**Table of Contents**

1. **Functions in JavaScript**
2. **Functions with Default Arguments**
3. **Functions with Variable Length Arguments**
4. **Generator Functions**
5. **Function Expressions**
6. **Arrow Functions**
7. **Nested Functions**
8. **Hoisting**
9. **Closures**
10. **Higher-Order Functions**
11. **Array Methods**
    * map()
    * filter()
    * reduce()
    * forEach()
    * some()
    * every()
12. **Regular Expressions (RegEx)**
13. **DOM Manipulation**
14. **Event Listeners and Event Propagation**
15. **Synchronous vs Asynchronous JavaScript**
16. **Promises in JavaScript**
17. **API Calls using Axios**
18. **Optional Chaining & Nullish Coalescing Operator**

**1. Functions**

**Description:**  
Functions are reusable blocks of code used to perform a task. They help divide complex problems into smaller chunks, improve readability, and promote code reuse.

**Usage:**  
To encapsulate logic, automate repeated actions, and keep code organized.

**Advantages:**

* Promotes DRY (Don't Repeat Yourself) principle
* Makes code modular and maintainable
* Can return values and accept parameters

**Disadvantages:**

* Nested functions can be hard to manage
* Overuse may clutter logic

**Syntax:**

function functionName(parameters) {

// logic

return value;

}

**Types of Functions:**

1. **Named Functions**  
   Functions declared with a name using the function keyword. These can be hoisted and reused anywhere in the scope.

function greet() {

console.log("Hello");

}

1. **Anonymous Functions**  
   Functions without a name, usually assigned to a variable or used as an argument.

const greet = function() {

console.log("Hi");

};

1. **Arrow Functions**  
   A shorter way to write functions using the => syntax. They inherit this from the surrounding context.

const add = (a, b) => a + b;

1. **Function Expressions**  
   Functions defined as part of an expression, stored in a variable.

const multiply = function(a, b) {

return a \* b;

};

1. **IIFE (Immediately Invoked Function Expression)**  
   A function that runs immediately after it is defined.

(function() {

console.log("IIFE runs immediately");

})();

1. **Generator Functions**  
   Special functions defined with an asterisk \* that can pause and resume execution using yield.

function\* generator() {

yield 1;

yield 2;

}

**2. Functions with Default Arguments**

**Description:**  
Functions that allow default values for parameters if no value or undefined is passed.

**Usage:**  
To simplify function calls when certain values are optional.

**Advantages:**

* Avoids need for manual checks for undefined
* Keeps code clean and minimal

**Disadvantages:**

* Less readable for some developers unfamiliar with the syntax

**Syntax:**

function greet(name = "Guest") {

console.log("Hello " + name);

}

**3. Functions with Variable Length Arguments**

**Description:**  
These use the rest operator ... to accept an indefinite number of arguments as an array.

**Usage:**  
When you don’t know how many arguments will be passed.

**Advantages:**

* Flexible inputs
* Can handle dynamic data

**Disadvantages:**

* Arguments must be managed manually

**Syntax:**

function sum(...args) {

return args.reduce((a, b) => a + b);

}

**4. Generator Functions**

**Description:**  
Special functions that can pause execution using yield and resume later. Defined using function\*.

**Usage:**  
Used for lazy evaluation, custom iterators, and async control flows.

**Advantages:**

* Efficient memory usage
* Better control over iteration

**Disadvantages:**

* Complex to understand and use

**Syntax:**

function\* genFunc() {

yield 1;

yield 2;

}

**5. Function Expression**

**Description:**  
A function defined inside a variable. May or may not have a name (anonymous).

**Usage:**  
Useful in callbacks or function parameters.

**Advantages:**

* Easy to pass as arguments
* Can be anonymous

**Disadvantages:**

* Not hoisted

**Syntax:**

const greet = function(name) {

return "Hello " + name;

};

**6. Arrow Functions**

**Description:**  
Shorter syntax for writing functions using =>. Inherits this from its context.

**Usage:**  
Commonly used in callbacks and short utility functions.

**Advantages:**

* Concise syntax
* Lexical this

**Disadvantages:**

* Cannot be used as constructors
* No arguments object

**Syntax:**

const add = (a, b) => a + b;

**7. Nested Functions**

**Description:**  
Functions declared inside another function.

**Usage:**  
Used to keep helper logic scoped within a parent function.

**Advantages:**

* Localized scope
* Encapsulation

**Disadvantages:**

* Can make code harder to read

**Syntax:**

function outer() {

function inner() {

console.log("Hello from inner");

}

inner();

}

**8. Hoisting**

**Description:**  
JavaScript moves variable and function declarations to the top of their scope before execution.

**Usage:**  
To understand why variables/functions behave unexpectedly.

**Advantages:**

* Allows use of functions before they appear in code

**Disadvantages:**

* Can cause bugs if not understood

**Syntax:**

console.log(x); // undefined

var x = 10;

**9. Closures**

**Description:**  
A closure is formed when an inner function accesses variables from an outer function even after the outer function has finished execution.

**Usage:**  
To create private variables or remember state.

**Advantages:**

* Maintains state between calls
* Useful for data hiding

**Disadvantages:**

* Memory issues if overused

**Syntax:**

function outer() {

let count = 0;

return function inner() {

count++;

console.log(count);

};

}

**10. Higher Order Functions**

**Description:**  
Functions that take other functions as arguments or return functions.

**Usage:**  
Used in functional programming patterns.

**Advantages:**

* Encourages code reuse
* Useful for abstraction

**Disadvantages:**

* May be complex to understand initially

**Syntax:**

function operate(callback, a, b) {

return callback(a, b);

}

**11. Array Methods**

**map()**

**Description**

map() returns a new array by applying a function to each element of the original array. The original array remains unchanged.

**Usage**: Used to transform or format data without modifying the original array.  
**Advantages**: Keeps original array intact, clean syntax, supports chaining.  
**Disadvantages**: Uses extra memory, slightly slower for large data, not meant for side effects.

**Syntax**

array.map((element, index, array) => {

return transformedValue;

});

**Example:**

const nums = [1, 2, 3];

const squares = nums.map(n => n \* n); // [1, 4, 9]

**filter()**

**Description**

The filter() method is used to create a new array that includes only the elements from the original array that pass a specific condition provided by a callback function. It does not modify the original array.

**Usage:** Used to create a new array with elements that satisfy a specific condition.

**Advantages:** Keeps the original array unchanged and improves readability for conditional filtering.

**Disadvantages:** Creates a new array and may be slightly slower for large datasets.

**Syntax**

array.filter((element, index, array) => {

return condition; // must return true or false

});

**Example**

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

const evens = numbers.filter(num => num % 2 === 0);

console.log(evens); // [2, 4]

**reduce()**

**Description**: Reduces the array to a single value by executing a reducer function on each element.  
**Usage**: Used for operations like summing, averaging, or transforming arrays into other formats.  
**Advantages**: Powerful and flexible for complex data transformations.  
**Disadvantages**: Can be confusing to read and debug for beginners.  
**Syntax**:

array.reduce((accumulator, currentValue) => {...}, initialValue)

**Example**:

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

const sum = nums.reduce((acc, curr) => acc + curr, 0); // 10

**forEach()**

**Description**: Executes a provided function once for each array element.  
**Usage**: Used to perform side effects like logging or updating UI for each item.  
**Advantages**: Simple to use and easy to understand.  
**Disadvantages**: Cannot break early or return a new array.  
**Syntax**:

array.forEach((element, index, array) => {...})

**Example**:

const fruits = ['apple', 'banana'];

fruits.forEach(fruit => console.log(fruit));

**some()**

**Description**: Checks if **at least one** element passes the test implemented by the function.  
**Usage**: Used to validate conditions like "is any value true?".  
**Advantages**: Stops early when a condition is met, saving performance.  
**Disadvantages**: Doesn’t return which element passed the test.  
**Syntax**:

array.some((element, index, array) => {...})

**Example**:

const nums = [1, 2, 3];

const hasEven = nums.some(num => num % 2 === 0); // true

**every()**

**Description**: Checks if **all** elements pass the test implemented by the function.  
**Usage**: Used to ensure all items meet a certain condition.  
**Advantages**: Exits early on failure, improving efficiency.  
**Disadvantages**: Fails the entire check if even one element fails.  
**Syntax**:

array.every((element, index, array) => {...})

**Example**:

const nums = [2, 4, 6];

const allEven = nums.every(num => num % 2 === 0); // true

**12. RegEx (Regular Expressions)**

**Description:**  
RegEx is a sequence of characters that defines a search pattern used for string matching and manipulation.

**Usage:**  
Validating input, searching within strings, replacing text patterns.

**Advantages:**

* Powerful string pattern matching
* Short and efficient code

**Disadvantages:**

* Can be complex and hard to read
* Difficult to debug

**Syntax:**

const pattern = /abc/;

pattern.test("abcdef"); // true

**13. DOM (Document Object Model) Manipulation**

**Description:**  
DOM is a programming interface that allows JavaScript to interact with and manipulate HTML and CSS content on a web page.

**Usage:**  
To dynamically change content, styles, structure, or attributes of web pages.

**Advantages:**

* Enables dynamic websites
* Makes web pages interactive

**Disadvantages:**

* Overuse can lead to messy code
* Can affect performance if not managed well

**Syntax:**

document.getElementById("demo").innerText = "Hello World";

**14. Event Propagation**

**Description:**  
Event propagation defines how events travel through the DOM tree. There are two phases: capturing and bubbling.

**Usage:**  
To manage how events are handled in parent-child element relationships.

**Advantages:**

* Offers control over event handling hierarchy

**Disadvantages:**

* Can cause confusion in event management

**Syntax:**

element.addEventListener("click", handler, true); // Capturing phase

**15. Event Listeners**

**Description:**  
Event listeners are functions that wait and respond to specific events like clicks, form submissions, or key presses.

**Usage:**  
To create interactive websites that respond to user actions.

**Advantages:**

* Non-intrusive way to handle events
* Allows multiple listeners

**Disadvantages:**

* Must be managed to avoid memory leaks

**Syntax:**

document.addEventListener("click", function() {

console.log("Clicked");

});

**16. Synchronous vs Asynchronous JavaScript**

**Description:**  
Synchronous JS runs code line by line, while asynchronous JS can handle tasks (like data fetches) without blocking other operations.

**Usage:**  
Used to write efficient, non-blocking code especially for API calls or timers.

**Advantages:**

* Async avoids freezing the UI
* Improves performance and UX

**Disadvantages:**

* Complex to manage async flow

**Syntax:**

setTimeout(() => console.log("Async"), 1000);

**17. Promises**

**Description:**  
Promises represent a value that may be available now, in the future, or never. Used for asynchronous operations.

**Usage:**  
To handle async tasks like API calls, file reading, etc.

**Advantages:**

* Improves code readability over callbacks
* Handles async errors gracefully

**Disadvantages:**

* Can still lead to nested chains (solved by async/await)

**Syntax:**

fetch("url")

.then(response => response.json())

.then(data => console.log(data));

**18. API Calling using Axios**

Axios is a promise-based HTTP client used to make requests to APIs.

**Usage:**  
To retrieve or send data to external servers.

**Advantages:**

* Easy to use
* Supports older browsers
* Automatically transforms JSON

**Disadvantages:**

* Adds external dependency

**Syntax:**

axios.get("https://api.example.com/data")

.then(res => console.log(res.data));

**19. Optional Chaining**

**Description:**  
Optional chaining (?.) lets you safely access deeply nested properties without having to explicitly check for null or undefined.

**Usage:**  
To prevent runtime errors when accessing nested object properties.

**Advantages:**

* Cleaner and safer code

**Disadvantages:**

* May mask bugs if overused

**Syntax:**

const user = {};

console.log(user?.address?.street);

**20. Nullish Coalescing Operator**

**Description:**  
Returns the right-hand operand when the left-hand is null or undefined.

**Usage:**  
To set default values while ignoring 0 or empty string.

**Advantages:**

* More precise than || for null checks

**Disadvantages:**

* May confuse beginners

**Syntax:**

const name = null;

const displayName = name ?? "Guest";