Chain of Responsibility Design Pattern in JavaScript

Overview:

The Chain of Responsibility design pattern is a behavioral pattern that allows a request to pass through a chain of handlers. Each handler decides whether to handle the request or pass it to the next handler in the chain.

Use Case:

This pattern is useful when multiple objects may handle a request, and the handler is not known in advance. It decouples the sender and receiver of a request.

JavaScript Implementation:

1. Define a base Logger class:
   - It has a level and a reference to the next logger.
   - It has a method logMessage() to decide whether to process or pass the message.
   - A write() method is implemented in subclasses.

2. Create constants for log levels: INFO, DEBUG, ERROR.

3. Create subclasses of Logger:
   - InfoLogger: Handles INFO level messages.
   - DebugLogger: Handles DEBUG level messages.
   - ErrorLogger: Handles ERROR level messages.

4. Chain Setup:

- Link ErrorLogger -> DebugLogger -> InfoLogger.

5. Example Usage:

- Send messages with different levels to the first logger in the chain.

JavaScript Code:

```
class Logger {
    constructor(level) {
        this.level = level;
        this.nextLogger = null;
    }

    setNextLogger(nextLogger) {
        this.nextLogger = nextLogger;
    }

    logMessage(level, message) {
        if (level >= this.level) {
            this.write(message);
        }
        if (this.nextLogger) {
            this.nextLogger.logMessage(level, message);
        }
    }
```

```
    write(message) {
        // To be implemented by subclasses
    }
}

const INFO = 1;
const DEBUG = 2;
const ERROR = 3;

class InfoLogger extends Logger {
    write(message) {
        console.log(`INFO: ${message}`);
    }
}

class DebugLogger extends Logger {
    write(message) {
        console.log(`DEBUG: ${message}`);
    }
}

class ErrorLogger extends Logger {
    write(message) {
        console.log(`ERROR: ${message}`);
```

```
    }
}

function getChainOfLoggers() {
    const errorLogger = new ErrorLogger(ERROR);
    const debugLogger = new DebugLogger(DEBUG);
    const infoLogger = new InfoLogger(INFO);

    errorLogger.setNextLogger(debugLogger);
    debugLogger.setNextLogger(infoLogger);

    return errorLogger;
}

const loggerChain = getChainOfLoggers();
loggerChain.logMessage(INFO, "This is an info.");
loggerChain.logMessage(DEBUG, "This is a debug.");
loggerChain.logMessage(ERROR, "This is an error.");
```

Output:

```
INFO: This is an info.
DEBUG: This is a debug.
INFO: This is a debug.
ERROR: This is an error.
```

DEBUG: This is an error.

INFO: This is an error.

Conclusion:

The Chain of Responsibility pattern helps in processing requests through a sequence of potential handlers. It's commonly used in logging frameworks, middleware, or validation systems in web development.

Real-World Examples:

1. Middleware in Express.js:

In Express.js, middleware functions are chained to handle HTTP requests. Each middleware can process the request, send a response, or pass it to the next middleware.

Example:

```
app.use((req, res, next) => {
    console.log('Logging request');
    next();
});

app.use((req, res, next) => {
    if (!req.headers['auth']) return res.status(403).send('Forbidden');
    next();
});
```

```
app.get('/', (req, res) => {
    res.send('Hello World');
});
```

2. Event Handling in DOM:

Multiple event handlers can be attached to a single DOM event. The event propagates through the chain (capturing -> target -> bubbling).

Example:

```
document.querySelector("#btn").addEventListener("click", () => console.log("Handler 1"));
document.querySelector("#btn").addEventListener("click", () => console.log("Handler 2"));
```

3. Form Validation Pipeline:

A form can have multiple validators (e.g., required, email format, length) that are chained to validate a field.

Example:

```
function required(value) {
    return value ? null : "This field is required";
}

function isEmail(value) {
    return /\S+@\S+\.\S+/.test(value) ? null : "Invalid email format";
}
```

```
function validate(value, validators) {
    for (let validator of validators) {
        const error = validator(value);
        if (error) return error;
    }
    return null;
}

const result = validate("test@", [required, isEmail]);
console.log(result); // Output: "Invalid email format"
```

These examples demonstrate how the Chain of Responsibility pattern is useful in real-world JavaScript development.