Chain of Responsibility (CoR) design pattern

The **Chain of Responsibility (CoR)** design pattern allows a request to pass through a chain of handlers where each handler decides either to process the request or pass it to the next handler.

🧠 Definition

CoR decouples the sender of a request from its receivers by giving more than one object a chance to handle the request.

🏦 Real-World Analogy

Customer Support System
When you raise a support ticket:

1. First handled by a bot.
2. If unresolved, passed to agent.
3. Further escalated to manager if needed.

This is **exactly how CoR works**.

✅ Participants

Role	Description
Handler	Defines an interface for handling a request.
ConcreteHandler	Implements the handling and chaining logic
Client	Starts the chain by sending request to the first handler

---

✅ Java Implementation: Support System

1. Abstract Handler

```
abstract class SupportHandler {
    protected SupportHandler next;

    public void setNext(SupportHandler next) {
        this.next = next;
    }

    public void handleRequest(String level, String message) {
        if (next != null)
            next.handleRequest(level, message);
    }
}
```

---

2. Concrete Handlers

```
class BotSupport extends SupportHandler {
    public void handleRequest(String level, String message) {
        if (level.equalsIgnoreCase("LOW")) {
            System.out.println("Bot handled: " + message);
        } else {
            super.handleRequest(level, message);
        }
    }
}

class AgentSupport extends SupportHandler {
    public void handleRequest(String level, String message) {
        if (level.equalsIgnoreCase("MEDIUM")) {
            System.out.println("Agent handled: " + message);
        } else {
            super.handleRequest(level, message);
        }
    }
}

class ManagerSupport extends SupportHandler {
    public void handleRequest(String level, String message) {
        if (level.equalsIgnoreCase("HIGH")) {
            System.out.println("Manager handled: " + message);
        } else {
            System.out.println("Unresolved request: " + message);
        }
    }
}
```

---

3. Client Code

```
java
CopyEdit
public class SupportSystem {
    public static void main(String[] args) {
        SupportHandler bot = new BotSupport();
        SupportHandler agent = new AgentSupport();
        SupportHandler manager = new ManagerSupport();

        bot.setNext(agent);
        agent.setNext(manager);

        bot.handleRequest("LOW", "Reset password");
        bot.handleRequest("MEDIUM", "Order not delivered");
        bot.handleRequest("HIGH", "Account hacked");
        bot.handleRequest("UNKNOWN", "Some weird request");
    }
}
```

✅ Output

```
Bot handled: Reset password
Agent handled: Order not delivered
Manager handled: Account hacked
```

```
Unresolved request: Some weird request
```

✅ Where in Spring Boot can you apply Chain of Responsibility?

✅ 1. Request Filtering using Spring Security Filters

Each security filter (like JWT filter, CORS filter, Authentication filter) passes the request down the chain.

`OncePerRequestFilter → JwtFilter → AuthenticationFilter → CustomFilter`

✅ 2. Spring Boot Exception Handling

Exception resolvers (like `HandlerExceptionResolver`) pass unhandled exceptions to the next resolver.

✅ 3. Interceptor Chain

Spring MVC's `HandlerInterceptor`s form a chain. Each one can pre-handle/post-handle requests.

✅ Spring Boot Custom Chain Example (Payment Approval)

1. Abstract Handler

```
public abstract class Approver {
    protected Approver next;

    public void setNext(Approver next) {
        this.next = next;
    }

    public abstract void approvePayment(int amount);
}
```

2. Concrete Approvers

```
@Component
public class Clerk extends Approver {
    public void approvePayment(int amount) {
        if (amount <= 1000) {
            System.out.println("Clerk approved payment of ₹" + amount);
        } else if (next != null) {
            next.approvePayment(amount);
        }
    }
}

@Component
public class Manager extends Approver {
    public void approvePayment(int amount) {
        if (amount <= 5000) {
            System.out.println("Manager approved payment of ₹" + amount);
```

```
        } else if (next != null) {
            next.approvePayment(amount);
        }
    }
}

@Component
public class Director extends Approver {
    public void approvePayment(int amount) {
        if (amount > 5000) {
            System.out.println("Director approved payment of ₹" + amount);
        }
    }
}
```

3. Chain Configuration

```
public class ApprovalChainConfig {

    @Bean
    public Approver approvalChain(Clerk clerk, Manager manager, Director
director) {
        clerk.setNext(manager);
        manager.setNext(director);
        return clerk;
    }
}
```

4. Use in Service

```
@Service
public class PaymentService {

    @Autowired
    private Approver approver;

    public void processPayment(int amount) {
        approver.approvePayment(amount);
    }
}
```

5. Controller

```
@RestController
@RequestMapping("/payments")
public class PaymentController {

    @Autowired
    private PaymentService paymentService;

    @PostMapping
    public String makePayment(@RequestParam int amount) {
        paymentService.processPayment(amount);
        return "Request Processed";
    }
}
```

✅ Summary

Feature	Benefit
Dynamic processing chain	Add/remove handlers at runtime
Separation of concerns	Each handler focuses on a specific logic
Scalable and flexible design	Extend without modifying code