Table of Content

Functions

- **Description**: Blocks of reusable code to perform specific tasks.
- **Usage**: Code reusability and organization.
- **Pros**: Modular code, easy debugging.
- **Cons**: Can lead to nested complexity.
- **Example**:

```
 function greet(name) {
  return `Hello'+name;}
greet("John");
```

Functions with Default Arguments

Description: You can assign default values to function parameters if no value or undefined is passed.

Usage: Used to prevent undefined values and make functions more flexible.

Pros

- Prevents errors from missing arguments
- Provides fallback values

Cons

- Can hide bugs if defaults are misused

Code Snippet

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

greet(); // Output: Hello, Guest!
```

Functions with Variable Length Arguments

Description

Using the ...rest parameter, functions can accept any number of arguments.

Usage
Used when the number of inputs is unknown or variable.

Pros

- Very flexible
- Reduces need for multiple overloaded functions

Cons

- Can lead to complexity if overused

Code Snippet

```
function sum(...numbers) {
  return numbers.reduce((acc, val) => acc + val, 0);
}


console.log(sum(1, 2, 3, 4)); // Output: 10
```

Generator Function

- **Description**: Function that can pause and resume using yield.
- **Usage**: Lazy evaluation, iterators.
- **Pros**: Efficient memory use.
- **Cons**: Complex syntax.
- **Example**:

```
function* numbers() {
  yield 1;
  yield 2;
  yield 3;
}
for (let num of numbers()) console.log(num);
```

Function Expression

- **Description**: Function assigned to a variable.
- **Usage**: Useful for callbacks and anonymous functions.
- **Pros**: More flexible than declarations.
- **Cons**: Cannot be hoisted.
- **Example**:

```
const greet = function(name) {
  return `Hello, ${name}`;
};
```

Arrow Functions

- **Description**: Short syntax for writing functions.
- **Usage**: Used in callbacks, one-liners.
- **Pros**: Concise syntax, this is lexically scoped.
- **Cons**: Not suited for methods or constructors.
- **Example**:

```
const greet = name => `Hi, ${name}`;
```

Nested Functions

- **Description**: Functions defined inside another function.
- **Usage**: Encapsulation and closures.
- **Pros**: Better scoping.
- **Cons**: Over-nesting can cause confusion.
- **Example**:

```
function outer() {
  function inner() {
    console.log("Inner function");
  }
  inner();
}
```

Hoisting

- **Description**: JavaScript's behavior of moving declarations to the top.
- **Usage**: Helps in understanding scope.
- **Pros**: Enables use before declaration.
- **Cons**: Can cause unexpected results.
- **Example**:

```
console.log(a);
var a = 5;
```

Closures

- **Description**: Function defined inside of another function, the inner function has access to the variables and scope of the outer function .
- **Usage**: Data privacy ,encapsulation, state maintenance.
- **Pros**: Powerful tool for state.
- **Cons**: May lead to memory leaks.

```
function outer() {
  let count = 0;
  return function() {
    return ++count;
  };
}
const counter = outer();
```

Higher Order Functions

- **Description**: Functions that take or return other functions.
- **Usage**: Functional programming.
- **Pros**: Cleaner and modular code.
- **Cons**: May reduce readability.
- **Example**:

```
function greet(fn) {
    fn();
}
greet(() => console.log("Hello!"));
```

Map

- **Description**: The map() method creates a new array by applying a callback function to each element of the original array. It does not modify the original array.
- **Usage**: Transforming arrays.
- **Syntax**:

```
array.map(function(currentValue, index, array) {
  // return element for new array
});
```

- **Example**:

```
[1, 2, 3].map(n => n * 2);
```

Filter

- **Description**: The filter() method creates a new array with all elements that pass a test (return true) provided by a callback function.
- **Syntax**:

  ```
  array.filter(function(element, index, array) {
    // return true to keep the element
  });
  ```

- Example:

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

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

  console.log(evens); // [2, 4, 6]
  ```

Reduce

- **Description**: The reduce() method executes a **r**educer function on each element of the array, resulting in a single output value.
- **Syntax**:

  ```
  array.reduce(function(accumulator, currentValue, index, array) {
    // return updated accumulator
  }, initialValue);
  ```

-
  ```
  const numbers = [1, 2, 3, 4];

  const sum = numbers.reduce((acc, cur) => acc + cur, 0);

  console.log(sum); // 10
  ```

forEach

- **Description**: Executes a function for each array element.
- **Example**:
  ```
  [1, 2, 3].forEach(n => console.log(n));
  ```

Some

- **Description**: Returns true if at least one element in the array satisfies the condition in the callback.
- **Syntax**:
  ```
  array.some((element, index, array) => {
    return condition;
  });
  ```
- Example
  ```
  const numbers = [1, 2, 3, 4];
  const hasEven = numbers.some(num => num % 2 === 0);
  console.log(hasEven); // true (because 2 and 4 are even)
  ```

Every

- **Description**: Returns true **only if all elements** satisfy the condition.
-
- **Syntax**:
  ```
  array.every((element, index, array) => {
    return condition;
  });
  ```
- **Example:**
  ```
  const numbers = [2, 4, 6];
  const allEven = numbers.every(num => num % 2 === 0);
  console.log(allEven); // true (all numbers are even)
  ```

Regular Expressions (RegEx)

- **Description**: Returns true **only if all elements** satisfy the condition.
- **Syntaxes**:
  const pattern = /pattern/flags;
- **RegEx Pattern**

Pattern	**Matches**
.	Any character (except newline)
\d	Any digit (0-9)
\w	Any word character (a-z, A-Z, 0-9, _)
\s	Whitespace
^	Start of string
$	End of string
[abc]	Any of a, b, or c
[^abc]	Not a, b, or c
`a	b`
*	0 or more times
+	1 or more times
?	0 or 1 time
{n}	Exactly n times

- **Example:**
  const str = "hello world";
  const result = /hello/.test(str);
  console.log(result); // true

DOM

- **Description**: DOM stands for **Document Object Model**.
  It is a programming interface that represents HTML and XML documents as a tree structure, where each node is an object representing a part of the document (like elements, attributes, text).
- JavaScript can use the DOM to access, modify, or remove elements on a webpage dynamically.

- **Example**:
```
<html>
  <body>
    <h1>Hello</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

- 🛠 **Accessing Elements in the DOM:**

Method	**Description**
document.getElementById()	Get element by ID
document.getElementsByClassName()	Get all elements with a class
document.getElementsByTagName()	Get all elements with a tag
document.querySelector()	Get the first match of a selector
document.querySelectorAll()	Get all matches of a selector

- **Modifying DOM Content:**

Property/Method	**What it does**
.innerText	Get/set text content
.innerHTML	Get/set HTML inside element
.setAttribute()	Set an attribute
.style	Change inline styles

- **Creating & Appending Elements:**

```
const newElement = document.createElement("div");
newElement.innerText = "I'm new!";
document.body.appendChild(newElement);
```

- ❌ **Removing Elements**:

```
const el = document.getElementById("demo");
el.remove();
```

- ⚡ **Use Cases**:

  1. Update UI content
  2. Handle user input
  3. Build dynamic web applications

Event Propagation

- **Description**: Event propagation refers to the order in which event handlers are called when an event occurs on nested elements.

There are 3 phases in event propagation:

1. **Capturing Phase (**from root to target)
   The event starts from the **document** and moves **downward** through each parent element until it reaches the target.

2. **Target Phase** (actual target element)
   The event reaches the **actual element** that was clicked or interacted with.

3. **Bubbling Phase** (from target back to root)
   The event then bubbles **upward**, moving from the target element's parent all the way up to the document.

Event Listener

- **Description**: An event listener JavaScript is a function that waits for a specific event to happen on an element — like a button click, mouse movement, key press, etc. When that event occurs, the event listener triggers a callback function you provide.
- **Syntax**:
  element.addEventListener(event, callback);
- **Example:**

```
<button id="myButton">Click Me</button>

<script>
  const button = document.getElementById('myButton');

  button.addEventListener('click', function () {
    alert('Button was clicked!');
  });
</script>
```

Exception Handling

- **Description**: Exception handling in JavaScript is done using the try...catch statement, and optionally finally. This allows you to gracefully handle errors that occur during the execution of code.
- **Syntax:**

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // (Optional) Code that will always run, no matter what
}
```

- **Example:**

```
try {
  let result = someUndefinedFunction(); // This will throw an error
  console.log(result);
} catch (err) {
  console.error("An error occurred:", err.message);
} finally {
  console.log("This will always run");
}
```

- **Real world Example:** User login, form validation, payment processing

Synchronous Asynchronous Javascript

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.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'