FE-Interview Questions

# Commonly Asked

* **Tree Shaking:**
  + Process used in modern build tools like webpack to eliminate unused codes from final bundle.
  + Reduces size of js files that the browser needs to execute hence improves **performance**.
* **How to ensure web application is accessible**
* Follow WCAG (Web content accessibility guidelines) guidelines
* Use semantic HTML for structure and aria roles (ARIA roles provide semantic meaning to content, allowing screen readers and other tools to present and support interaction with an object in a way that is consistent with user expectations of that type of object. ARIA roles can be used to describe elements that don't natively exist in HTML or exist but don't yet have full browser support.)
* Ensure keyboard navigation
* Provide alt text for image
* **Event Delegation**
* Method in which instead of adding event listeners to each child add listener to parent
* Leverages event bubbling to catch events from child elements.
* Reduced memory usage, dynamically handling events from elements added after the initial page load
* **SEO Challenges while using in single page application**
* Content is dynamically loaded, making it hard for search engine crawlers to index.
* Using history api to update URL for different views.
* Using SSR
* **CORS**
* Cross-Origin Resource Sharing (CORS) is a World Wide Web Consortium (W3C) standard that allows web applications to access resources from a different domain than the one that served the web page
* **Difference between. then () and async**
* . then () is used with promises for asynchronous operation, chaining multiple calls for sequential execution.
* **async** makes asynchronous calls looks synchronous making it readable and improves error handling
* **Closure in JS**
* Feature where inner function has accessibility to the outer function variables.
* **Flexbox VS Grid**

|  |  |
| --- | --- |
| **Flexbox** | **Grid** |
| 1-D layout method ideal for arranging items in single row or column offering control over alignment and spacing. | 2-D layout perfect for complex layouts. aligning contents with rows and columns. |
| Used for simple linear layout | Complex multi-dimensional layout |

* **Security in frontend**
* Implement content security policy (added layer of security that helps to detect and mitigate certain types of attacks, including Cross-Site Scripting ([XSS](https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting)) and data injection attacks.) can be implemented by using <meta>. Ex:

<meta

http-equiv="Content-Security-Policy"

content="default-src 'self'; img-src https://\*; child-src 'none';" />

* Use https for secure communication
* Store sensitivity data securely
* Keep dependencies UpToDate to avoid treats
* **Server-side rendering vs client-side rendering**

|  |  |
| --- | --- |
| **SSR** | **CSR** |
| Generates HTML on the server-side and send it to the client improving SEO and initial load | Renders pages directly in browsers using js results in faster subsequent page load and a smoother user experience |
| Use for static sites or SEO is priority | Use for dynamic app-like experience |

* **Object.freeze and const**
* Both are different concepts.
* Const applies to binding variables. It creates immutable binding.
* Object. freeze works on object values. It makes objects immutable i.e.; changing its property is not possible.
* **Importance of “Use strict”**
* Literal expression which enforces a stricter parsing and error handling in js code files or functions during runtime.
* Makes code easy to manage
* **Temporal dead zone**

The period between entering the scope and being declared is the one when these keywords are not able to accessed.

* **Currying in JS**

Advanced technique of working with functions that can accepts multiple arguments. It will transform this function into a series of functions, where every function will accept one argument.

* **ES6 Features**
* Let and const keywords (added difference between var, const and let)
* Block scoped literals
* Whereas, Var is Globally scoped or block scoped.
* Var can be updated and redeclared within its scope but let cannot be redeclared but updated within its scope.
* Var can be hoisted and initialized as undefined. But let and const cannot be.
* Arrow functions
* Multiline string
* Destructuring assignment
* Enhanced object literals
* Promises

## Array Methods

1. **Array.map()**

Returns a new array with the results of calling a provided function on every element in this array.

// Code

const list = [1, 2, 3, 4];

list.map((el) => el \* 2); // [2, 4, 6, 8]

1. **Array.filter()**

Returns a new array with all elements that pass the test implemented by the provided function.

const list = [1, 2, 3, 4];

list.filter((el) => el % 2 === 0); // [2, 4]

1. **Array.reduce()**

Reduce the array to a single value. The value returned by the function is stored in an accumulator (result/total).

const list = [1, 2, 3, 4, 5];

list.reduce((total, item) => total + item, 0); // 15

1. **Array.reduceRight()**

Executes a reducer function (that you provide) on each element of the array resulting in a single output value (from right to left).

// Code

const list = [1, 2, 3, 4, 5];

list.reduceRight((total, item) => total + item, 0); // 15

1. **Array.fill()**

Fill the elements in an array with a static value.

// Code

const list = [1, 2, 3, 4, 5];

list.fill(0); // [0, 0, 0, 0, 0]

1. **.find()**

Returns the value of the first element in the array that satisfies the provided testing function. Otherwise undefined is returned.

// Code

const list = [1, 2, 3, 4, 5];

list.find((el) => el === 3); // 3

list.find((el) => el === 6); // undefined

1. **.indexOf()**

Returns the first index at which a given element can be found in the array, or -1 if it is not present.

// Code

const list = [1, 2, 3, 4, 5];

list.indexOf(3); // 2

list.indexOf(6); // -1

1. **.lastIndexOf()**

Returns the last index at which a given element can be found in the array, or -1 if it is not present. The array is searched backwards, starting at fromIndex.

// Code

const list = [1, 2, 3, 4, 5];

list.lastIndexOf(3); // 2

list.lastIndexOf(3, 1); // -1

1. **.findIndexOf()**

Returns the index of the first element in the array that satisfies the provided testing function. Otherwise -1 is returned.

const array = [5, 12, 8, 130, 44];

array.findIndex((element) => element > 13); // 3

1. **.includes()**

Returns true if the given element is present in the array.

// Code

const list = [1, 2, 3, 4, 5];

list.includes(3); // true

list.includes(6); // false

1. **.pop()**

Removes the last element from an array and returns that element.

// Code

const list = [1, 2, 3, 4, 5];

list.pop(); // 5

list; // [1, 2, 3, 4]

1. **.push()**

Appends new elements to the end of an array, and returns the new length.

// Code

const list = [1, 2, 3, 4, 5];

list.push(6); // 6

list; // [1, 2, 3, 4, 5, 6]

1. **.shift**

Removes the first element from an array and returns that element.

// Code

const list = [1, 2, 3, 4, 5];

list.shift(); // 1

list; // [2, 3, 4, 5]

1. **.unshift()**

Adds new elements to the beginning of an array, and returns the new length.

// Code

const list = [1, 2, 3, 4, 5];

list.unshift(0); // 6

list; // [0, 1, 2, 3, 4, 5]

1. **.splice()**

Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

// Code

const list = [1, 2, 3, 4, 5];

list.splice(1, 2); // [2, 3]

list; // [1, 4, 5]

1. **.slice()**

Returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included). The original array will not be modified.

// Code

const list = [1, 2, 3, 4, 5];

list.slice(1, 3); // [2, 3]

list; // [1, 2, 3, 4, 5]

1. **.join()**

Joins all elements of an array into a string.

// Code

const list = [1, 2, 3, 4, 5];

list.join(', '); // "1, 2, 3, 4, 5"

1. **.reverse()**

Reverses the order of the elements in an array.

// Code

const list = [1, 2, 3, 4, 5];

list.reverse(); // [5, 4, 3, 2, 1]

list; // [5, 4, 3, 2, 1]

1. **.sort()**

Sorts the elements of an array in place and returns the array. The default sort order is according to string Unicode code points.

const array = ['D', 'B', 'A', 'C'];

array.sort(); // 😀 ['A', 'B', 'C', 'D']

// OR

const array = [4, 1, 3, 2, 10];

array.sort(); // 😧 [1, 10, 2, 3, 4]

array.sort((a, b) => a - b); // 😀 [1, 2, 3, 4, 10]

1. **.some()**

Returns true if at least one element in the array passes the test implemented by the provided function.

// Code

const list = [1, 2, 3, 4, 5];

list.some((el) => el === 3); // true

list.some((el) => el === 6); // false

1. **.every()**

Returns true if all elements in the array pass the test implemented by the provided function.

// Code

const list = [1, 2, 3, 4, 5];

list.every((el) => el === 3); // false

const list = [2, 4, 6, 8, 10];

list.every((el) => el%2 === 0); // true

1. **.from()**

Creates a new array from an array-like or iterable object.

const range = (n) => Array.from({ length: n }, (\_, i) => i + 1);

console.log(range(10)); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

1. **.of()**

Creates a new array with a variable number of arguments, regardless of number or type of the arguments.

// Code

const list = Array.of(1, 2, 3, 4, 5);

list; // [1, 2, 3, 4, 5]

1. **.isArray()**

Returns true if the given value is an array.

// Code

Array.isArray([1, 2, 3, 4, 5]); // true

Array.isArray(5); // false

1. **.at()**

Returns a value at the specified index.

// Code

const list = [1, 2, 3, 4, 5];

list.at(1); // 2

list.at(-1); // 5

list.at(-2); // 4

1. **.copyWithin()**

Copies array elements within the array. Returns the modified array.

// Code

const list = [1, 2, 3, 4, 5];

list.copyWithin(0, 3, 4); // [4, 2, 3, 4, 5]

* first argument is the target at which to start copying elements from.
* second argument is the index at which to start copying elements from.
* third argument is the index at which to stop copying elements from.

1. **.flat()**

Returns a new array with all sub-array elements concatenated into it recursively up to the specified depth.

// Code

const list = [1, 2, [3, 4, [5, 6]]];

list.flat(Infinity); // [1, 2, 3, 4, 5, 6]

1. **.flatMap()**

Returns a new array formed by applying a given callback function to each element of the array,

// Code

const list = [1, 2, 3];

list.flatMap((el) => [el, el \* el]); // [1, 1, 2, 4, 3, 9]

1. **concat()**

The concat method is used to join two or more arrays. It does not change the existing arrays but returns a new array, containing the values of the joined array.

const arr1 = [1,2,3];

const arr2 = [4,5,6];

const arr3 = arr1.concat(arr2);

console.log(arr3); //[1,2,3,4,5,6]

1. **.forEach()**

The forEach method helps to loop over an array by executing a provided callback function for each element in an array.

const numbers = [1, 2, 3];

numbers.forEach((ele) => {

console.log(ele);

});

1. .**toString()**

This method converts the elements of a specified array into string form, without affecting the original array.

const numbers = [1, 2, 3, 4, 5];

numbers.toString(); //'1,2,3,4,5'

## Data Structures

* **Center-Aligned Pascal's Triangle in JavaScript**

function generatePascalsTriangle(numRows) {

if (numRows === 0) return [];

const triangle = Array(numRows).fill().map((\_, rowIndex) => {

if (rowIndex === 0) return [1];

const prevRow = triangle[rowIndex - 1];

return [1, ...prevRow.slice(1).map((val, i) => val + prevRow[i]), 1];

});

return triangle;

}

function printPascalsTriangle(triangle) {

const numRows = triangle.length;

const maxRowLength = triangle[numRows - 1].length;

triangle.forEach(row => {

const padding = ' '.repeat((maxRowLength - row.length) \* 2);

console.log(padding + row.join(' '));

});

}

// Example usage:

const numRows = 5;

const pascalsTriangle = generatePascalsTriangle(numRows);

printPascalsTriangle(pascalsTriangle);

output:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

* **LEFT aligned pascals triangle with array**

function generatePascalsTriangle(numRows) {

if (numRows === 0) return [];

const triangle = Array(numRows).fill().map((\_, rowIndex) => {

if (rowIndex === 0) return [1];

const prevRow = triangle[rowIndex - 1];

return [1, ...prevRow.slice(1).map((val, i) => val + prevRow[i]), 1];

});

return triangle;

}

// Example usage:

const numRows = 5;

const pascalsTriangle = generatePascalsTriangle(numRows);

console.log(pascalsTriangle);

output:

[

[1],

[1, 1],

[1, 2, 1],

[1, 3, 3, 1],

[1, 4, 6, 4, 1]

]

* **Fibinocci series:**

function generateFibibnocci(){

if(n<=0) return[]

if(n===0) return[0]

if(n===2) return[0,1]

return Array(n-2).fill().reduce((acc)=>

{

const newVal = acc[acc.length-1]+acc[acc.length-2];

return [...acc,newVal]},[0,1])

}

const n = 10;

const fibonacciSeries = generateFibibnocci(n);

console.log(fibonacciSeries);

//output

// [object Array] (10)

[0,1,1,2,3,5,8,13,21,34]

* **Reverse String**

function reverseString(revString){

return revString.split("").reverse().join("")

}

const rev= reverseString("fdgdfgdfh")

console.log("i am reverse string",rev)

* **Check fibinocci or not**

function isPerfectSquare(x) {

const s = Math.sqrt(x);

return s \* s === x;

}

function isFibonacci(num) {

return isPerfectSquare(5 \* num \* num + 4) || isPerfectSquare(5 \* num \* num - 4);

}

// Example usage:

const number = 21;

if (isFibonacci(number)) {

console.log(`${number} is a Fibonacci number.`);

} else {

console.log(`${number} is not a Fibonacci number.`);

}

* **Remove Duplicates**

const numbers3 = [1, 2, 2, 3, 4, 4, 5];

const uniqueNumbers = numbers3.filter((num, index, arr) => arr.indexOf(num) === index);

console.log("array is: [1, 2, 2, 3, 4, 4, 5] remove duplicate values:", uniqueNumbers); // [1, 2, 3, 4, 5]

**OR**

const uniqueElem = [...new Set(numbers3)]

console.log("array is: [1, 2, 2, 3, 4, 4, 5] remove duplicate values:", uniqueElem); // [1, 2, 3, 4, 5]

* **Odd or Even**

const isEven = num => [num].every(n => n % 2 === 0); const isOdd = num => [num].every(n => n % 2 !== 0); // Example usage: const number = 4; console.log(`${number} is ${isEven(number) ? 'even' : 'odd'}.`);

* **Length Of Longest Substring**

function lengthOfLongestSubstring(s) {

const map = new Map();

let start = 0;

return s.split('').reduce((maxLength, char, end) => {

if (map.has(char)) {

start = Math.max(map.get(char) + 1, start);

}

map.set(char, end);

return Math.max(maxLength, end - start + 1);

}, 0);

}

**OR**

function lengthOfLongestSubstring(s) {

const map = new Map();

let start = 0;

let maxLength = 0;

for (let end = 0; end < s.length; end++) {

if (map.has(s[end])) {

start = Math.max(map.get(s[end]) + 1, start);

}

map.set(s[end], end);

maxLength = Math.max(maxLength, end - start + 1);

}

return maxLength;

}

// Example usage:

const str = "abcabcbb";

console.log(lengthOfLongestSubstring(str)); // Output: 3 ("abc")

* **Merge Two Sorted Arrays**

function mergeSortedArrays(arr1, arr2) {

return [...arr1, ...arr2].sort((a, b) => a - b);

}

// Example usage:

const arr1 = [1, 3, 5];

const arr2 = [2, 4, 6];

console.log(mergeSortedArrays(arr1, arr2)); // Output: [1, 2, 3, 4, 5, 6]

* **Find the Intersection of Two Arrays**

function intersect(nums1, nums2) {

const set1 = new Set(nums1);

return [...new Set(nums2.filter(num => set1.has(num)))];

}

// Example usage:

const nums1 = [1, 2, 2, 1];

const nums2 = [2, 2];

console.log(intersect(nums1, nums2)); // Output: [2]

* **Find Duplicate Elements in an Array**

function findDuplicates(arr) {

const counts = arr.reduce((countMap, num) => {

countMap[num] = (countMap[num] || 0) + 1;

return countMap;

}, {});

return Object.keys(counts).filter(num => counts[num] > 1).map(Number);

}

// Example usage:

const arr = [1, 2, 3, 2, 4, 5, 1];

console.log(findDuplicates(arr)); // Output: [1, 2]

* **Check if a String is a Palindrome**

function isPalindrome(s) {

const sanitized = s.toLowerCase().replace(/[^a-z0-9]/gi, '');

return sanitized === [...sanitized].reverse().join('');

}

// Example usage:

const str = "A man, a plan, a canal, Panama";

console.log(isPalindrome(str)); // Output: true

* **Rotate an Array**

function rotateArray(nums, k) {

k = k % nums.length;

return [...nums.slice(-k), ...nums.slice(0, -k)];

}

// Example usage:

const nums = [1, 2, 3, 4, 5, 6, 7];

const k = 3;

console.log(rotateArray(nums, k)); // Output: [5, 6, 7, 1, 2, 3, 4]

* **Find the Longest Palindromic Substring**

function longestPalindrome(s) {

let start = 0;

let maxLength = 1;

const expandAroundCenter = (left, right) => {

while (left >= 0 && right < s.length && s[left] === s[right]) {

left--;

right++;

}

return right - left - 1;

};

for (let i = 0; i < s.length; i++) {

const len1 = expandAroundCenter(i, i); // Odd length

const len2 = expandAroundCenter(i, i + 1); // Even length

const len = Math.max(len1, len2);

if (len > maxLength) {

maxLength = len;

start = i - Math.floor((len - 1) / 2);

}

}

return s.slice(start, start + maxLength);

}

// Example usage:

const str = "babad";

console.log(longestPalindrome(str)); // Output: "bab" or "aba"