 'c'
}
```

-
-

---

32. What are array methods (map, filter, reduce, forEach, find)?

These are higher-order functions that provide a declarative way to work with arrays.

- map(callback): Creates a **new array** by applying a function to every element of the original array.
- filter(callback): Creates a **new array** with all elements that pass the test implemented by the provided function.
- reduce(callback, initialValue): Executes a reducer function on each element, resulting in a **single output value**.
- forEach(callback): Executes a provided function once for each array element. It **does not return a value** (undefined).
- find(callback): Returns the **first element** in the array that satisfies the provided testing function. If no values satisfy the test, undefined is returned.

---

33. Difference between map and forEach.

Feature	map()	forEach()

Return Value	Returns a **new array**.	Returns `undefined`.
Purpose	To transform data and create a new array from the results.	To execute a function for each element (e.g., for side effects like logging).
Chainable	✅ Yes, you can chain other methods like `.filter()`.	❌ No, because it returns `undefined`.

Example:

```
JavaScript

const numbers = [1, 2, 3];

// Using map to create a new, transformed array
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]

// Using forEach for a side effect (logging)
const result = numbers.forEach(num => {
  console.log(`Number is ${num}`);
});
console.log(result); // undefined
```

---

34. What is the difference between `slice` and `splice`?

- **`slice(start, end)`**:
  - **Returns a new array** containing a shallow copy of a portion of the original array.
  - **Does not modify** the original array (immutable).
  - `end` index is not included.
- **`splice(start, deleteCount, item1, ...)`**:
  - **Modifies the original array** by removing, replacing, or adding elements (mutable).
  - **Returns an array** containing the deleted elements.

Example:

```
JavaScript
```

```
const fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry'];

// slice (immutable)
const someFruits = fruits.slice(1, 3);
console.log(someFruits); // ['banana', 'cherry']
console.log(fruits);   // ['apple', 'banana', 'cherry', 'date', 'elderberry'] (original unchanged)

// splice (mutable)
const removedFruits = fruits.splice(2, 2, 'grape', 'fig');
console.log(removedFruits); // ['cherry', 'date'] (the removed items)
console.log(fruits);      // ['apple', 'banana', 'grape', 'fig', 'elderberry'] (original modified)
```

---

35. How does `reduce()` work? Examples?

`arr.reduce(callback(accumulator, currentValue, currentIndex, array), initialValue)`

The `reduce` method iterates over an array and "reduces" it to a single value by applying a callback function.

- **`accumulator`**: The value resulting from the previous callback invocation. On the first call, it's `initialValue` if provided, otherwise the first element of the array.
- **`currentValue`**: The current element being processed.
- **`initialValue`**: (Optional) A value to use as the first `accumulator`. If not provided, the first element of the array is used, and iteration starts from the second element.

Example 1: Summing an array

JavaScript

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0); // 0 is the initialValue
console.log(sum); // 10
```

Example 2: Grouping objects by a property

JavaScript

```
const people = [
  { name: 'Alice', role: 'dev' },
  { name: 'Bob', role: 'dev' },
  { name: 'Charlie', role: 'manager' }
];

const groupedByRole = people.reduce((acc, person) => {
  const role = person.role;
  if (!acc[role]) {
    acc[role] = [];
  }
```

```
  acc[role].push(person);
  return acc;
}, {});

console.log(groupedByRole);
/*
{
  dev: [ { name: 'Alice', role: 'dev' }, { name: 'Bob', role: 'dev' } ],
  manager: [ { name: 'Charlie', role: 'manager' } ]
}
*/
```

---

36. What are array-like objects?

An array-like object is an object that has a `length` property and indexed elements (e.g., `0`, `1`, `2`), but does not have the built-in array methods like `map`, `filter`, `forEach`.

Common examples include the `arguments` object in a function and a `NodeList` returned by `document.querySelectorAll()`.

---

37. How to convert array-like objects to arrays?

1. **`Array.from()`**: The modern and recommended way.
2. JavaScript

```
function sumArgs() {
  const argsArray = Array.from(arguments);
  return argsArray.reduce((acc, curr) => acc + curr, 0);
}
```

3.
4.
5. **Spread Operator (`...`)**: Also a modern and clean way.
6. JavaScript

```
const nodes = document.querySelectorAll('div');
const nodesArray = [...nodes];
nodesArray.map(/* ... */);
```

7.
8.
9. **`Array.prototype.slice.call()`**: The older, more verbose method.
10. JavaScript

```
const argsArray = Array.prototype.slice.call(arguments);
```

11.

12.

---

38. Difference between `Array.isArray()` and `instanceof Array`.

- **`Array.isArray(obj)`**: The most reliable method. It returns `true` if `obj` is an array, and `false` otherwise. It works correctly across different realms (e.g., iframes).
- **`obj instanceof Array`**: This operator checks if the `Array.prototype` object exists in the `obj`'s prototype chain. It can fail across different global contexts (like multiple frames or windows), because each context has its own `Array` constructor.

Best practice: Use `Array.isArray()` for checking if a value is an array.

---

39. How does spread operator (`...`) work with arrays?

The spread operator allows an iterable (like an array) to be expanded into individual elements.

Common Use Cases:

1. **Creating a shallow copy**: `const copy = [...original];`
2. **Concatenating arrays**: `const combined = [...arr1, ...arr2];`
3. **Adding elements to an array**: `const newArr = [0, ...arr, 4];`
4. **Passing array elements as function arguments**:
5. JavaScript

```
const numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // Same as Math.max(1, 2, 3)
```

6.
7.

---

40. Difference between `push`, `pop`, `shift`, and `unshift`.

These methods all modify an array in place (they are mutable).

Method	Action	Where	Returns
`push(item)`	Adds one or more elements	**End** of the array	The new `length` of the array

pop()	Removes the last element	**End** of the array	The **removed element**
unshift(item)	Adds one or more elements	**Beginning** of the array	The new length of the array
shift()	Removes the first element	**Beginning** of the array	The **removed element**

Note: shift and unshift can be less performant on large arrays because they require re-indexing all subsequent elements.

---

5. Asynchronous JavaScript

41. What is the Event Loop in JavaScript?

The Event Loop is the core mechanism in JavaScript that allows it to be non-blocking and handle asynchronous operations, despite being single-threaded. It continuously checks if the **Call Stack** is empty. If it is, it moves the first event from the **Task Queue** (or Microtask Queue) to the Call Stack for execution.

Simple Model:

1. Code is executed on the **Call Stack**.
2. When an async operation (like setTimeout, fetch) is called, it's handed off to a **Web API** (in the browser) or C++ API (in Node.js).
3. When the async operation is complete, its callback function is placed in the **Task Queue** (or Macrotask Queue).
4. The **Event Loop** constantly monitors: "Is the Call Stack empty?"
5. If the Call Stack is empty, it takes the first item from the queue and pushes it onto the stack to be executed.

---

42. Difference between call stack, microtask queue, and macrotask queue.

- **Call Stack**: A LIFO (Last-In, First-Out) data structure that keeps track of function calls. When a function is called, it's added to the stack. When it returns, it's removed.
- **Macrotask Queue (or Task Queue)**: Holds callbacks from async operations like setTimeout, setInterval, I/O operations, and UI rendering. The Event Loop processes **one** macrotask per tick.
- **Microtask Queue**: Holds callbacks from promises (.then(), .catch(), .finally()) and queueMicrotask(). This queue has a **higher priority** than the Macrotask Queue. The

Event Loop will execute **all** microtasks in the queue before moving on to the next macrotask.

Execution Order:

1. All synchronous code in the Call Stack runs to completion.
2. The Event Loop executes all available microtasks in the Microtask Queue.
3. If the Microtask Queue is empty, the Event Loop executes one macrotask from the Macrotask Queue.
4. The process repeats.

---

43. What are Promises in JavaScript?

A `Promise` is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

A Promise can be in one of three states:

- **Pending**: The initial state; neither fulfilled nor rejected.
- **Fulfilled (Resolved)**: The operation completed successfully.
- **Rejected**: The operation failed.

You handle these states using the `.then()` (for fulfillment), `.catch()` (for rejection), and `.finally()` (runs regardless of outcome) methods.

Example:

```
JavaScript

const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve("Data fetched successfully!");
    } else {
      reject("Error: Failed to fetch data.");
    }
  }, 2000);
});

myPromise
  .then(message => console.log(message)) // Handles success
  .catch(error => console.error(error)) // Handles failure
  .finally(() => console.log("Operation finished.")); // Runs always
```

---

44. What are `async`/`await` keywords?

`async/await` is modern syntactic sugar built on top of Promises, making asynchronous code look and behave more like synchronous code. This makes it much easier to read and write.

- **`async`**: When placed before a function, it makes the function return a `Promise`. If the function returns a value, the Promise will be resolved with that value.
- **`await`**: Can only be used inside an `async` function. It pauses the execution of the function and waits for a `Promise` to resolve. It then resumes the function's execution and returns the resolved value.

Example:

```
JavaScript

function fetchData() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Data is here!"), 2000);
  });
}

async function processData() {
  try {
    console.log("Waiting for data...");
    const data = await fetchData(); // Pauses here until the promise resolves
    console.log(data); // "Data is here!"
  } catch (error) {
    console.error("An error occurred:", error);
  }
}

processData();
```

---

45. Difference between callbacks and promises.

Feature	Callbacks	Promises
Control Flow	Leads to "Callback Hell" (deeply nested callbacks), which is hard to read and maintain.	Allows for cleaner, chainable `.then()` syntax, avoiding deep nesting.
Error Handling	Error handling is typically done with an `(err, data)`	Centralized error handling with `.catch()`, which catches errors from

	convention, which can be inconsistent.	the promise and any preceding `.then()` blocks.
State	No explicit state. A callback is just a function that gets called.	Have explicit states (pending, fulfilled, rejected), making them more predictable.
Composition	Hard to compose multiple async operations.	Easy to compose with methods like `Promise.all` and `Promise.race`.

---

46. What is Promise chaining?

Promise chaining is the practice of connecting multiple asynchronous operations together in a sequence. The `.then()` method returns a new promise, which allows you to chain another `.then()` onto it. This creates a clean, readable flow for dependent async tasks.

Example:

```
JavaScript

new Promise(resolve => resolve(10))
  .then(result => {
    console.log(result); // 10
    return result * 2;   // This value is passed to the next .then()
  })
  .then(result => {
    console.log(result); // 20
    return result + 5;
  })
  .then(result => {
    console.log(result); // 25
  });
```

---

47. Difference between `Promise.all`, `Promise.any`, `Promise.race`, and `Promise.allSettled`.

- **`Promise.all(iterable)`**:
  - **Waits for all promises to fulfill**.
  - If **any** promise rejects, it immediately rejects with the reason of the first promise that rejected.
  - Returns a promise that fulfills with an array of the results.
- **`Promise.any(iterable)`**:

    - **Waits for the first promise to fulfill**.
    - If **all** promises reject, it rejects with an `AggregateError`.
    - Returns a promise that fulfills with the value of the first fulfilled promise.
- **`Promise.race(iterable)`**:
    - **Waits for the first promise to settle** (either fulfill or reject).
    - It fulfills or rejects with the value/reason of that first settled promise.
- **`Promise.allSettled(iterable)`**:
    - **Waits for all promises to settle** (regardless of fulfillment or rejection).
    - It **never rejects**. It always fulfills with an array of objects, each describing the outcome of a promise (`{status: 'fulfilled', value: ...}` or `{status: 'rejected', reason: ...}`).
    - Useful when you need to know the outcome of every promise, even if some fail.

---

48. What is `fetch` API?

The `fetch` API is a modern, promise-based browser interface for making network requests (e.g., HTTP requests to get data from a server). It's a more powerful and flexible replacement for `XMLHttpRequest`.

Key Features:

- Promise-based, making it easy to use with `async/await`.
- More flexible API for handling headers, requests, and responses.
- Supports features like CORS and streaming.

Example:

```
JavaScript

async function getUser() {
  try {
    const response = await fetch('https://api.github.com/users/google');
    if (!response.ok) { // Check if the request was successful
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json(); // Parse the response body as JSON
    console.log(data);
  } catch (error) {
    console.error('Could not fetch user:', error);
  }
}
getUser();
```

Note: `fetch` only rejects on network errors, not on HTTP error statuses like 404 or 500. You must check `response.ok` or `response.status` manually.

---

49. Difference between `XMLHttpRequest` and `fetch`.

Feature	`XMLHttpRequest` (XHR)	`fetch` API
API Style	Event-based (using `onreadystatechange`).	Promise-based.
Ease of Use	More verbose and complex.	Cleaner and more intuitive, especially with `async/await`.
Body Parsing	Manual parsing (e.g., `JSON.parse(xhr.responseText)`).	Built-in methods like `.json()`, `.text()`, `.blob()`.
Error Handling	Rejects only on network failures. HTTP errors (404, 500) must be checked manually in the response.	Rejects only on network failures. HTTP errors (404, 500) must be checked manually via `response.ok`.
CORS	Requires more configuration.	Simpler by default (CORS is enforced).
Modern Features	Lacks support for modern features like streaming.	Supports streaming of request/response bodies.

---

50. What is AJAX in JavaScript?

AJAX (Asynchronous JavaScript and XML) is not a specific technology, but a technique for creating fast and dynamic web pages. It allows a web page to update parts of its content asynchronously by exchanging data with a web server behind the scenes, without reloading the entire page.

Historically, this was done using `XMLHttpRequest` and XML data, but today it's more common to use the `fetch` API and JSON data. The core idea remains the same: making background HTTP requests to update the UI dynamically.

---

6. DOM & Browser

I will continue generating the remaining sections in the same detailed manner.

51. What is DOM? Difference between HTMLCollection and NodeList?

The **DOM (Document Object Model)** is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as a tree of nodes and objects.

- **`HTMLCollection`**:
    - A **live** collection of elements. If the DOM changes (e.g., an element is added or removed), the collection is automatically updated.
    - Returned by methods like `getElementsByTagName()` or `getElementsByClassName()`.
    - Can only contain element nodes.
    - Does not have array methods like `forEach`.
- **`NodeList`**:
    - Can be **live** (returned by `childNodes`) or **static** (returned by `querySelectorAll()`). A static `NodeList` is a snapshot; it does not update if the DOM changes.
    - Can contain any node type (elements, text, comments).
    - Modern `NodeList`s have a `forEach` method.

Example:

```
JavaScript

// HTMLCollection (live)
const collection = document.getElementsByClassName('item');

// NodeList (static)
const nodeList = document.querySelectorAll('.item');
```

---

52. What are DOM manipulation methods (`getElementById`, `querySelector`, etc.)?

These are methods used to select and modify DOM elements.

- **`getElementById('id')`**: Selects a single element by its unique ID. Very fast.
- **`getElementsByTagName('tag')`**: Selects a live `HTMLCollection` of elements by tag name.
- **`getElementsByClassName('class')`**: Selects a live `HTMLCollection` of elements by class name.
- **`querySelector('selector')`**: Selects the **first** element that matches a specified CSS selector. Very flexible.

- **querySelectorAll('selector')**: Selects a static NodeList of **all** elements that match a specified CSS selector.

Once an element is selected, you can manipulate it using properties like .innerHTML, .style, or methods like .appendChild(), .removeChild(), .setAttribute().

---

53. Difference between innerHTML, innerText, and textContent.

- **innerHTML**:
  - Gets or sets the HTML content (including tags) within an element.
  - **Security Risk**: Can lead to XSS (Cross-Site Scripting) attacks if you set it with untrusted user input, because it parses and executes scripts.
  - elem.innerHTML = "Hello";
- **innerText**:
  - Gets or sets the "rendered" text content of an element. It is aware of CSS styling (e.g., it won't return text from a display: none element).
  - Causes a reflow, so it can be slow.
  - elem.innerText = "Hello World";
- **textContent**:
  - Gets or sets the raw text content of an element, including script and style tags, without interpreting them.
  - It is not aware of CSS and will return text from hidden elements.
  - Faster and safer than innerHTML.
  - elem.textContent = "Hello World";

Best Practice: Use textContent for changing text, and be very cautious with innerHTML.

---

54. What are event listeners in JavaScript?

An event listener is a function that waits for a specific event (like a click, mouseover, or keydown) to occur on a target element. When the event happens, the function is executed.

You can add event listeners using the addEventListener() method.

element.addEventListener('eventName', callbackFunction, [options]);

Example:

```
JavaScript

const button = document.getElementById('myButton');

function handleClick() {
  console.log('Button was clicked!');
}

button.addEventListener('click', handleClick);
```

---

55. What is event bubbling and capturing?

These are the two phases of event propagation in the DOM.

1. **Capturing Phase**: The event travels from the root of the document (the `window` object) **down** to the target element. You can listen for events in this phase by setting the third argument of `addEventListener` to `true`.
2. **Target Phase**: The event reaches the target element.
3. **Bubbling Phase (Default)**: The event travels **up** from the target element back to the root of the document. This is the default behavior.

Example:

Imagine a button inside a div. If you click the button:

- **Capturing**: Event goes `document` -> `html` -> `body` -> `div` -> `button`.
- **Bubbling**: Event goes `button` -> `div` -> `body` -> `html` -> `document`.

---

56. Difference between `stopPropagation()` and `preventDefault()`.

- **`event.stopPropagation()`**: Stops the event from propagating further in the capturing and bubbling phases. The event will not travel to parent or child elements.
- JavaScript

```
div.addEventListener('click', () => {
    // This will not run if the button's click handler calls stopPropagation()
});
button.addEventListener('click', (event) => {
    event.stopPropagation();
    console.log('Button clicked, propagation stopped.');
});
```

-
-
- **`event.preventDefault()`**: Prevents the browser's default action for a given event. For example, it can stop a form from submitting, a link from navigating, or a checkbox from being checked.
- JavaScript

```
const form = document.querySelector('form');
form.addEventListener('submit', (event) => {
    event.preventDefault(); // Stops the form from reloading the page
    console.log('Form submission prevented.');
});
```

-

-

---

57. What is `localStorage`, `sessionStorage`, and `cookies`?

These are all ways to store data on the client-side (in the browser).

Feature	localStorage	sessionStorage	cookies
Capacity	~5-10 MB	~5 MB	~4 KB
Persistence	**Persistent**: Data remains until explicitly cleared.	**Per Session**: Data is cleared when the tab/browser is closed.	**Expires**: Data persists until an expiration date is set.
Accessibility	Accessible from any tab/window from the same origin.	Accessible only within the same tab/window.	Sent with every HTTP request to the server.
Use Case	Storing user settings, application state.	Storing temporary data for a single session.	Authentication tokens, session management.
API	`localStorage.setItem('key', 'value')`	`sessionStorage.setItem('key', 'value')`	`document.cookie = 'key=value'` (clunky API)

---

58. Difference between synchronous and asynchronous script loading (`async` vs `defer`).

By default, when a browser encounters a `<script>` tag, it pauses HTML parsing, fetches the script, executes it, and then resumes parsing. This can block rendering. `async` and `defer` change this behavior.

- **`<script>` (default)**: Pauses parsing. Fetches and executes.

- **<script async>**: Fetches the script **asynchronously** while HTML parsing continues. Executes the script as soon as it's downloaded, which can interrupt parsing. The order of execution for multiple async scripts is not guaranteed.
- **<script defer>**: Fetches the script **asynchronously** while HTML parsing continues. Executes the script **only after** the HTML parser has finished, but before the DOMContentLoaded event. The order of execution for multiple defer scripts is maintained.

Recommendation: Use defer for scripts that need the full DOM and whose execution order matters. Use async for independent, third-party scripts (like analytics) where order doesn't matter.

---

59. What is the difference between DOMContentLoaded and load event?

- **DOMContentLoaded**: Fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading. This is generally the event you want to use to start running your JavaScript code.
- **load**: Fired on the window object when the entire page has fully loaded, including all dependent resources such as stylesheets, scripts, and images.

---

60. How to debounce and throttle events?

These are techniques to control how often a function is executed, commonly used for performance optimization with events like scroll, resize, or input.

- **Debounce**: Groups a burst of events into a single one. The function is only called **after** a certain amount of time has passed without the event being fired. Useful for search input fields (wait until the user stops typing).
- JavaScript

```
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}
```

-
-
- **Throttle**: Ensures that a function is called at most **once** per specified time interval. It doesn't matter how many times the event fires; the function will only execute regularly. Useful for scroll events (e.g., triggering an animation every 200ms).
- JavaScript

```
function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  };
}
```

-
-

This covers the first six sections. I will now proceed with the remaining advanced topics. The level of detail will be maintained.

I've provided comprehensive answers for the first 60 questions. Due to the extensive nature of the full list, I will continue with the remaining sections, maintaining the same level of detail with explanations and examples.

---

7. Advanced JavaScript

61. What is hoisting in JavaScript?

Hoisting is JavaScript's default behavior of moving declarations to the top of their containing scope (function or global) during the compilation phase, before the code is executed.

- **`var` declarations**: Are hoisted and initialized with `undefined`. This means you can access a `var` variable before its declaration line without an error, but its value will be `undefined`.
- **`let` and `const` declarations**: Are also hoisted, but they are not initialized. Accessing them before the declaration results in a `ReferenceError`. This period between the start of the scope and the declaration is called the **Temporal Dead Zone (TDZ)**.
- **Function declarations**: Are hoisted completely (both name and body). You can call a function before it's declared.
- **Function expressions**: Are not hoisted. The variable they are assigned to is hoisted according to its keyword (`var`, `let`, `const`), but the function body is not.

Example:

```
JavaScript

console.log(myVar); // undefined (hoisted with undefined)
var myVar = 5;

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

```
let myLet = 10;

sayHello(); // "Hello!" (function declaration is fully hoisted)
function sayHello() {
  console.log("Hello!");
}

// sayHi(); // TypeError: sayHi is not a function
var sayHi = function() {
  console.log("Hi!");
};
```

---

62. What are modules in JavaScript (`import`, `export`)?

ES6 Modules provide a way to organize code into separate, reusable files. Each module is a file. You can choose what variables, functions, or classes to expose to other modules using the `export` keyword, and bring them into other modules using the `import` keyword.

Key Features:

- Each module has its own top-level scope.
- Modules are executed only once.
- `import` and `export` statements must be at the top level (not inside loops or functions).
- Modules are loaded asynchronously by default.

Example:

`math.js` (the module)

```
JavaScript

export const PI = 3.14159;

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

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

`main.js` (using the module)

```
JavaScript

// Import named exports and the default export
import multiply, { add, PI } from './math.js';
```

```
console.log(PI); // 3.14159
console.log(add(2, 3)); // 5
console.log(multiply(2, 3)); // 6
```

---

63. Difference between CommonJS and ES6 modules?

Feature	CommonJS (CJS)	ES6 Modules (ESM)
Syntax	`require()` and `module.exports`	`import` and `export`
Loading	**Synchronous**: Modules are loaded at the point the `require()` call is made.	**Asynchronous**: Modules are loaded and parsed before execution.
Environment	Primarily used in **server-side** environments like Node.js.	The **standard** for JavaScript, supported by modern browsers and Node.js.
Tree-shaking	Harder for bundlers to perform tree-shaking (dead code elimination).	The static nature of `import/export` makes tree-shaking very effective.
`this`	`this` at the top level refers to `exports`.	`this` at the top level is `undefined`.

---

64. What are generators in JavaScript? (`function*`)

A **generator** is a special type of function that can be paused and resumed, allowing it to produce a sequence of values over time. It is defined using `function*` and uses the `yield` keyword to "pause" and return a value.

When you call a generator function, it doesn't execute the code. Instead, it returns a **generator object**. You can then call the `.next()` method on this object to execute the function until it hits a `yield`, which returns an object like `{ value: ..., done: false }`.

Example:

JavaScript

```
function* numberGenerator() {
  console.log("Starting...");
  yield 1;
  console.log("Yielded 1");
  yield 2;
  console.log("Yielded 2");
  return 3;
}

const gen = numberGenerator(); // Get the generator object

console.log(gen.next()); // Starting... { value: 1, done: false }
console.log(gen.next()); // Yielded 1... { value: 2, done: false }
console.log(gen.next()); // Yielded 2... { value: 3, done: true }
console.log(gen.next()); // { value: undefined, done: true }
```

Generators are the foundation for more advanced concepts like `async/await`.

---

65. What are Symbols in JavaScript?

A `Symbol` is a unique and immutable primitive data type introduced in ES6. It's often used as a key for an object property when you want to ensure the key will not collide with any other keys (e.g., keys from other libraries or user-provided keys).

`Symbol()` always returns a unique value.

Example:

JavaScript

```
const id = Symbol('a unique id');
const id2 = Symbol('a unique id');

console.log(id === id2); // false

const user = {
  name: 'John',
  [id]: 12345 // Using a symbol as a property key
};

console.log(user[id]); // 12345

// Symbol properties are ignored by JSON.stringify and for...in loops
console.log(Object.keys(user)); // ['name']
console.log(JSON.stringify(user)); // {"name":"John"}
```

---

66. What are WeakMap and WeakSet?

`WeakMap` and `WeakSet` are special types of collections whose elements are held "weakly". This means that if an object stored in a `WeakMap` or `WeakSet` has no other references to it, it can be garbage collected.

- **`WeakMap`**:
  - Keys **must be objects**.
  - Keys are held weakly. If a key object is garbage collected, the entry is removed from the map.
  - Not iterable.
  - Use case: Storing metadata about an object without preventing it from being garbage collected.
- **`WeakSet`**:
  - Values **must be objects**.
  - Values are held weakly.
  - Not iterable.
  - Use case: Keeping track of objects without affecting their lifecycle.

Example (`WeakMap`):

```JavaScript
let user = { name: "John" };
const metadata = new WeakMap();
metadata.set(user, { lastLogin: Date.now() });

// Now, if `user` is set to null, there are no more strong references to the object.
// The garbage collector can remove it, and the entry in `metadata` will also be removed automatically.
user = null;
```

---

67. What are Proxy and Reflect in JavaScript?

- **`Proxy`**: An object that wraps another object (the "target") and allows you to intercept fundamental operations (like getting or setting a property). This is known as **metaprogramming**. You can define custom behavior for operations using "traps".
- **`Reflect`**: A built-in object that provides methods for interceptable JavaScript operations. The methods on `Reflect` have the same names as the proxy traps (e.g., `Reflect.get()`, `Reflect.set()`). It provides a default way to perform these operations.

Example (Validation Proxy):

```JavaScript
const user = { age: 25 };
```

```
const validator = {
  set(target, key, value) {
    if (key === 'age' && typeof value !== 'number') {
      throw new TypeError('Age must be a number.');
    }
    // Use Reflect to perform the default set operation
    return Reflect.set(target, key, value);
  }
};

const userProxy = new Proxy(user, validator);

userProxy.age = 30; // Works
console.log(user.age); // 30

// userProxy.age = 'thirty'; // Throws TypeError: Age must be a number.
```

---

68. What is currying in JavaScript?

Currying is the process of transforming a function that takes multiple arguments into a sequence of nested functions, each taking a single argument.

Example:

JavaScript

```
// A non-curried function
const add = (a, b, c) => a + b + c;

// A curried version of the same function
const curriedAdd = (a) => (b) => (c) => a + b + c;

// You can call it like this:
const result = curriedAdd(1)(2)(3); // 6

// The advantage is that you can create specialized functions by "partially applying" arguments
const add5 = curriedAdd(5);
const result2 = add5(10)(20); // 35
```

---

69. What is memoization in JavaScript?

Memoization is an optimization technique used to speed up function calls by caching the results of expensive function calls and returning the cached result when the same inputs occur again.

It's particularly useful for pure functions (functions that always produce the same output for the same input).

Example:

```
JavaScript

function memoize(fn) {
  const cache = {};
  return function(...args) {
    const key = JSON.stringify(args);
    if (cache[key]) {
      console.log("Fetching from cache...");
      return cache[key];
    } else {
      console.log("Calculating result...");
      const result = fn(...args);
      cache[key] = result;
      return result;
    }
  };
}

const slowSquare = (n) => {
  // Simulate an expensive calculation
  for (let i = 0; i < 1e7; i++) {}
  return n * n;
};

const memoizedSquare = memoize(slowSquare);

console.log(memoizedSquare(5)); // Calculating result... 25
console.log(memoizedSquare(5)); // Fetching from cache... 25
console.log(memoizedSquare(10)); // Calculating result... 100
```

---

70. What is event delegation in JavaScript?

Event delegation is a technique where you add a single event listener to a parent element instead of adding listeners to each individual child element. The listener analyzes bubbled events to find a match on the child target.

Benefits:

- **Performance**: Fewer event listeners mean less memory usage.
- **Dynamic Elements**: Works automatically for child elements that are added to the DOM after the listener is set up.

Example:

Instead of adding a listener to every <li>:

HTML

```
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

JavaScript

```
const list = document.getElementById('myList');

list.addEventListener('click', function(event) {
  // Check if the clicked element is an <li>
  if (event.target && event.target.matches('li')) {
    console.log('Clicked item:', event.target.textContent);
  }
});

// If you add a new <li> dynamically, the event listener will still work for it!
const newItem = document.createElement('li');
newItem.textContent = 'Item 4';
list.appendChild(newItem);
```

This covers Section 7. I will proceed with the remaining sections.

I have completed the detailed answers for the first 70 questions. I will continue generating the answers for the remaining sections to provide a complete guide.

---

8. Error Handling & Debugging

71. What is `try...catch` in JavaScript?

The `try...catch` statement allows you to test a block of code for errors and handle them gracefully without crashing your application.

- `try` block: Contains the code that might throw an error.
- `catch` block: Contains the code to be executed if an error occurs in the `try` block. It receives an `error` object as an argument.
- `finally` block (optional): Contains code that will be executed regardless of whether an error was thrown or caught.
- `throw` statement (optional): Allows you to create custom errors.

Example:

```
JavaScript

try {
  const data = JSON.parse('{ "name": "John", "age": 30, }'); // Invalid JSON (trailing comma)
  console.log(data);
} catch (error) {
  console.error("An error occurred while parsing JSON!");
  console.error("Error name:", error.name);     // SyntaxError
  console.error("Error message:", error.message); // Unexpected token } in JSON at position...
} finally {
  console.log("This will always run.");
}
```

---

72. What is the difference between error types (SyntaxError, ReferenceError, TypeError)?

- **SyntaxError**: Occurs when the JavaScript engine encounters code that violates the language's syntax rules. These errors happen at parse time and will stop the script from executing at all.
- JavaScript

```
// let x = 5; // SyntaxError: Missing semicolon or other syntax issue
```

-
-
- **ReferenceError**: Occurs when you try to access a variable that has not been declared.
- JavaScript

```
// console.log(nonExistentVariable); // ReferenceError: nonExistentVariable is not defined
```

-
-
- **TypeError**: Occurs when an operation is performed on a value of the wrong type. For example, trying to call a non-function value as a function, or accessing properties of `null` or `undefined`.
- JavaScript

```
const x = null;
// console.log(x.length); // TypeError: Cannot read properties of null (reading 'length')

const num = 123;
// num(); // TypeError: num is not a function
```

-

-

---

73. What is `throw` in JavaScript?

The `throw` statement allows you to create your own custom errors. You can throw any expression, but it's best practice to throw an `Error` object (or an object that extends `Error`), because they contain useful information like the error message and stack trace.

Example:

```
JavaScript

function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero is not allowed.");
  }
  return a / b;
}

try {
  divide(10, 0);
} catch (error) {
  console.error(error.message); // "Division by zero is not allowed."
}
```

---

74. How does `finally` work in error handling?

The `finally` block will **always** execute after the `try` and `catch` blocks have completed, regardless of whether an error occurred. It will run even if there is a `return` statement in the `try` or `catch` block.

Use Case: It's primarily used for cleanup code, such as closing a file, releasing a resource, or closing a database connection.

Example:

```
JavaScript

function process() {
  try {
    console.log("Trying...");
    // throw new Error("An error!");
    return "Returned from try";
  } catch (err) {
    console.log("Caught error");
    return "Returned from catch";
  } finally {
    console.log("Finally block executed"); // This will run before the return statement completes
```

```
  }
}
console.log(process()); // Logs: "Trying...", "Finally block executed", "Returned from try"
```

---

75. How to handle asynchronous errors in JavaScript?

- **Promises (`.catch()`)**: For promise-based code, you can chain a `.catch()` block to handle any rejections that occur in the promise or any preceding `.then()` blocks.
- JavaScript

```
fetch('invalid-url')
  .then(response => response.json())
  .catch(error => {
    console.error("Promise chain error:", error);
  });
```

-
-
- **`async/await` (`try...catch`)**: This is the modern and often cleaner way. You can wrap your `await` calls in a standard `try...catch` block.
- JavaScript

```
async function fetchData() {
  try {
    const response = await fetch('invalid-url');
    const data = await response.json();
  } catch (error) {
    console.error("Async/await error:", error);
  }
}
```

-
-

---

76. What is stack trace in JavaScript?

A **stack trace** is a report of the active stack frames at a certain point in time during the execution of a program. When an error is thrown, the stack trace shows the sequence of function calls that led to the error, from the most recent call to the least recent. This is extremely useful for debugging, as it helps you pinpoint where the error originated.

---

77. Difference between `console.log`, `console.error`, `console.warn`, `console.table`.

These are methods on the `console` object for logging information to the web console.

- `console.log()`: For general output of logging information.

- `console.warn()`: Outputs a warning message. Browsers often display this with a yellow icon.
- `console.error()`: Outputs an error message. Browsers often display this with a red icon and include a stack trace.
- `console.table()`: Displays tabular data as a table, which is very useful for logging arrays of objects.

Example:

```JavaScript
const users = [
  { id: 1, name: "Alice", role: "admin" },
  { id: 2, name: "Bob", role: "user" }
];
console.table(users);
```

---

78. What are debugging tools in Chrome DevTools?

Chrome DevTools is a powerful suite of tools for debugging web applications. Key features for JavaScript debugging include:

- **Sources Panel**: Allows you to view your source code, set breakpoints, step through code execution (step over, step into, step out), inspect variables, and view the call stack.
- **Console Panel**: For logging information, interacting with the page's JavaScript context, and viewing error messages.
- **Breakpoints**: Pause code execution at a specific line.
    - **Conditional Breakpoints**: Pause only when a certain condition is true.
    - **Logpoints**: Log a message to the console without pausing execution.
- **Watch Expressions**: Monitor the value of specific variables as you step through code.
- **Call Stack**: Shows the chain of function calls that led to the current point of execution.

---

79. How to detect memory leaks in JavaScript?

A memory leak occurs when a program allocates memory but fails to release it when it's no longer needed. In JavaScript, this usually happens when there are unintended references to objects, preventing the garbage collector from cleaning them up.

Tools for detection:

- **Chrome DevTools Memory Panel**:
    - **Heap Snapshot**: Takes a snapshot of the memory heap. You can compare snapshots taken at different times to see which objects are being created and not released.

    - **Allocation Timeline**: Shows memory allocations over time. Spikes can indicate potential leaks.
- **Common Causes**:
    - **Global Variables**: Accidental global variables that are never cleaned up.
    - **Closures**: Closures that hold references to large objects longer than necessary.
    - **Detached DOM Elements**: Keeping a reference in JavaScript to a DOM element that has been removed from the page.
    - **Event Listeners**: Forgetting to remove event listeners from elements that are removed from the DOM.

---

80. What is `debugger` statement in JavaScript?

The `debugger` statement invokes any available debugging functionality, such as setting a breakpoint. If a browser's developer tools are open, code execution will pause at the `debugger` statement, just as if you had manually set a breakpoint in the Sources panel. It's a useful way to programmatically pause execution for inspection.

Example:

```
JavaScript

function calculate(x) {
  if (x < 0) {
    debugger; // Execution will pause here if DevTools is open
  }
  return Math.sqrt(x);
}
```

---

9. JavaScript ES6+ Features

81. What are template literals in JavaScript?

Template literals are strings enclosed in backticks (`) instead of single or double quotes. They allow for:

- **String Interpolation**: Embedding expressions directly inside the string using `${expression}`.
- **Multi-line Strings**: You can create strings that span multiple lines without needing `\n`.

Example:

```
JavaScript

const name = "Alice";
```

```
const age = 30;

// Old way
const greetingOld = "Hello, my name is " + name + " and I am " + age + " years old.";

// New way with template literals
const greetingNew = `Hello, my name is ${name} and I am ${age} years old.
This is a new line.`;

console.log(greetingNew);
```

---

82. What is destructuring assignment?

Destructuring is a syntax that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Object Destructuring:

```
JavaScript

const person = { firstName: 'John', lastName: 'Doe', age: 42 };
const { firstName, age } = person; // Extracts firstName and age
console.log(firstName, age); // "John" 42
```

Array Destructuring:

```
JavaScript

const [first, second, , fourth] = [1, 2, 3, 4];
console.log(first, second, fourth); // 1 2 4 (skips the third element)
```

---

83. What is default parameter in functions?

Default parameters allow you to initialize named parameters with default values if no value or `undefined` is passed.

Example:

```
JavaScript

function greet(name = "Guest", greeting = "Hello") {
  console.log(`${greeting}, ${name}!`);
}

greet("Alice");    // "Hello, Alice!"
greet();           // "Hello, Guest!"
```

```
greet("Bob", "Hi"); // "Hi, Bob!"
```

---

84. What is rest operator (`...args`)?

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array. It must be the last parameter in a function definition.

Example:

```
JavaScript

function sum(...numbers) {
  // `numbers` is an array containing all arguments
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3));       // 6
console.log(sum(10, 20, 30, 40)); // 100
```

---

85. Difference between `for...of` and `for...in`.

This is a repeat of question 31, but important to reiterate.

- **`for...in`**: Iterates over the **enumerable property keys** of an object. Use it for plain objects, not arrays.
- **`for...of`**: Iterates over the **values** of an **iterable** object (like Array, String, Map, Set). This is the preferred way to loop over the values of an array.

---

86. What is optional chaining (`?.`)?

Optional chaining (`?.`) permits reading the value of a property located deep within a chain of connected objects without having to explicitly validate that each reference in the chain is valid. If a reference is `null` or `undefined`, the expression short-circuits and returns `undefined` instead of throwing an error.

Example:

```
JavaScript

const user = {
  name: 'Alice',
  // address is missing
};

// Without optional chaining (throws TypeError)
// const city = user.address.city;
```

```
// With optional chaining
const city = user?.address?.city;
console.log(city); // undefined (no error thrown)
```

---

87. What is nullish coalescing operator (??)?

The nullish coalescing operator (??) is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

It's a way to provide a default value for potentially nullish values, without falling back on falsy values like 0, '', or false (which the || operator would do).

Example:

```
JavaScript

const volume = 0;

// Using OR operator (incorrectly treats 0 as a falsy value to be replaced)
const setting1 = volume || 50;
console.log(setting1); // 50

// Using nullish coalescing operator (correctly keeps 0)
const setting2 = volume ?? 50;
console.log(setting2); // 0

const username = null;
const defaultUser = username ?? "Guest";
console.log(defaultUser); // "Guest"
```

---

88. What is dynamic import in JavaScript?

Dynamic import() is a function-like expression that allows you to load an ES module asynchronously and on-demand. Unlike static import, it can be used anywhere in your code (e.g., inside an if block or a function). It returns a Promise that resolves to the module object.

Use Case: Code splitting - loading parts of your application only when they are needed, which improves initial load performance.

Example:

```
JavaScript

const button = document.getElementById('loadModuleBtn');

button.addEventListener('click', async () => {
```

```
  try {
    const module = await import('./utils.js');
    module.doSomething();
  } catch (error) {
    console.error("Failed to load module", error);
  }
});
```

---

89. What are private fields in classes (#field)?

Private class fields are properties or methods that are only accessible from within the class they are defined in. You create them by prefixing their name with a hash #. This provides true encapsulation, preventing outside code from accessing or modifying internal state.

Example:

JavaScript

```
class Counter {
  #count = 0; // Private field

  increment() {
    this.#count++;
  }

  getCount() {
    return this.#count;
  }
}

const c = new Counter();
c.increment();
console.log(c.getCount()); // 1
// console.log(c.#count); // SyntaxError: Private field '#count' must be declared in an enclosing class
```

---

90. What are async iterators (for await...of)?

An async iterator is a protocol for iterating over asynchronous data sources, like data streamed from a server. The for await...of loop is used to iterate over these async iterables. Each iteration of the loop waits for a Promise returned by the iterator to resolve before proceeding.

Example:

JavaScript

```
const asyncDataStream = {
  [Symbol.asyncIterator]() {
    let i = 0;
    return {
      next() {
        if (i >= 3) {
          return Promise.resolve({ done: true });
        }
        return new Promise(resolve => setTimeout(() => {
          resolve({ value: `Data chunk ${++i}`, done: false });
        }, 1000));
      }
    };
  }
};

async function processStream() {
  for await (const chunk of asyncDataStream) {
    console.log(chunk); // Logs "Data chunk 1", then 2, then 3 (one per second)
  }
}

processStream();
```

This completes Section 9. I will now proceed to the final sections.

I will now complete the final sections of your interview question list.

---

10. JavaScript & Performance

91. How to optimize performance of JavaScript code?

Performance optimization is a broad topic, but key strategies include:

- **Reduce DOM Manipulation**: Minimize direct interactions with the DOM. Batch updates, use document fragments, or leverage virtual DOMs (in frameworks like React).
- **Use Efficient Loops**: Use `for` loops for performance-critical tasks, as they are often faster than `forEach` or `for...of`.
- **Debounce and Throttle**: Limit the rate at which event handlers are called for events like `scroll`, `resize`, and `input`.
- **Optimize Algorithms**: Choose the right data structures and algorithms (e.g., use `Map` for frequent lookups instead of searching an array).
- **Avoid Memory Leaks**: Clean up event listeners, timers, and references to removed DOM elements.

- **Code Splitting / Lazy Loading**: Use dynamic `import()` to load JavaScript only when it's needed.
- **Use Web Workers**: Offload heavy computations to a background thread to avoid blocking the main UI thread.
- **Minimize Network Requests**: Bundle assets, use caching, and reduce the size of payloads.

---

92. Difference between debounce and throttle?

This is a repeat of question 60, but it's crucial for performance discussions.

- **Debounce**: Executes a function only **after** a certain period of inactivity. If the event fires again within that period, the timer resets. **Use case**: A search bar input field; you only want to send the API request after the user has stopped typing.
- **Throttle**: Executes a function at most **once** every specified interval. **Use case**: A scroll event listener; you might want to check the scroll position, but not on every single pixel scrolled, maybe just every 200ms.

---

93. What is lazy loading in JavaScript?

Lazy loading is a design pattern for deferring the initialization of an object or resource until the point at which it is needed. In web development, it commonly refers to:

- **Image/Video Lazy Loading**: Loading images or videos only when they are about to enter the viewport. This is often achieved using the `IntersectionObserver` API or the `loading="lazy"` attribute on `<img>` tags.
- **Component/Module Lazy Loading (Code Splitting)**: Using dynamic `import()` to load JavaScript for a specific component or feature only when the user navigates to it or interacts with it. This reduces the initial bundle size and improves load times.

---

94. What is tree-shaking in JavaScript bundlers?

Tree-shaking is a form of dead code elimination. Modern JavaScript bundlers like Webpack, Rollup, and Vite use it to remove unused code from your final bundle. It works by statically analyzing your ES6 `import` and `export` statements. If you import a module but only use one function from it, a tree-shaking-enabled bundler will only include that one function in the final output, making the bundle smaller.

For it to be effective, you must use ES6 module syntax (`import`/`export`).

---

95. What is Web Workers in JavaScript?

Web Workers provide a way to run scripts in a **background thread**, separate from the main execution thread that handles the UI. This is useful for performing long-running or computationally intensive tasks without freezing the user interface and making the page unresponsive.

- Communication between the main thread and a worker is done via messages (`postMessage()` and the `onmessage` event handler).
- Workers do **not** have access to the DOM, `window`, or `document` objects.

Example (`main.js`):

```
JavaScript

const myWorker = new Worker('worker.js');

myWorker.postMessage([1, 2, 3, 4, 5]); // Send data to the worker

myWorker.onmessage = function(e) {
  console.log('Result from worker:', e.data); // Receive data from the worker
};
```

Example (`worker.js`):

```
JavaScript

self.onmessage = function(e) {
  const data = e.data;
  const result = data.reduce((acc, val) => acc + val, 0); // Heavy calculation
  self.postMessage(result); // Send result back
};
```

---

96. Difference between synchronous and asynchronous iteration.

- **Synchronous Iteration**: The standard iteration you are familiar with using `for...of`. It iterates over a data source where all the values are readily available in memory (e.g., an array). The loop proceeds from one item to the next without pausing.
- **Asynchronous Iteration**: Used for iterating over data sources that deliver their values asynchronously (e.g., data streamed over a network, lines read from a file). It uses the `for await...of` loop, which waits for each value (which is a `Promise`) to resolve before moving to the next iteration.

---

97. What are service workers? How do they work?

A **Service Worker** is a type of Web Worker that acts as a proxy between the web application, the browser, and the network. It runs in the background, separate from the web page, and can continue running even when the page is closed.

Key Capabilities & Use Cases:

- **Offline Support**: By intercepting network requests (`fetch` events), a service worker can serve cached assets, allowing an application to work offline.

- **Push Notifications**: Can receive push messages from a server and display notifications to the user.
- **Background Sync**: Can defer actions until the user has a stable internet connection.

It's a foundational technology for **Progressive Web Apps (PWAs)**.

---

98. What is IndexedDB in JavaScript?

IndexedDB is a low-level, transactional database system built into modern browsers for client-side storage of significant amounts of structured data. It's a key-value store that allows for indexing, which enables high-performance searches.

- It's an asynchronous API (originally event-based, but often used with promise-based wrappers).
- It's much more powerful than `localStorage`, supporting transactions, versioning, and storage of complex objects.
- Use cases: Caching application data for offline use, storing user-generated content before it's uploaded.

---

99. What is caching in browsers?

Browser caching is the process of storing local copies of web resources (like HTML, CSS, JS files, and images) to improve performance and reduce server load. When a user revisits a page, the browser can load the resources from its local cache instead of re-downloading them from the network.

Types of Caching:

- **HTTP Cache**: Controlled by HTTP headers like `Cache-Control`, `Expires`, and `ETag`. This is the most common form of browser caching.
- **Service Worker Cache**: Using the Cache API within a service worker gives developers fine-grained programmatic control over what is cached and how responses are served, enabling offline capabilities.
- **Memory Cache / Disk Cache**: The browser's internal mechanisms for storing resources.

---

100. How to improve JavaScript memory usage?

- **Avoid Global Variables**: Globals are never garbage collected unless the page is closed.
- **Use `let` and `const`**: Block-scoping helps variables go out of scope sooner, making them eligible for garbage collection.
- **Clean Up Event Listeners**: Remove event listeners from DOM elements when they are no longer needed, especially in Single Page Applications (SPAs). Use `removeEventListener()`.
- **Manage Closures**: Be mindful of large objects being kept alive by closures.

- **Use `WeakMap` and `WeakSet`**: To associate metadata with objects without preventing them from being garbage collected.
- **Optimize Data Structures**: Choose data structures that are appropriate for the task to avoid unnecessary memory overhead.

---

11. JavaScript Design Patterns

101. What is a Singleton pattern in JavaScript?

The **Singleton** pattern ensures that a class has only one instance and provides a global point of access to it. It's useful when exactly one object is needed to coordinate actions across the system (e.g., a configuration manager or a logger).

Example:

```
JavaScript

const Singleton = (function() {
  let instance;

  function createInstance() {
    const object = new Object("I am the instance");
    return object;
  }

  return {
    getInstance: function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

console.log(instance1 === instance2); // true
```

---

102. What is Module pattern?

The **Module** pattern is used to create private and public members (variables and functions) by leveraging closures and IIFEs. It helps in organizing code, avoiding global scope pollution, and creating encapsulated, reusable pieces of code. This was the standard before ES6 modules were introduced.

Example:

```
JavaScript
```

```javascript
const ShoppingCart = (function() {
  // Private members
  let items = [];

  // Public members
  return {
    addItem: function(item) {
      items.push(item);
    },
    getItemCount: function() {
      return items.length;
    },
    getTotal: function() {
      return items.reduce((acc, item) => acc + item.price, 0);
    }
  };
})();

ShoppingCart.addItem({ name: 'Book', price: 20 });
console.log(ShoppingCart.getItemCount()); // 1
// console.log(ShoppingCart.items); // undefined (private)
```

---

103. What is Factory pattern?

The **Factory** pattern provides a generic interface for creating objects, where we can let subclasses decide which class to instantiate. It's used when we want to create different objects based on some condition, without exposing the creation logic to the client.

Example:

```
JavaScript
```

```javascript
function EmployeeFactory() {
  this.create = function(name, type) {
    let employee;
    if (type === 'fulltime') {
      employee = new FullTime(name);
    } else if (type === 'parttime') {
      employee = new PartTime(name);
    }
    // ... other types
    return employee;
  };
}
```

```
function FullTime(name) { this.name = name; this.rate = 15; }
function PartTime(name) { this.name = name; this.rate = 10; }

const factory = new EmployeeFactory();
const employees = [];
employees.push(factory.create('John Doe', 'fulltime'));
employees.push(factory.create('Jane Smith', 'parttime'));
```

---

104. What is Observer/Publisher-Subscriber pattern?

The **Observer** (or Pub/Sub) pattern defines a one-to-many dependency between objects. When one object (the "subject" or "publisher") changes its state, all of its dependents ("observers" or "subscribers") are notified and updated automatically.

This pattern is heavily used in event-driven systems.

Example:

JavaScript

```
class Subject { // Publisher
  constructor() { this.observers = []; }
  subscribe(observer) { this.observers.push(observer); }
  unsubscribe(observer) { /* ... remove observer ... */ }
  notify(data) {
    this.observers.forEach(observer => observer.update(data));
  }
}

class Observer { // Subscriber
  update(data) {
    console.log('Received new data:', data);
  }
}

const subject = new Subject();
const observer1 = new Observer();
const observer2 = new Observer();

subject.subscribe(observer1);
subject.subscribe(observer2);

subject.notify('New sale announced!'); // Both observers will receive this notification
```

I will continue with the final questions.

I'm now providing the answers for the final set of questions.

---

12. JavaScript in Real-World & Frameworks

111. Difference between JavaScript and TypeScript?

TypeScript (TS) is a **superset** of JavaScript. This means all valid JavaScript code is also valid TypeScript code. TypeScript adds **static types** to JavaScript, which allows you to catch type-related errors during development (at "compile time") rather than at runtime.

Feature	JavaScript	TypeScript
Typing	**Dynamic**: Types are checked at runtime.	**Static**: Types are checked at compile time.
Tooling	Standard JS tooling.	Richer tooling: better autocompletion, refactoring, and error checking in IDEs.
Learning Curve	Easier to start for beginners.	Steeper curve due to learning the type system.
Codebase	Prone to runtime errors in large applications.	More robust and maintainable for large-scale projects.
Compilation	Not needed. Runs directly in the browser/Node.js.	Must be **transpiled** to plain JavaScript before it can be executed.

Example (TypeScript):

```
TypeScript

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

add(5, 10); // Works
// add(5, "10"); // Compile-time error: Argument of type 'string' is not assignable to parameter of type 'number'.
```

---

112. What is transpiling (Babel)?

Transpiling is the process of converting source code written in one language (or a newer version of a language) into equivalent source code in another language (or an older version).

Babel is a popular JavaScript transpiler. Its primary use is to convert modern JavaScript code (ES6+) into a backward-compatible version (like ES5) that can run in older browsers or environments that don't support the latest features. It allows developers to use new language features without worrying about compatibility.

---

113. What is bundling (Webpack, Parcel, Vite)?

Bundling is the process of taking your JavaScript modules (and other assets like CSS and images) and combining them into one or more optimized files (bundles) for the browser.

Why bundle?

- **Reduces HTTP Requests**: Loading one large file is often faster than loading many small files.
- **Optimization**: Bundlers can perform optimizations like minification (removing whitespace/comments), tree-shaking (removing unused code), and code splitting.
- **Module Resolution**: They handle `import`/`export` statements, which browsers didn't natively support for a long time.

Popular bundlers include **Webpack** (highly configurable), **Parcel** (zero-configuration), and **Vite** (extremely fast due to its use of native ES modules during development).

---

114. Difference between Node.js and browser JavaScript?

While both use the V8 JavaScript engine, they run in different environments and have different capabilities.

Feature	Browser JavaScript	Node.js
Environment	Runs in a web browser.	Runs on a server.
Global Object	`window`	`global`

APIs	Access to **DOM APIs** (`document`, `window`) for UI manipulation and **Web APIs** (`fetch`, `localStorage`).	Access to **backend APIs** for file system (`fs`), networking (`http`), OS, etc. No access to DOM.
Modules	Primarily uses ES Modules (`import`/`export`).	Historically used CommonJS (`require`), but now also supports ES Modules.
Purpose	Client-side scripting, user interaction, DOM manipulation.	Server-side applications, APIs, build tools, scripting.

---

115. What are Single Page Applications (SPAs)?

A **Single Page Application** is a web application that interacts with the user by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages. The page does not reload during use; instead, content is fetched and updated as needed.

- **Pros**: Faster, more fluid user experience (like a desktop app), can work offline with service workers.
- **Cons**: Can have a slower initial load time, more complex to manage state, can be challenging for SEO without solutions like Server-Side Rendering.
- **Frameworks**: React, Angular, and Vue are commonly used to build SPAs.

---

116. What is Virtual DOM?

The **Virtual DOM (VDOM)** is a programming concept where a virtual representation of a UI is kept in memory and synced with the "real" DOM. It's a performance optimization technique used by frameworks like React and Vue.

How it works:

1. When the state of your application changes, a new Virtual DOM tree is created.
2. This new tree is compared with the previous Virtual DOM tree (a process called "diffing").
3. The framework calculates the most efficient way to make these changes to the real DOM (e.g., changing only the text of one element instead of re-creating the entire element).
4. These minimal changes are then "batched" and applied to the real DOM.

This process is faster than manipulating the real DOM directly and frequently, as DOM operations are slow.

---

117. Difference between React, Angular, and Vue in terms of JavaScript usage?

- **Angular**:
  - A full-fledged **framework** (opinionated).
  - Uses **TypeScript** exclusively.
  - Implements a more traditional object-oriented approach with dependency injection, services, and modules.
  - Uses a real DOM but employs change detection strategies to optimize updates.
- **React**:
  - A **library** for building user interfaces (less opinionated).
  - Primarily uses **JavaScript (with JSX)**, but often used with TypeScript.
  - Promotes a **functional programming** style with concepts like components, props, state, and hooks.
  - Uses a **Virtual DOM** for efficient rendering.
- **Vue**:
  - A **progressive framework** (can be adopted incrementally).
  - Can be used with either **JavaScript or TypeScript**.
  - Combines ideas from both Angular (templates) and React (component-based architecture). It's often seen as easier to learn.
  - Uses a **Virtual DOM**.

---

118. What is SSR (Server-Side Rendering) in JavaScript frameworks?

Server-Side Rendering (SSR) is the process of rendering a normally client-side-only single-page application on the server and then sending a fully rendered HTML page to the client.

Benefits:

- **Faster First Contentful Paint (FCP)**: The user sees content immediately, as the browser doesn't have to wait for JavaScript to download and execute to render the initial view.
- **Improved SEO**: Search engine crawlers can easily index the page because the content is present in the initial HTML response.

Frameworks like **Next.js** (for React) and **Nuxt.js** (for Vue) are popular for implementing SSR.

---

119. What is hydration in React/Next.js?

Hydration is the process that happens after an SSR page is delivered to the browser. The server sends a static HTML page, and once the client-side JavaScript loads, React

"hydrates" the static markup by attaching event listeners and making the page interactive. It essentially "brings to life" the server-rendered HTML, turning it into a fully functional client-side application without re-rendering everything from scratch.

---

120. What are micro-frontends?

Micro-frontends is an architectural style where a web application is broken down into smaller, independent, and deployable frontend applications. Each micro-frontend can be developed, tested, and deployed independently by a different team, often using different technology stacks.

This approach is analogous to microservices on the backend. It helps manage complexity in large, enterprise-scale applications by allowing teams to work autonomously. The final application is composed by integrating these individual pieces together at runtime or build time.