JavaScript

Closures and lexical scope: -

Closures and lexical scope are fundamental concepts in JavaScript that help in creating private variables, function factories, and maintaining state across function executions.

1.Lexical Scope: -

Lexical scope means that a function's scope is determined by where it is declared in the code. Inner functions have access to variables from their outer functions.

Example of Lexical Scope: -

```
function outer() {
    let outerVariable = "I'm from outer function";
    function inner() {
        console.log(outerVariable); // Accessing outer function's variable
    }
    inner();
}
outer(); // Output: I'm from outer function
```

2.Closures: -

A closure is created when an inner function retains access to variables from its outer function, even after the outer function has finished execution.

Closures are useful for data privacy, function factories, and maintaining state.

Example of Closure: -

```
function counter() {
    let count = 0; // Private variable
    return function () {
        count++;
        console.log(count);
    };
}
const increment = counter();
increment(); // Output: 1
increment(); // Output: 2
```

```
increment(); // Output: 3
```

Even after counter() has finished execution, the inner function still has access to count.

Practical Use Cases of Closures: -

a) Data Privacy (Emulating Private Variables)

Closures help create private variables that can't be accessed directly from outside.

```
function createBankAccount(initialBalance) {
    let balance = initialBalance;
    return {
        deposit(amount) {
            balance += amount;
            console.log(`Deposited: $${amount}, New Balance: $${balance}`);
        },
        withdraw(amount) {
            if (amount <= balance) {
                balance -= amount;
                console.log(`Withdrew: $${amount}, Remaining Balance: $${balance}`);
            } else {
                console.log("Insufficient funds");
            }
        },
        getBalance() {
            return balance;
        }
    };
}
const myAccount = createBankAccount(1000);
myAccount.deposit(500);   // Deposited: $500, New Balance: $1500
myAccount.withdraw(200);  // Withdrew: $200, Remaining Balance: $1300
console.log(myAccount.balance); // Undefined (can't access directly)
```

Balance remains private and can only be accessed via methods.

b) Function Factories

Closures allow functions to generate customized behavior.

```
function multiplier(factor) {
   return function (number) {
      return number * factor;
   };
}
const double = multiplier(2);
const triple = multiplier(3);
console.log(double(5)); // Output: 10
console.log(triple(5)); // Output: 15
```

The inner function remembers factor, creating reusable functions.

Differences between Lexical Scope and Closures: -

Feature	**Lexical Scope**	**Closures**
Definition	Function scope determined at declaration	Function that retains access to outer scope even after execution
When it Occurs	When a function is declared	When an inner function is returned and used later
Access	Inner functions access outer function's variables	Inner function retains variables even after outer function execution
Example	Accessing variables inside function	Maintaining state after function execution

WebSockets - Real-time Communication: -

WebSockets enable real-time, bidirectional communication between a client and a server over a persistent connection. Unlike HTTP requests, which are unidirectional and require polling, WebSockets allow continuous data exchange with minimal latency.

Persistent connection (reduces overhead from repeated requests).

Low latency, real-time updates.

Full-duplex communication (both client and server can send/receive data simultaneously).

Ideal for chat applications, live notifications, real-time analytics, collaborative tools, etc.

WebSocket Basics: -

a) Creating a WebSocket Connection

JavaScript provides the WebSocket API for client-side WebSocket connections.

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

wss:// is the secure WebSocket protocol (similar to https:// for secure HTTP).

b) Handling WebSocket Events

Event	**Description**
onopen	Triggered when the connection is established
onmessage	Fired when a message is received from the server
onclose	Triggered when the connection is closed
onerror	Fired when an error occurs

Example: - Handling WebSocket Events

```
const socket = new WebSocket("wss://example.com/socket");
// Connection opened
socket.onopen = () => {
    console.log("WebSocket connection established");
    socket.send("Hello Server!");
};
// Receiving messages
socket.onmessage = (event) => {
    console.log("Message from server:", event.data);
};
// Handling errors
socket.onerror = (error) => {
    console.error("WebSocket error:", error);
};
// Connection closed
socket.onclose = () => {
    console.log("WebSocket connection closed");
};
```

This ensures the client can send and receive messages from the server.

WebSockets on the Server (Node.js Example)

For real-time communication, a WebSocket server is required. The ws library in Node.js helps create WebSocket servers.

Setting Up a WebSocket Server: -

```
const WebSocket = require("ws");

const server = new WebSocket.Server({ port: 8080 });

server.on("connection", (socket) => {
    console.log("Client connected");
    socket.on("message", (message) => {
        console.log("Received:", message);
        socket.send("Message received: " + message);
    });
    socket.on("close", () => {
        console.log("Client disconnected");
    });
});
```

This sets up a WebSocket server listening on port 8080.

Broadcasting Messages to All Clients: -

A common use case is broadcasting messages to all connected clients.

```
server.on("connection", (socket) => {
    socket.on("message", (message) => {
        server.clients.forEach(client => {
            if (client.readyState === WebSocket.OPEN) {
                client.send(message);
            }
        });
    });
});
```

This allows messages from one client to be sent to all connected clients (useful for chat apps, real-time updates, etc.).

Real-time Chat Application Example: -

Client-Side WebSocket Implementation

```
const socket = new WebSocket("ws://localhost:8080");
document.getElementById("sendBtn").addEventListener("click", () => {
    const message = document.getElementById("messageInput").value;
    socket.send(message);
});
socket.onmessage = (event) => {
    const chatBox = document.getElementById("chatBox");
    chatBox.innerHTML += `<p>${event.data}</p>`;
};
```

Server-Side WebSocket for Chat

```
const WebSocket = require("ws");
const server = new WebSocket.Server({ port: 8080 });
server.on("connection", (socket) => {
    socket.on("message", (message) => {
        server.clients.forEach(client => {
            if (client.readyState === WebSocket.OPEN) {
                client.send(message);
            }
        });
    });
});
```

This simple chat app allows multiple clients to exchange messages in real-time.

WebSockets vs. HTTP Polling

Feature	**WebSockets**	**HTTP Polling**
Connection Type	Persistent	Repeated requests
Latency	Low	Higher
Server Load	Low	High (due to frequent requests)
Ideal Use Cases	Chat apps, live updates	Periodic data fetching

WebSockets are ideal for real-time applications, whereas HTTP polling is better suited for non-frequent updates.

WebSocket Security Considerations: -

Use wss:// instead of ws:// for encrypted communication.

Implement authentication (e.g., tokens, sessions), encryption.