Notes - JavaScript - Gang of Four - Behavioral - Chain of Responsibility

Dr Nick Hayward

A brief introduction to the Chain of Responsibility (CoR) pattern with JavaScript/TypeScript examples.

What is the Chain of Responsibility?

CoR passes a request along a chain of handlers. Each handler decides whether to process the request or pass it to the next handler. This decouples senders from receivers and lets you add/reorder handlers without changing the client.

Benefits:

- decoupling: client only talks to the chain head
- flexibility: add/remove/reorder handlers without changing caller code
- efficiency: early handlers can short-circuit the rest of the chain

Common use cases:

- HTTP middleware pipelines (auth, logging, validation, compression)
- input validation and normalization pipelines
- UI event bubbling (child to parent)
- command processing with fallbacks

Worked example 1 — classic setNext() middleware chain

```
// Base handler with default pass-through
class Handler {
    setNext(next) { this.next = next; return next; }
    handle(req) {
        if (this.next) return this.next.handle(req);
        return { ok: true, req };
    }
}

class AuthHandler extends Handler {
    handle(req) {
        if (!req.user) return { ok: false, error: 'Unauthenticated' };
        return super.handle(req);
    }
}

class LogHandler extends Handler {
    handle(req) {
        console.log('REQUEST', { path: req.path, user: !!req.user });
        return super.handle(req);
    }
}

class ValidationHandler extends Handler {
    handle(req) {
        if (typeof req.body?.amount !== 'number') {
            return { ok: false, error: 'Invalid amount' };
        }
        return super.handle(req);
    }
```

```
}

// build chain: log -> auth -> validate
const log = new LogHandler();
const auth = log.setNext(new AuthHandler());
auth.setNext(new ValidationHandler());

// client talks only to chain head
console.log(log.handle({ path: '/pay', user: null, body: { amount: 10 } }));
console.log(log.handle({ path: '/pay', user: { id: 'u1' }, body: { amount: 10 } }));
console.log(log.handle({ path: '/pay', user: { id: 'u1' }, body: { amount: 'NaN' } }));
```

Why it works: Each handler focuses on one concern and either returns a result or forwards the request. The client is unaware of chain internals.

Worked example 2 — widget bubbling with dynamic dispatch

```
class Event {
    constructor(name) { this.name = name; }
    toString() { return this.name; }
}

class Widget {
    constructor(parent = null) { this.parent = parent; }
    handle(event) {
        const method = `handle_${event.name}`;
        if (typeof this[method] === 'function') {
            this[method](event);
        } else if (this.parent) {
            this.parent.handle(event);
        } else if (typeof this.handle_default === 'function') {
            this.handle_default(event);
        }
    }
}

class MainWindow extends Widget {
    handle_close(evt) { console.log(`Main Window: ${evt}`); }
    handle_default(_evt) { console.log('Main Window: default'); }
}

class SendDialog extends Widget {
    handle_paint(evt) { console.log(`Send Dialog: ${evt}`); }
}

class MsgTxt extends Widget {
    handle_down(evt) { console.log(`Msg Event: ${evt}`); }
}

// build parent chain: MsgTxt -> SendDialog -> MainWindow
const mw = new MainWindow();
const sd = new SendDialog(mw);
const msg = new MsgTxt(sd);
```

```
for (const name of ['down', 'paint', 'unhandled', 'close']) {
    const evt = new Event(name);
    console.log(`sending event ${evt} to main window`);
    mw.handle(evt);
    console.log(`sending event ${evt} to send dialog`);
    sd.handle(evt);
    console.log(`sending event ${evt} to msg txt`);
    msg.handle(evt);
}
```

Why it works: Each widget handles only what it knows; otherwise the event bubbles to its parent, ultimately reaching a default handler.

Edge cases and trade-offs

- chain order matters: document and test the sequence
- debugging flow: add structured logging/ids to trace which handler acted
- termination: define clear return values to stop the chain when done
- performance: very long chains add overhead — measure performance
- dedicated errors: decide whether to throw or return error objects