Table of Content

1. Functions
2. Functions with default arguments
3. Functions with variable length arguments
4. Generator function
5. Function expression
6. Arrow functions
7. Nested Functions
8. Hoisting
9. Clousers
10. Higher order functions
11. map
12. filter
13. reduce
14. forEach
15. some
16. every
17. RegEx
18. DOM (manipulations)
19. Event Propagation
20. EventListener
21. Exception Handling
22. Synchronous Asynchronous JavaScript (Promises)
23. API Calling (using Axios)
24. Option chaining and nullish operator

# Functions

1. **Desc:**

Block of code, written to do a specific task.

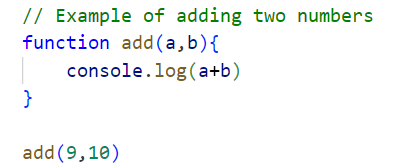
1. **Usage:**

Used to prevent redundantly writing the same code.

1. **Advantages:**

Can be called any number of times.

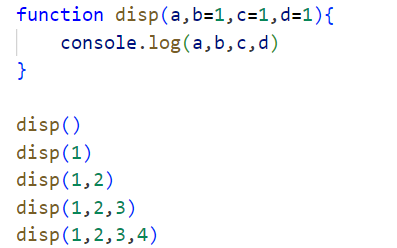
1. **Declaration of function and calling a function:**

****

# Functions with default arguments

**While calling a function when, there are no arguments passed the default arguments will be considered at the time of that function call.**

**Example:**

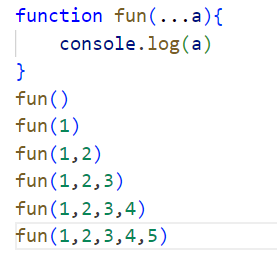


# Functions with variable length arguments

When we are not sure how many elements are going to passes as arguments in the function call, “**variable length arguments**” are used.

We must use the “**rest operator**”: “...a” to accept and store the data elements

**Example:**



# Generator function

When we must get multiple return values form the function we can use the generator function.

It will return the generator object.

**Usage**: Used for lazy iteration, handling asynchronous flows, or producing data on demand

**Advantages:**

* Great for managing large datasets or infinite sequences.
* Easy to pause and resume logic.

**Disadvantages:**

* Slightly harder to understand at first.

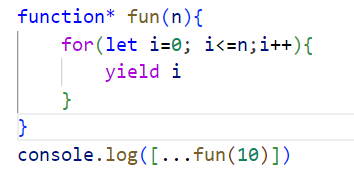
**Syntax:**

function\* genFunc() {

yield 1;

yield 2; }

**Example:**

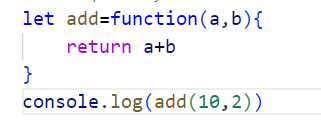


# Function Expression

1. **Desc:**

You can assign function to a variable.

1. **Advantages:**
   1. The ability to be anonymous, assigned to variables.
   2. Used as closures or callbacks, making them versatile for dynamic programming and event handling.
   3. Can be anonymous.
   4. Scoped to block.
2. **Disadvantage:**
   1. Not hoisted like function declarations
3. **Example:**

****

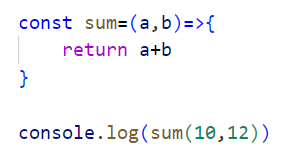
# Arrow functions

* Arrow functions have shorter syntax than regular function expressions.
* Arrow functions have implicit return statements.
* Arrow functions increase readability.

1. **Usage**: Best for callbacks, concise logic.
2. **Advantages**:
   1. No binding of `this`.
   2. Cleaner, shorter syntax.
3. **Disadvantages**:
   1. Not suitable for `this`
   2. sensitive code (like constructors).
4. **Syntax**:

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

**Example:**



**Note**:

An arrow function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

**Differences & Limitations:**

* Does not have its own bindings to this or super, and should not be used as methods.
* Does not have arguments, or new. target keywords.
* Not suitable for call, apply and bind methods, which generally rely on establishing a scope.
* Can not be used as constructors.
* Can not use yield, within its body.



# Nested Functions

**Description**: A function defined inside another function.

**Usage**: Encapsulation, closures, private helper methods.

**Advantages**: Access to outer function's scope.

**Disadvantages**: Can lead to complex and harder-to-read code.

**Syntax**:

function outer() {

function inner() {

console.log('Inner');

} inner();

}

# Hoisting

**Description**: JavaScript's default behavior of moving declarations to the top.

**Usage**: Allows functions/variables to be used before they're declared.

**Advantages**:- Enables flexibility in code organization.

**Disadvantages**:- Can cause confusion or bugs if not understood.

**Syntax**: hoistedFunc(); // Works function hoistedFunc() {

console.log('Hoisted!');

}

# Closures

**Description**: A function that retains access to its outer scope even after the outer function has finished executing.

**Usage**: Creating private variables.

**Advantages**:- Powerful for data hiding and encapsulation.

**Syntax**:

function outer() {

let count = 0; return function() {

count++; return count;

};

}

const counter = outer();

# Higher Order Functions

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

**Usage**: Used in functional programming to create reusable utilities.

**Advantages**:- Promotes code reusability and modular design.

**Syntax**:

function higherOrder(fn) {

return function(val) {

return fn(val);

};

}

# map()

**Description**: Creates a new array by applying a function to each element.

**Usage**: Data transformation.

**Advantages**:- Clean and readable.

**Syntax**:

[1, 2, 3].map(x => x \* 2); // [2, 4, 6]

**Note:**

The main difference between a for loop and the map() method is that a for loop allows you to have more control over the iteration process, while the map() method provides a more concise and functional way to transform elements of an array into a new array.

# filter()

**Description**: Returns a new array with elements that pass a test.

**Usage**: Filtering data based on condition.

**Advantages**:-  The filter method will filter out the elements of an array based on the specified test condition. It will iterate through all the elements and imply the test condition on them. It will then return a new array of elements that will pass the condition.

**Syntax**:

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

# reduce()

**Description**: Reduces array to a single value using a reducer function.

**Usage**: Summing, grouping, complex aggregation.

**Advantages**:- Very flexible.

**Disadvantages**:- Can be hard to read for newcomers.

**Syntax**:

[1, 2, 3].reduce((acc, val) => acc + val, 0); // 6

# forEach()

**Description**: Executes a function for each array element.

**Usage**: Performing side effects (like logging).

**Advantages**:- Simple iteration.

**Disadvantages**:- Doesn't return anything.

**Syntax**:

[1, 2, 3].forEach(x => console.log(x));

# some()

**Description**: Checks if any array element passes the test.

**Usage**: Quick checks.

**Advantages**:- Stops at the first match.

**Syntax**:

[1, 2, 3].some(x => x > 2); // true

# every()

**Description**: Checks if \*all\* elements pass the test.

**Usage**: Validating all values.

**Advantages**: - Clean way to validate arrays.

**Syntax**:

[1, 2, 3].every(x => x > 0); // true

# RegEx

**Description**: Short for Regular Expressions - a way to match patterns in strings.

**Usage**: Validations, search, replace operations.

**Advantages**:- Powerful for string manipulation.

**Disadvantages**:- Can become unreadable for complex patterns

# DOM (Manipulations)

**Description**:

DOM (Document Object Model) is the tree-like structure of an HTML page that JavaScript can interact with.

**Usage**:

Used to access, modify, add, or remove HTML elements dynamically.

**Advantages**:

Enables dynamic content.

Makes pages interactive.

**Disadvantages**:

Can get messy with large trees.

Over-manipulation may lead to performance issues.

**Syntax**:

document.getElementById("id").innerText = "Updated text";

document.querySelector(".class").style.color = "blue";

# Event Propagation

**Description**:

It’s how events flow through the DOM: either bubbling (default) or capturing phase.

**Usage**:

Used to handle events at different stages in the DOM tree (parent/child level).

**Advantages**:

Control over event flow.

Helps in delegation.

**Disadvantages**:

Misuse can cause unintentional event triggers.

**Syntax**:

element.addEventListener('click', callback, true); // true = capture

# EventListener

**Description**:

A method to attach an event handler to an element without overwriting existing ones.

**Usage**:

Used to listen and respond to user interactions like clicks, keypresses, etc.

**Advantages**:

Can attach multiple listeners.

Can remove listeners later.

**Syntax**:

element.addEventListener('click', () => {

console.log('Clicked!');

});

# Exception Handling

**Description**:

Used to catch and handle runtime errors gracefully.

**Usage**:

Wrap risky code inside try...catch blocks to avoid app crashes.

**Advantages**:

Prevents app breakdowns.

Easier debugging/logging.

**Disadvantages**:

Overuse may hide actual bugs.

**Syntax**:

try {

// risky code

} catch (error) {

console.error(error.message);

} finally {

// always runs}

# Synchronous vs Asynchronous JavaScript (Promises)

**Description**:

JS is single-threaded, but can run async code using Promises to handle non-blocking operations.

**Usage**:

Used for tasks like API calls, timeouts, or file reading.

**Advantages**:

Keeps UI responsive.

Clean error handling via .catch().

**Syntax**:

const promise = new Promise((resolve, reject) => {

// async task

});

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

# API Calling (using Axios)

**Description**:

Axios is a promise-based HTTP client for making API requests.

**Usage**:

Used to fetch or send data to external servers.

**Advantages**:

Easy syntax.

Automatic JSON parsing.

Better error handling than fetch.

**Syntax**:

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

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

.catch(err => console.error(err));

# Optional Chaining & Nullish Coalescing Operator

**Description**:

?. allows safe access to deeply nested properties.

?? provides fallback only if value is null or undefined.

**Usage**:

Helps avoid runtime errors when accessing optional data.

**Advantages**:

Cleaner, safer code.

Avoids long condition checks.

**Syntax**:

const user = {};

console.log(user?.profile?.name); // undefined, not error

const value = null;

console.log(value ?? 'default'); // 'default'