JavaScript Interview Notes

1. Hoisting

- **Definition**: JavaScript's behavior of moving variable and function declarations to the top of their scope during compilation
- **Variables**: `var` declarations are hoisted but initialized as `undefined`
- **Functions**: Function declarations are fully hoisted (can be called before declaration)
- **let/const**: Hoisted but in "temporal dead zone" until declaration line

```javascript
console.log(x); // undefined (not error)
var x = 5;

// Function hoisting
sayHello(); // Works fine
function sayHello() { console.log("Hello"); }
```

2. DOM (Document Object Model)

- **Definition**: Programming interface for HTML documents
- **Structure**: Tree-like representation of HTML elements
- **Access Methods**:
  - `getElementById()`, `querySelector()`, `getElementsByClassName()`
- **Manipulation**: Change content, attributes, styles dynamically
- **Not part of JavaScript** but provides API to interact with web pages

3. DOM Events

- **Definition**: Actions that occur in the browser (clicks, key presses, page loads)
- **Event Handling**: `addEventListener()`, `removeEventListener()`
- **Event Object**: Contains information about the event
- **Event Phases**: Capturing → Target → Bubbling
- **preventDefault()**: Stops default browser behavior
- **stopPropagation()**: Stops event bubbling

```javascript
```

```
button.addEventListener('click', function(event) {
    event.preventDefault();
    console.log('Button clicked');
});
```

4. var vs let vs const

Feature	var	let	const
Scope	Function/Global	Block	Block
Hoisting	Yes (undefined)	Yes (TDZ)	Yes (TDZ)
Redeclaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	Not allowed
Initialization	Optional	Optional	Required

```javascript
var a = 1; // Function scoped
let b = 2; // Block scoped
const c = 3; // Block scoped, immutable
```

5. for...of

- **Purpose**: Iterates over iterable objects (arrays, strings, maps, sets)
- **Returns**: Values of the iterable
- **Syntax**: `for (variable of iterable)`

```javascript
const arr = [1, 2, 3];
for (const value of arr) {
    console.log(value); // 1, 2, 3
}
```

6. for...in

- **Purpose**: Iterates over enumerable properties of objects
- **Returns**: Property keys/indices
- **Syntax**: `for (variable in object)`

```javascript
```

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

7. Named vs Anonymous Functions

Named Function:

```javascript
function greet() { // Has a name
    console.log("Hello");
}
```

Anonymous Function:

```javascript
const greet = function() { // No name
    console.log("Hello");
};
```

- **Named**: Better debugging, can call itself (recursion)
- **Anonymous**: Often used as callbacks, cleaner syntax

8. Arrow Functions

- **Syntax**: `(params) => expression` or `(params) => { statements }`
- **No `this` binding**: Inherits `this` from enclosing scope
- **No `arguments` object**
- **Cannot be used as constructors**
- **Shorter syntax for simple functions**

```javascript
const add = (a, b) => a + b;
const multiply = (a, b) => {
    return a * b;
};
```

9. Objects and JSON

Objects:

- JavaScript data structure with key-value pairs
- Keys can be strings or symbols
- Values can be any data type

JSON (JavaScript Object Notation):

- Text-based data format
- Subset of JavaScript object syntax
- Keys must be strings in double quotes
- Limited data types (string, number, boolean, null, object, array)

```javascript
// Object
const obj = {name: "John", age: 30};

// JSON
const json = '{"name": "John", "age": 30}';
JSON.parse(json); // Convert JSON to object
JSON.stringify(obj); // Convert object to JSON
```

10. Primitive vs Non-Primitive Data Types

Primitive (Pass by Value):

- string, number, boolean, undefined, null, symbol, bigint
- Stored directly in variable
- Immutable

Non-Primitive (Pass by Reference):

- Objects, arrays, functions
- Variable stores reference to memory location
- Mutable

```javascript
```

```javascript
// Primitive
let a = 5;
let b = a; // b gets copy of value
a = 10; // b is still 5

// Non-primitive
let obj1 = {x: 5};
let obj2 = obj1; // obj2 gets reference
obj1.x = 10; // obj2.x is also 10
```

11. Closures

- **Definition**: Function that has access to outer function's variables even after outer function returns
- **Created**: When inner function is defined inside outer function
- **Use Cases**: Data privacy, function factories, callbacks

```javascript
function outer(x) {
    return function inner(y) {
        return x + y; // Access to 'x' even after outer() returns
    };
}
const addFive = outer(5);
console.log(addFive(3)); // 8
```

12. Currying

- **Definition**: Technique of transforming function with multiple arguments into sequence of functions with single argument
- **Purpose**: Partial application, function reusability

```javascript
```

```javascript
// Normal function
function add(a, b, c) {
    return a + b + c;
}

// Curried function
function curriedAdd(a) {
    return function(b) {
        return function(c) {
            return a + b + c;
        };
    };
}
// Usage: curriedAdd(1)(2)(3) = 6
```

13. Callbacks

- **Definition**: Function passed as argument to another function
- **Purpose**: Execute code after asynchronous operation completes
- **Problem**: Callback hell (nested callbacks)

javascript

```javascript
function fetchData(callback) {
    setTimeout(() => {
        callback("Data received");
    }, 1000);
}

fetchData(function(data) {
    console.log(data); // "Data received"
});
```

14. Promises

- **Definition**: Object representing eventual completion/failure of asynchronous operation
- **States**: Pending, Fulfilled, Rejected
- **Methods**: `.then()`, `.catch()`, `.finally()`
- **Static Methods**: `Promise.all()`, `Promise.race()`, `Promise.resolve()`

javascript

```javascript
```

```javascript
const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Success"), 1000);
});

promise
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

15. Async/Await

- **Definition**: Syntactic sugar for working with Promises
- **async**: Declares asynchronous function (returns Promise)
- **await**: Pauses execution until Promise resolves
- **Error Handling**: Use try-catch blocks

```javascript
async function fetchData() {
    try {
        const response = await fetch('/api/data');
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error:', error);
    }
}
```

16. ES6 (ECMAScript 2015)

Key Features:

- let/const keywords
- Arrow functions
- Template literals
- Destructuring
- Classes
- Modules (import/export)
- Default parameters
- Rest/spread operators
- Promises
- Map and Set

17. Spread and Rest Operators

Spread (`...`): Expands iterables

```javascript
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]
```

Rest (`...`): Collects multiple elements

```javascript
function sum(...numbers) {
    return numbers.reduce((a, b) => a + b, 0);
}
sum(1, 2, 3, 4); // 10
```

18. Export and Import

Export:

```javascript
// Named exports
export const name = "John";
export function greet() {}

// Default export
export default class User {}
```

Import:

```javascript
// Named imports
import { name, greet } from './module.js';

// Default import
import User from './module.js';

// Import all
import * as utils from './module.js';
```

19. Shallow Copy vs Deep Copy

Shallow Copy:

- Copies only first level properties
- Nested objects still share references

```javascript
const original = {a: 1, b: {c: 2}};
const shallow = {...original}; // or Object.assign({}, original)
shallow.b.c = 3; // Changes original.b.c too
```

Deep Copy:

- Copies all levels recursively
- Complete independence

```javascript
const deep = JSON.parse(JSON.stringify(original));
// or use libraries like Lodash cloneDeep()
```

20. Destructuring

Array Destructuring:

```javascript
const [first, second, ...rest] = [1, 2, 3, 4, 5];
// first = 1, second = 2, rest = [3, 4, 5]
```

Object Destructuring:

```javascript
const {name, age, city = "Unknown"} = {name: "John", age: 30};
// name = "John", age = 30, city = "Unknown"
```

21. Prototype

- **Definition**: Mechanism by which JavaScript objects inherit features from one another
- **Every object has prototype**: Accessible via `__proto__` or `Object.getPrototypeOf()`
- **Constructor functions**: Have `prototype` property
- **Prototype Chain**: Objects inherit from their prototype's prototype

```javascript
```

```javascript
function Person(name) {
    this.name = name;
}

Person.prototype.greet = function() {
    return `Hello, I'm ${this.name}`;
};

const john = new Person("John");
console.log(john.greet()); // "Hello, I'm John"
```

22. How to Submit a Form Through JavaScript?

Multiple Methods:

1. Using form.submit():

```javascript
const form = document.getElementById('myForm');
form.submit(); // Submits form directly
```

2. Using FormData and fetch():

```javascript
const form = document.getElementById('myForm');
const formData = new FormData(form);

fetch('/submit', {
    method: 'POST',
    body: formData
})
.then(response => response.json())
.then(data => console.log(data));
```

3. Using XMLHttpRequest:

```javascript
```

```javascript
const form = document.getElementById('myForm');
const formData = new FormData(form);
const xhr = new XMLHttpRequest();

xhr.open('POST', '/submit');
xhr.send(formData);
```

4. Programmatically creating and submitting:

```javascript
const form = document.createElement('form');
form.method = 'POST';
form.action = '/submit';

const input = document.createElement('input');
input.name = 'username';
input.value = 'john';
form.appendChild(input);

document.body.appendChild(form);
form.submit();
```

23. Can We Create a Form Inside Another Form?

Answer: NO

- **HTML Specification**: Nested forms are **not allowed** in HTML
- **Browser Behavior**: Inner form will be ignored or moved outside
- **Invalid HTML**: Will not validate according to HTML standards

Example of Invalid HTML:

```html
<!-- This is INVALID -->
<form action="/outer">
  <input name="field1">
  <form action="/inner">  <!-- This will be ignored -->
    <input name="field2">
  </form>
</form>
```

Alternatives:

1. **Use fieldsets** for grouping form elements
2. **Separate forms** placed adjacent to each other
3. **Use divs** for visual grouping with single form
4. **JavaScript handling** for complex form logic

```html
<!-- Correct approach -->
<form action="/submit">
  <fieldset>
    <legend>Personal Info</legend>
    <input name="name">
  </fieldset>
  <fieldset>
    <legend>Contact Info</legend>
    <input name="email">
  </fieldset>
</form>
```

24. In JavaScript, String is Mutable or Immutable?

Answer: IMMUTABLE

Key Points:

- **Primitive data type**: Strings are primitive values
- **Cannot be changed**: Original string remains unchanged
- **New string created**: String operations return new strings
- **Memory efficiency**: JavaScript optimizes string storage

Examples:

```javascript
```

```javascript
// String immutability demonstration
let str = "Hello";
str.toUpperCase(); // Returns "HELLO" but doesn't change str
console.log(str); // Still "Hello"

// Assignment creates new string
let original = "JavaScript";
let modified = original.replace("Script", "");
console.log(original); // "JavaScript" (unchanged)
console.log(modified); // "Java" (new string)

// Even concatenation creates new strings
let a = "Hello";
let b = a + " World";
console.log(a); // "Hello" (unchanged)
console.log(b); // "Hello World" (new string)

// Index assignment doesn't work
let text = "Hello";
text[0] = "h"; // This doesn't change the string
console.log(text); // Still "Hello"
```

Why Strings are Immutable:

- **Security**: Prevents accidental modifications
- **Performance**: Enables string interning and caching
- **Predictability**: Functions can't unexpectedly modify string arguments
- **Thread safety**: Multiple references to same string are safe

Working with Immutable Strings:

javascript

```javascript
// Wrong approach (inefficient)
let result = "";
for (let i = 0; i < 1000; i++) {
    result += "a"; // Creates new string each time
}

// Better approach
let parts = [];
for (let i = 0; i < 1000; i++) {
    parts.push("a");
}
let result = parts.join(""); // Single string creation
```

25. Essential Array Methods (6 Most Important)

1. map()

- **Purpose**: Creates new array by transforming each element
- **Returns**: New array with same length
- **Doesn't modify**: Original array

```javascript
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8]
console.log(numbers); // [1, 2, 3, 4] (unchanged)
```

2. filter()

- **Purpose**: Creates new array with elements that pass a test
- **Returns**: New array (can be shorter than original)
- **Use case**: Remove unwanted elements

```javascript
const ages = [12, 16, 18, 25, 30];
const adults = ages.filter(age => age >= 18);
console.log(adults); // [18, 25, 30]

// Filter objects
const users = [{name: "John", active: true}, {name: "Jane", active: false}];
const activeUsers = users.filter(user => user.active);
```

3. reduce()

- **Purpose**: Reduces array to single value
- **Parameters**: callback(accumulator, currentValue, index, array), initialValue
- **Use cases**: Sum, finding max/min, grouping, flattening

```javascript
```

```javascript
// Sum of numbers
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // 10

// Find maximum
const max = numbers.reduce((acc, num) => Math.max(acc, num));

// Group by property
const people = [{name: "John", age: 25}, {name: "Jane", age: 25}];
const groupedByAge = people.reduce((acc, person) => {
    acc[person.age] = acc[person.age] || [];
    acc[person.age].push(person);
    return acc;
}, {});
```

4. forEach()

- **Purpose**: Executes function for each array element
- **Returns**: undefined (doesn't return new array)
- **Use case**: Side effects, logging, DOM manipulation

```javascript
const fruits = ['apple', 'banana', 'orange'];
fruits.forEach((fruit, index) => {
    console.log(`${index}: ${fruit}`);
});
// 0: apple
// 1: banana
// 2: orange

// Cannot break out of forEach (use for...of instead)
```

5. find()

- **Purpose**: Returns first element that matches condition
- **Returns**: Element or undefined
- **Stops**: As soon as match is found

```javascript
```

```javascript
const users = [
    {id: 1, name: "John"},
    {id: 2, name: "Jane"},
    {id: 3, name: "Bob"}
];

const user = users.find(u => u.id === 2);
console.log(user); // {id: 2, name: "Jane"}

const notFound = users.find(u => u.id === 5);
console.log(notFound); // undefined
```

6. includes()

- **Purpose**: Checks if array contains specific element
- **Returns**: Boolean (true/false)
- **Comparison**: Uses strict equality (===)

```javascript
const fruits = ['apple', 'banana', 'orange'];
console.log(fruits.includes('banana')); // true
console.log(fruits.includes('grape')); // false

// Case sensitive
console.log(fruits.includes('Apple')); // false

// With objects (checks reference, not content)
const obj = {name: "John"};
const arr = [obj];
console.log(arr.includes(obj)); // true
console.log(arr.includes({name: "John"})); // false (different reference)
```

Bonus: Array Method Chaining

```javascript
const numbers = [1, 2, 3, 4, 5, 6];
const result = numbers
    .filter(n => n % 2 === 0)  // [2, 4, 6]
    .map(n => n * 2)           // [4, 8, 12]
    .reduce((acc, n) => acc + n, 0); // 24

console.log(result); // 24
```

Quick Reference:

- **Mutating**: push(), pop(), shift(), unshift(), splice(), sort(), reverse()
- **Non-mutating**: map(), filter(), reduce(), forEach(), find(), includes(), slice(), concat()
- **Remember**: Always check if method returns new array or modifies original

---

Quick Tips for Interviews:

1. **Practice coding examples** for each concept
2. **Understand the "why"** behind each feature
3. **Know common pitfalls** and how to avoid them
4. **Be able to explain** concepts in simple terms
5. **Prepare real-world examples** of when you'd use each feature