1. Introduction to JavaScript

What is JavaScript?

JavaScript is a versatile programming language primarily used to make web pages interactive. It can manipulate content, respond to user actions, and communicate with servers.

History and Evolution

- **1995**: JavaScript was created by Brendan Eich at Netscape.
- **1997**: Standardized as ECMAScript (ES1).
- **2009**: Major update (ES5) introduced new features.
- **2015**: ES6/ES2015 brought modern features like `let`, `const`, and arrow functions.
- **Today**: Widely used for frontend (browsers) and backend (Node.js) development.

JavaScript in Browsers and Node.js

- **Browsers**: JavaScript runs directly in the browser to enhance user interaction.
- **Node.js**: A runtime that allows JavaScript to run on servers.

Setting up the Development Environment

1. **Browser Console**: Open the developer tools (`F12` or `Ctrl+Shift+I`) and use the **Console** tab to run JavaScript.

2. **VS Code**: Install VS Code and set up a file with a `.js` extension to write and test JavaScript code.

javascript
Copy code

```
// Example: Browser Console
console.log("Hello from the console!");
```

---

2. Basics of JavaScript

a. Console

The `console` object provides methods to log messages, errors, and other useful information.

javascript
Copy code

```
// Logging messages
console.log("This is a regular log.");
console.error("This is an error message.");
console.warn("This is a warning message.");

// Other methods
const data = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
];
console.table(data); // Displays data in a table format

console.time("Timer"); // Starts a timer
// Simulating a task
for (let i = 0; i < 1000000; i++) {}
console.timeEnd("Timer"); // Ends the timer

console.group("Grouped Logs");
console.log("Log 1");
```

```
console.log("Log 2");
console.groupEnd();
```

---

b. Data Types

1. **Primitive Types**: Represent single values.
    - **String**: Text data (`"Hello"`)
    - **Number**: Numeric values (`42`, `3.14`)
    - **Boolean**: Logical values (`true`, `false`)
    - **Undefined**: Variable declared but not assigned
    - **Null**: Represents no value
    - **Symbol**: Unique identifier
    - **BigInt**: Large integers beyond `Number` limits
2. **Reference Types**: Represent collections or objects.
    - **Objects**: `{ key: value }`
    - **Arrays**: `[1, 2, 3]`
    - **Functions**: Callable blocks of code

javascript
Copy code

```
let age = 25; // Number
let name = "John"; // String
let isStudent = false; // Boolean
let scores = [90, 80, 85]; // Array
let person = { name: "John", age: 25 }; // Object
let greet = () => "Hello"; // Function

// Type checking
console.log(typeof age); // number
console.log(scores instanceof Array); // true
```

---

c. Variables

- **`var`**: Function-scoped, hoisted, allows re-declaration.

- **`let`**: Block-scoped, hoisted, no re-declaration.
- **`const`**: Block-scoped, hoisted, immutable reference.

javascript
Copy code
```
var x = 10; // Function-scoped
let y = 20; // Block-scoped
const z = 30; // Immutable reference

// Hoisting Example
console.log(a); // undefined (var hoisted)
var a = 5;

// Scope Example
{
  let blockScoped = "Hello";
  console.log(blockScoped); // Accessible here
}
// console.log(blockScoped); // Error: not defined
```

---

3. Operators

Arithmetic Operators

javascript
Copy code
```
let x = 10, y = 3;
console.log(x + y); // 13
console.log(x - y); // 7
console.log(x * y); // 30
console.log(x / y); // 3.333...
console.log(x % y); // 1 (remainder)
```

Comparison Operators

javascript
Copy code
```
console.log(5 == "5"); // true (loose equality)
console.log(5 === "5"); // false (strict equality)
console.log(5 > 3); // true
console.log(5 <= 5); // true
```

Logical Operators

javascript
Copy code
```
console.log(true && false); // false
console.log(true || false); // true
console.log(!true); // false
```

Ternary Operator

javascript
Copy code
```
const age = 18;
const canVote = age >= 18 ? "Yes" : "No";
console.log(canVote); // Yes
```

---

4. Control Structures

Conditional Statements

javascript
Copy code
```
let score = 85;

if (score > 90) {
  console.log("Grade: A");
} else if (score > 75) {
  console.log("Grade: B");
```

```
} else {
  console.log("Grade: C");
}

let color = "red";

switch (color) {
  case "red":
    console.log("Stop");
    break;
  case "green":
    console.log("Go");
    break;
  default:
    console.log("Caution");
}
```

Loops

- **For Loop**

javascript
Copy code

```
for (let i = 1; i <= 5; i++) {
  console.log(`Iteration ${i}`);
}
```

- **While Loop**

javascript
Copy code

```
let count = 1;
while (count <= 5) {
  console.log(`Count: ${count}`);
  count++;
```

```
}
```

- **Do-While Loop**

javascript
Copy code
```
let num = 5;
do {
  console.log(`Number: ${num}`);
  num--;
} while (num > 0);
```

Loop Control

javascript
Copy code
```
for (let i = 1; i <= 5; i++) {
  if (i === 3) continue; // Skip iteration
  if (i === 5) break; // Exit loop
  console.log(i);
}
```

---

5. Functions

Function Declaration and Invocation

Functions are reusable blocks of code. A declared function can be called as needed.

javascript
Copy code
```
// Function Declaration
function greet(name) {
  // 'name' is a parameter
  return `Hello, ${name}!`;
}
```

```
// Function Invocation
console.log(greet("Alice")); // Outputs: Hello, Alice!
```

Real-World Example: Calculate the area of a rectangle.

javascript
Copy code

```
function calculateArea(length, width) {
  return length * width;
}

console.log(calculateArea(5, 10)); // Outputs: 50
```

Function Expressions

A function expression assigns a function to a variable.

javascript
Copy code

```
const sayHello = function(name) {
  return `Hi, ${name}!`;
};

console.log(sayHello("Bob")); // Outputs: Hi, Bob!
```

Arrow Functions (=>)

A shorthand for writing functions introduced in ES6.

javascript
Copy code

```
// Arrow Function Example
const multiply = (a, b) => a * b;
```

```
console.log(multiply(4, 5)); // Outputs: 20
```

Real-World Example: Format a price.

javascript
Copy code

```
const formatPrice = (price) => `$${price.toFixed(2)}`;
console.log(formatPrice(19.99)); // Outputs: $19.99
```

Default and Rest Parameters

Default parameters provide fallback values if arguments are missing.

javascript
Copy code

```
function greetUser(name = "Guest") {
  return `Welcome, ${name}!`;
}

console.log(greetUser()); // Outputs: Welcome, Guest!
console.log(greetUser("Alice")); // Outputs: Welcome, Alice!
```

Rest parameters allow functions to handle multiple arguments as an array.

javascript
Copy code

```
function sumAll(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sumAll(1, 2, 3, 4)); // Outputs: 10
```

Returning Values

Functions can return a value to the caller using the `return` keyword.

javascript
Copy code
```
function square(num) {
  return num * num;
}

console.log(square(6)); // Outputs: 36
```

Scope and Closures

Scope determines where variables are accessible. Closures allow a function to "remember" variables from its parent scope.

javascript
Copy code
```
function outerFunction(outerVariable) {
  return function innerFunction(innerVariable) {
    return `Outer: ${outerVariable}, Inner: ${innerVariable}`;
  };
}

const myClosure = outerFunction("outside");
console.log(myClosure("inside")); // Outputs: Outer: outside,
Inner: inside
```

---

6. Working with Arrays

Creating Arrays

Arrays are used to store multiple values in a single variable.

javascript

Copy code

```
const fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits); // Outputs: [ 'Apple', 'Banana', 'Cherry' ]
```

Accessing and Modifying Elements

Array elements are accessed by their index (starting from 0).

javascript
Copy code

```
const fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[1]); // Outputs: Banana

// Modifying an element
fruits[1] = "Blueberry";
console.log(fruits); // Outputs: [ 'Apple', 'Blueberry',
'Cherry' ]
```

Array Methods

- **push**: Adds elements to the end.
- **pop**: Removes the last element.
- **shift**: Removes the first element.
- **unshift**: Adds elements to the beginning.
- **splice**: Modifies the array by adding/removing elements.
- **slice**: Returns a portion of the array.

javascript
Copy code

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

// Adding and removing elements
numbers.push(6); // [1, 2, 3, 4, 5, 6]
numbers.pop();   // [1, 2, 3, 4, 5]
```

```
// Extracting part of an array
const sliced = numbers.slice(1, 3); // [2, 3]
```

Iteration

Iterate through arrays using methods like `forEach`, `map`, `filter`, `reduce`.

javascript
Copy code

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

// forEach example
numbers.forEach(num => console.log(num));

// map example
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]

// filter example
const evens = numbers.filter(num => num % 2 === 0);
console.log(evens); // [2, 4]
```

---

7. Strings

String Methods

- **concat**: Joins two strings.
- **includes**: Checks if a substring exists.
- **slice**: Extracts part of a string.
- **substring**: Similar to `slice`.
- **toUpperCase/toLowerCase**: Changes case.

javascript
Copy code

```
const message = "Hello, World!";
```

```
console.log(message.includes("World")); // true
console.log(message.toUpperCase()); // HELLO, WORLD!
```

---

8. Objects

Creating and Accessing Objects

javascript
Copy code

```
const person = {
  name: "Alice",
  age: 25,
  greet() {
    return `Hi, I'm ${this.name}!`;
  }
};

console.log(person.name); // Alice
console.log(person.greet()); // Hi, I'm Alice!
```

Adding, Updating, and Deleting Properties

javascript
Copy code

```
person.job = "Developer"; // Adding
person.age = 26; // Updating
delete person.job; // Deleting
```

Object Destructuring

javascript
Copy code

```
const { name, age } = person;
console.log(name, age); // Alice 26
```

Spread and Rest Operators

javascript

Copy code

```
// Spread
const personWithCity = { ...person, city: "New York" };

// Rest
const { city, ...rest } = personWithCity;
console.log(rest); // { name: 'Alice', age: 26 }
```

Object Methods and `this`

javascript

Copy code

```
const car = {
  brand: "Tesla",
  getDetails() {
    return `This car is a ${this.brand}`;
  }
};

console.log(car.getDetails()); // This car is a Tesla
```

---

9. Asynchronous JavaScript

Callbacks

A callback is a function passed as an argument to another function and executed later.

javascript

Copy code

```
// Simulating a delay using setTimeout
```

```
function fetchData(callback) {
  console.log("Fetching data...");
  setTimeout(() => {
    callback("Data received!");
  }, 2000);
}

// Using the callback
fetchData((message) => {
  console.log(message); // Outputs after 2 seconds: Data received!
});
```

Promises

Promises simplify asynchronous code by representing a value that will be available in the future.

javascript

Copy code

```
// Creating a Promise
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
      const success = true; // Simulating success

      success ? resolve("Data fetched successfully!") :
reject("Error fetching data");

    }, 2000);

  });

};


// Consuming the Promise

fetchData()

  .then((data) => console.log(data)) // Logs: Data fetched
successfully!

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

  .finally(() => console.log("Operation complete."));
```

---

async and await

`async` and `await` simplify working with Promises.

javascript

Copy code

```
const fetchData = () => {

  return new Promise((resolve) => {

    setTimeout(() => resolve("Fetched with async/await"), 2000);
```

```
  });
};

async function displayData() {
  console.log("Fetching...");
  const data = await fetchData();
  console.log(data); // Outputs: Fetched with async/await
}

displayData();
```

Event Loop and Microtasks

JavaScript's **event loop** processes tasks from the **call stack** and **microtasks** (e.g., Promises).

javascript

Copy code

```
console.log("Start");

setTimeout(() => console.log("Timeout"), 0);

Promise.resolve("Promise resolved").then((msg) =>
console.log(msg));
```

```
console.log("End");


// Output order: Start -> End -> Promise resolved -> Timeout
```

---

10. Error Handling

try-catch and finally

Handle runtime errors gracefully.

javascript

Copy code

```
try {
  console.log("Trying...");
  throw new Error("Something went wrong!");
} catch (error) {
  console.error(error.message); // Outputs: Something went wrong!
} finally {
  console.log("Execution complete.");
}
```

Custom Errors with throw

Manually throw errors when needed.

javascript

Copy code

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero is not allowed!");
  }
  return a / b;
}

try {
  console.log(divide(10, 0));
} catch (error) {
  console.error(error.message); // Outputs: Division by zero is not allowed!
}
```

Debugging with Browser DevTools

Use **breakpoints**, **watch variables**, and the **call stack** in DevTools to identify issues in code.

---

11. Object-Oriented Programming (OOP)

Classes and Constructors

Classes are templates for creating objects.

javascript

Copy code

```
class Car {
  constructor(brand, model) {
    this.brand = brand;
    this.model = model;
  }

  displayInfo() {
    return `${this.brand} ${this.model}`;
  }
}

const myCar = new Car("Tesla", "Model S");
console.log(myCar.displayInfo()); // Outputs: Tesla Model S
```

Inheritance

Use `extends` to create child classes and `super` to call the parent constructor.

javascript

Copy code

```
class ElectricCar extends Car {
  constructor(brand, model, range) {
    super(brand, model); // Call parent constructor
    this.range = range;
  }

  displayInfo() {
    return `${super.displayInfo()} with a range of ${this.range} miles.`;
  }
}

const myElectricCar = new ElectricCar("Tesla", "Model 3", 300);
console.log(myElectricCar.displayInfo());
```

Encapsulation: Public and Private Properties

Use # to define private properties.

javascript

Copy code

```
class BankAccount {
```

```
  #balance;

  constructor(initialBalance) {
    this.#balance = initialBalance;
  }

  deposit(amount) {
    this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount(100);
account.deposit(50);
console.log(account.getBalance()); // Outputs: 150
// console.log(account.#balance); // Error: Private field
```

Prototypes and Prototype Chain

javascript

Copy code

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function () {
  return `${this.name} makes a sound.`;
};

const dog = new Animal("Dog");
console.log(dog.speak()); // Outputs: Dog makes a sound.
```

---

12. Advanced JavaScript

ES6+ Features

- **Template Literals**: Interpolate variables into strings.
- **Default Parameters**: Set default values for function arguments.
- **Destructuring**: Extract values from arrays/objects.
- **Spread/Rest**: Expand or condense collections.

javascript

Copy code

```
const user = { name: "Alice", age: 25 };
```

```
const { name, age } = user; // Destructuring

console.log(`${name} is ${age} years old.`);


const numbers = [1, 2, 3];

console.log([...numbers, 4, 5]); // Spread operator
```

Modules

- `export` allows sharing variables/functions.
- `import` brings them into another file.

javascript

Copy code

```
// utils.js

export function greet(name) {

  return `Hello, ${name}!`;

}


// main.js

import { greet } from "./utils.js";

console.log(greet("Alice"));
```

Generators

Generators yield values one at a time.

javascript

Copy code

```
function* counter() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = counter();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

---

13. Working with APIs

Overview

APIs (Application Programming Interfaces) enable communication between applications. In web development, REST APIs are commonly used to interact with external data sources.

---

1. Fetching Data Using `fetch`

The `fetch()` method is used to make HTTP requests.

javascript
Copy code

```
// Fetch data from a REST API
fetch("https://jsonplaceholder.typicode.com/posts")
  .then((response) => {
    if (!response.ok) throw new Error("Network response was not
ok");
    return response.json(); // Parse JSON data
  })
  .then((data) => {
    console.log("Fetched Posts:", data); // Array of posts
  })
  .catch((error) => {
    console.error("Error fetching data:", error.message);
  });
```

---

2. Parsing JSON Data

APIs often return data in JSON format. We parse this JSON data to use it in our application.

javascript
Copy code

```
// Simulating JSON data
const jsonData = `{
  "id": 1,
  "title": "JavaScript APIs",
  "completed": false
}`;

// Parse JSON string to a JavaScript object
const task = JSON.parse(jsonData);
console.log(task.title); // Outputs: JavaScript APIs

// Convert JavaScript object back to JSON
```

```
const jsonString = JSON.stringify(task, null, 2);
console.log(jsonString);
/*
{
  "id": 1,
  "title": "JavaScript APIs",
  "completed": false
}
*/
```

---

3. Handling Errors in API Calls

Errors can occur due to network issues or invalid responses. Handling errors ensures a better user experience.

javascript
Copy code

```
async function fetchWithErrorHandling() {
  try {
    const response = await
fetch("https://jsonplaceholder.typicode.com/posts/1");
    if (!response.ok) throw new Error(`HTTP error! Status:
${response.status}`);
    const data = await response.json();
    console.log("Post:", data);
  } catch (error) {
    console.error("Error:", error.message);
  }
}

fetchWithErrorHandling();
```

---

4. CRUD Operations with REST APIs

CRUD (Create, Read, Update, Delete) operations demonstrate complete interaction with an API.

Create (POST)

Add a new resource to the server.

javascript
Copy code

```
async function createPost() {
  const newPost = {
    title: "New API Post",
    body: "This is an example of a POST request.",
    userId: 1,
  };

  try {
    const response = await
fetch("https://jsonplaceholder.typicode.com/posts", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(newPost),
    });

    const createdPost = await response.json();
    console.log("Created Post:", createdPost);
  } catch (error) {
    console.error("Error creating post:", error.message);
  }
}

createPost();
```

Read (GET)

Fetch existing resources.

javascript
Copy code

```
async function getPosts() {
  try {
    const response = await
fetch("https://jsonplaceholder.typicode.com/posts");
    const posts = await response.json();
    console.log("Posts:", posts);
  } catch (error) {
    console.error("Error fetching posts:", error.message);
  }
}

getPosts();
```

Update (PUT or PATCH)

Modify an existing resource.

- **PUT**: Updates the entire resource.
- **PATCH**: Updates part of the resource.

javascript
Copy code

```
async function updatePost(postId) {
  const updatedData = { title: "Updated Title", body: "Updated
body content." };

  try {
    const response = await fetch(
      `https://jsonplaceholder.typicode.com/posts/${postId}`,
      {
        method: "PUT", // Use PATCH for partial updates
        headers: { "Content-Type": "application/json" },
```

```
        body: JSON.stringify(updatedData),
      }
    );

    const updatedPost = await response.json();
    console.log("Updated Post:", updatedPost);
  } catch (error) {
    console.error("Error updating post:", error.message);
  }
}

updatePost(1);
```

Delete (DELETE)

Remove an existing resource.

javascript
Copy code

```
async function deletePost(postId) {
  try {
    const response = await fetch(
      `https://jsonplaceholder.typicode.com/posts/${postId}`,
      { method: "DELETE" }
    );

    if (response.ok) {
      console.log(`Post with ID ${postId} deleted
successfully.`);
    } else {
      throw new Error("Failed to delete the post.");
    }
  } catch (error) {
    console.error("Error deleting post:", error.message);
  }
```

```
}

deletePost(1);
```

---

5. Real-World Example: Display Posts in the Browser

This example fetches posts and displays them in a list.

html
Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>API Example</title>
</head>
<body>
  <h1>Posts</h1>
  <ul id="postList"></ul>

  <script>
    async function fetchAndDisplayPosts() {
      try {
        const response = await
fetch("https://jsonplaceholder.typicode.com/posts");
        const posts = await response.json();

        const postList = document.getElementById("postList");
        posts.forEach((post) => {
          const li = document.createElement("li");
          li.textContent = `${post.id}: ${post.title}`;
          postList.appendChild(li);
        });
```

```
      } catch (error) {
        console.error("Error fetching posts:", error.message);
      }
    }

    fetchAndDisplayPosts();
  </script>
</body>
</html>
```

---

14. Browser Storage

Browser storage is used to store data locally in a user's browser. It includes **Local Storage**, **Session Storage**, and **Cookies**.

Local Storage

- Stores data with no expiration time.
- Data persists even after the browser is closed.

javascript
Copy code

```
// Set an item in Local Storage
localStorage.setItem("username", "JohnDoe");

// Get an item from Local Storage
const username = localStorage.getItem("username");
console.log(username); // Outputs: JohnDoe

// Remove an item from Local Storage
localStorage.removeItem("username");

// Clear all items from Local Storage
localStorage.clear();
```

Example Use Case: Save user preferences (e.g., theme).

Session Storage

- Stores data for the duration of the page session.
- Data is cleared once the tab/browser is closed.

javascript
Copy code

```
// Set an item in Session Storage
sessionStorage.setItem("authToken", "abcdef12345");

// Get an item from Session Storage
const token = sessionStorage.getItem("authToken");
console.log(token); // Outputs: abcdef12345

// Remove an item from Session Storage
sessionStorage.removeItem("authToken");

// Clear all items from Session Storage
sessionStorage.clear();
```

Example Use Case: Temporary data, such as form inputs during a session.

Cookies

- Small pieces of data stored with an expiration date.
- Sent with every HTTP request to the server.

javascript
Copy code

```
// Set a cookie
```

```
document.cookie = "user=JaneDoe; expires=Fri, 31 Dec 2025
23:59:59 GMT; path=/";

// Get cookies
console.log(document.cookie); // Outputs: user=JaneDoe

// Delete a cookie (set expiration date to past)
document.cookie = "user=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
path=/";
```

Example Use Case: Track user sessions or preferences across the server.

---

15. Regular Expressions

Regular Expressions (RegEx) are patterns used for text matching and manipulation.

Basics: Patterns and Flags

- **Patterns**: Define the string structure you want to match.
- **Flags**:
  - `g`: Global match.
  - `i`: Case-insensitive match.
  - `m`: Multi-line mode.

javascript
Copy code

```
const regex = /hello/i; // Match "hello" (case-insensitive)
const testStr = "Hello World!";
console.log(regex.test(testStr)); // Outputs: true
```

Common Methods

1. **test()**: Tests for a match.

2. **match()**: Returns matched substrings.
3. **replace()**: Replaces matched substrings.
4. **search()**: Finds the index of the first match.
5. **split()**: Splits a string based on matches.

Examples

javascript

Copy code

```
// Test for a pattern
const pattern = /JavaScript/;
console.log(pattern.test("I love JavaScript!")); // true

// Match substrings
const str = "I love JavaScript and JavaScript loves me!";
const matches = str.match(/JavaScript/g);
console.log(matches); // ["JavaScript", "JavaScript"]

// Replace matches
const newStr = str.replace(/JavaScript/g, "JS");
console.log(newStr); // I love JS and JS loves me!

// Search for a pattern
const index = str.search(/JavaScript/);
console.log(index); // 7

// Split based on a pattern
const parts = str.split(/and/);
console.log(parts); // ["I love JavaScript ", " JavaScript loves me!"]
```

---

16. Performance Optimization

Performance optimization improves application speed and efficiency.

Debouncing

Limits how often a function executes by delaying it until after a specified time of inactivity.

javascript
Copy code

```
function debounce(func, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => func.apply(this, args), delay);
  };
}

// Example: Debounced search
const searchInput = document.getElementById("search");
searchInput.addEventListener(
  "input",
  debounce((event) => {
    console.log("Searching for:", event.target.value);
  }, 300)
);
```

Throttling

Ensures a function executes at most once in a specified time interval.

javascript
Copy code

```
function throttle(func, interval) {
  let lastCall = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= interval) {
```

```
      lastCall = now;
      func.apply(this, args);
    }
  };
}

// Example: Throttled resize event
window.addEventListener(
  "resize",
  throttle(() => {
    console.log("Window resized!");
  }, 500)
);
```

Memory Leaks

Common causes of memory leaks:

1. Unreferenced global variables.
2. Forgotten timers (e.g., `setInterval`).
3. Detached DOM nodes.

Garbage Collection

JavaScript automatically removes unused objects to free memory, but avoid patterns that prevent it.

---

17. Advanced Topics

Web Workers

Run scripts in the background to keep the UI responsive.

Main File:

javascript
Copy code
```
const worker = new Worker("worker.js");
worker.postMessage("Start Worker!");

worker.onmessage = (event) => {
  console.log("Message from Worker:", event.data);
};
```

worker.js:

javascript
Copy code
```
onmessage = (event) => {
  console.log("Worker Received:", event.data);
  postMessage("Hello from Worker!");
};
```

Service Workers and Progressive Web Apps (PWAs)

- Service Workers act as a proxy for network requests, enabling offline capabilities.

Register a Service Worker:

javascript
Copy code
```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker
    .register("/service-worker.js")
    .then(() => console.log("Service Worker Registered"))
    .catch((error) => console.error("Service Worker Registration
Failed:", error));
}
```

WebSockets

Enable real-time, bi-directional communication.

javascript
Copy code

```
const socket = new WebSocket("ws://example.com/socket");

socket.onopen = () => {
  console.log("Connected to WebSocket!");
  socket.send("Hello Server!");
};

socket.onmessage = (event) => {
  console.log("Message from Server:", event.data);
};
```

IndexedDB

Client-side database for large structured data.

javascript
Copy code

```
const request = indexedDB.open("MyDatabase", 1);

request.onupgradeneeded = (event) => {
  const db = event.target.result;
  db.createObjectStore("Users", { keyPath: "id" });
};

request.onsuccess = (event) => {
  const db = event.target.result;

  const transaction = db.transaction("Users", "readwrite");
  const store = transaction.objectStore("Users");
  store.add({ id: 1, name: "Alice", age: 25 });
```

```
  transaction.oncomplete = () => {
    console.log("Data added successfully!");
  };
};
```
