Object Oriented Analysis and Design with Java

UE19CS353

Prof.J.Ruby Dinakar
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE19CS353: Object Oriented Analysis and Design with Java

OO Design Patterns & Anti-Patterns with Sample implementation in Java

Prof. J.Ruby Dinakar

Department of Computer Science and Engineering

PES UNIVERSITY ONLINE

Unit-5

Behavioral Patterns – Chain of Responsibility Pattern

Behavioral patterns

Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns.

These patterns increase flexibility in carrying out this communication.

Behavioral patterns influence how state and behavior flow through a system.

By optimizing how state and behavior are transferred and modified, you can simplify, optimize, and increase the maintainability of an application

Object Oriented Analysis and Design with Java

List of common behavioral design patterns

Chain of responsibility: Command objects are handled or passed on to other objects by logic-containing processing objects

Command: Command objects encapsulate an action and its parameters

Interpreter: Implement a specialized computer language to rapidly solve a specific set of problems

Iterator: Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation

Mediator: Provides a unified interface to a set of interfaces in a subsystem

Memento: Provides the ability to restore an object to its previous state (rollback)

Observer: also known as Publish/Subscribe or Event Listener. Objects register to observe an event that may be raised by another object

State: A clean way for an object to partially change its type at runtime

Strategy: Algorithms can be selected on the fly

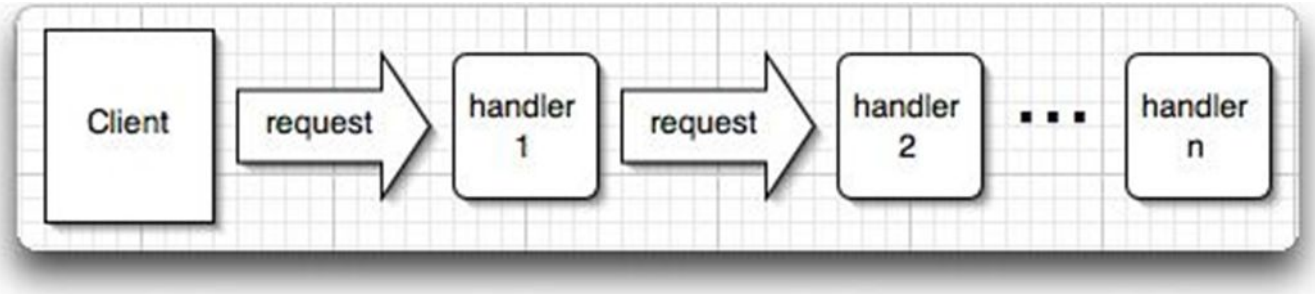
Template Method: Describes the program skeleton of a program

Visitor: A way to separate an algorithm from an object

Chain of Responsibility Pattern

The Chain of Responsibility pattern establishes a chain within a system, so that a message can either be handled at the level where it is first received, or be directed to an object that can handle it.

it is used to manage algorithms, relationships and responsibilities between objects



Intent

Avoid coupling sender of request to its receiver by giving more than one object a chance to handle request.

Chain receiving objects and pass request along until an object handles it.

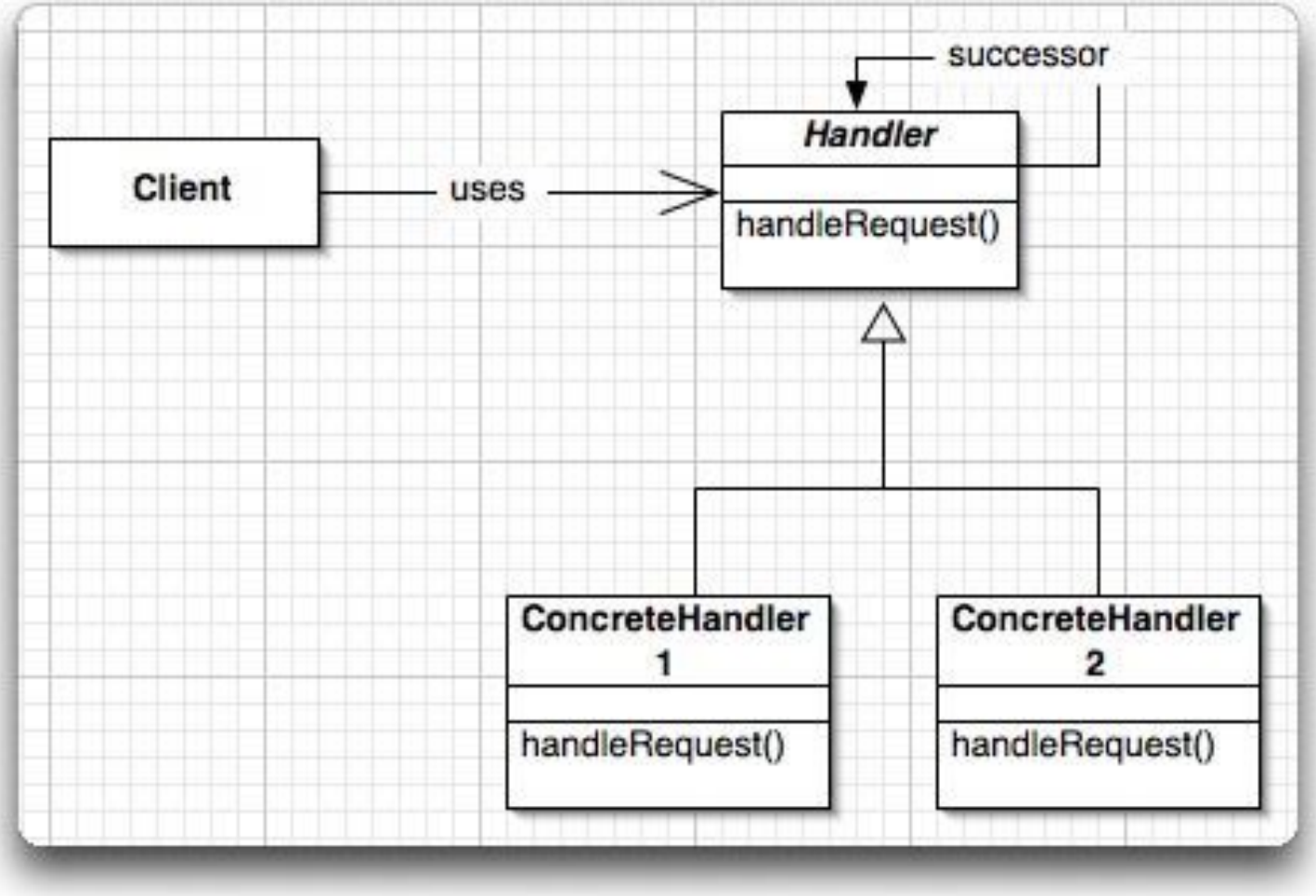
Motivation

The Chain of Responsibility is intended to promote loose coupling between the sender of a request and its receiver by giving more than one object an opportunity to handle the request.

The receiving objects are chained and pass the request along the chain until one of the objects handles it.

The set of potential request handler objects and the order in which these objects form the chain can be decided dynamically at runtime by the client depending on the current state of the application.

Structure


successor
Handler
handleRequest()
Client
uses
ConcreteHandler 1
handleRequest()
ConcreteHandler 2
handleRequest()

Participants

Handler:

- This can be an interface which will primarily receive the request and dispatches the request to a chain of handlers. It has reference to the only first handler in the chain and does not know anything about the rest of the handlers.

ConcreteHandler:

- handles requests it is responsible for
- can access its successor
- if it does not handle the request, forwards the request to its successor

Client:

- initiates the request to a ConcreteHandlerobject on the chain

Example – Scenario 1

Scenario

An enterprise has been getting more email than they can handle. The enterprise gets 4 types of e-mail.

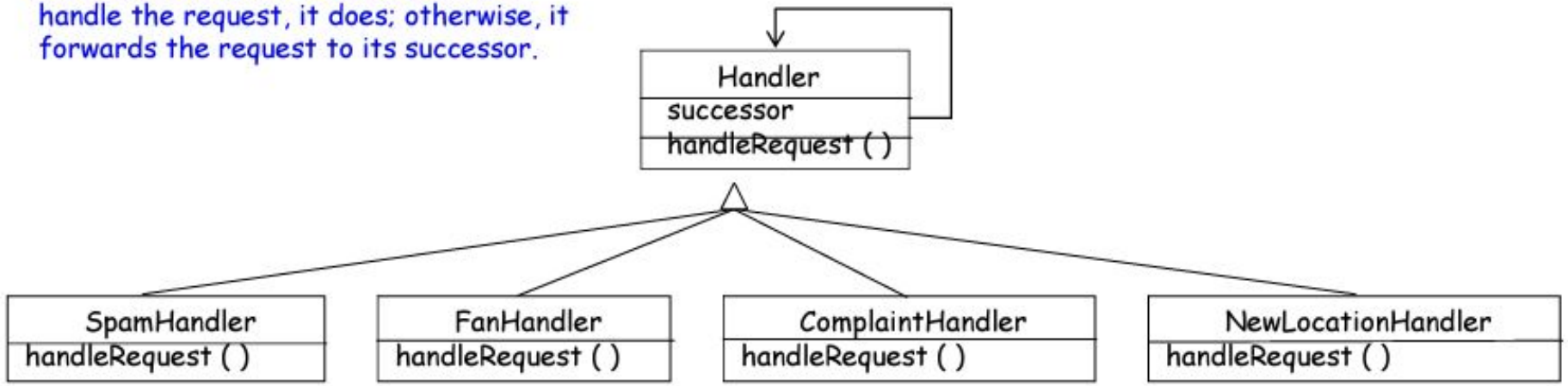
They are:

- Fan mail with complements,
- Complaints,
- Requests for new features,
- Spam.

Your task:

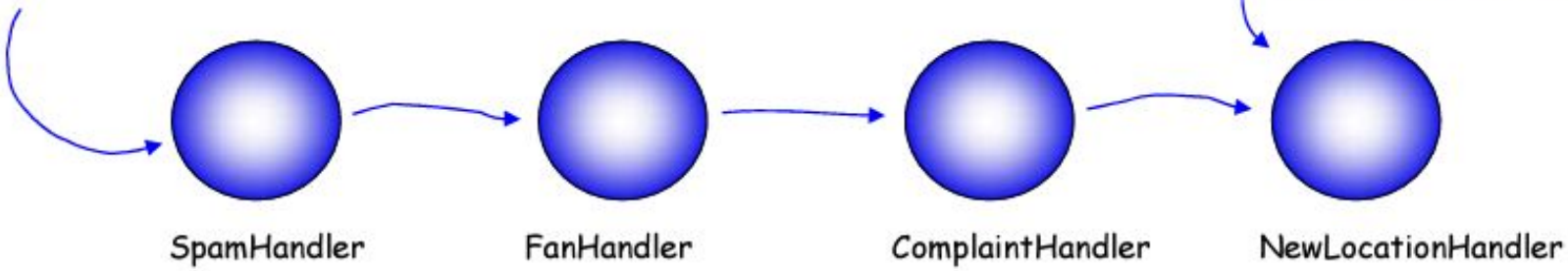
- The enterprise has already written the AI detectors that can tell whether an email is fan, complaint, request or spam,
- You need to create a design that can use the detectors to handle incoming email.

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



Each email is passed to the first handler

Email is not handled if it falls off the end of the chain -- although, you can always implement a catch-all handler



Consequences

Reduced Coupling

- Objects are free from knowing what object handles the request

Added Flexibility in assigning responsibilities to objects

- Can change chain at runtime
- Can subclass for special handlers

Receipt is guaranteed

- Request could fall off the chain
- Request could be dropped with bad chain

Applicability

Use Chain of Responsibility

when

- More than one object may handle a request and the handler isn't known a priori.
- You want to issue a request to one of several objects without specifying the receiver explicitly
- The Set of objects than can handle a request should be specified dynamically

Implementation – Example Scenario 2

Let us consider the transaction approval process in a company. Suppose we want to approve the transactions based on certain conditions?

For instance, in the transaction approval application user keys in transaction details into the application and this transaction need to be processed by any one of the higher level employee in the company.

All transactions lesser than 1,00,000 amount can be approved by manager, lesser than 10,00,000 can be approved by vice president and lesser than 25,00,000 can be approved by CEO.

The transaction approval behavior can be implemented with a simple if else conditions, by checking if amount is less than 1,00,000 or else if amount is less than 10,00,000 and so on. But this approach is static, means we can not change these conditions dynamically. Chain of responsibility design pattern can be used in this situation.

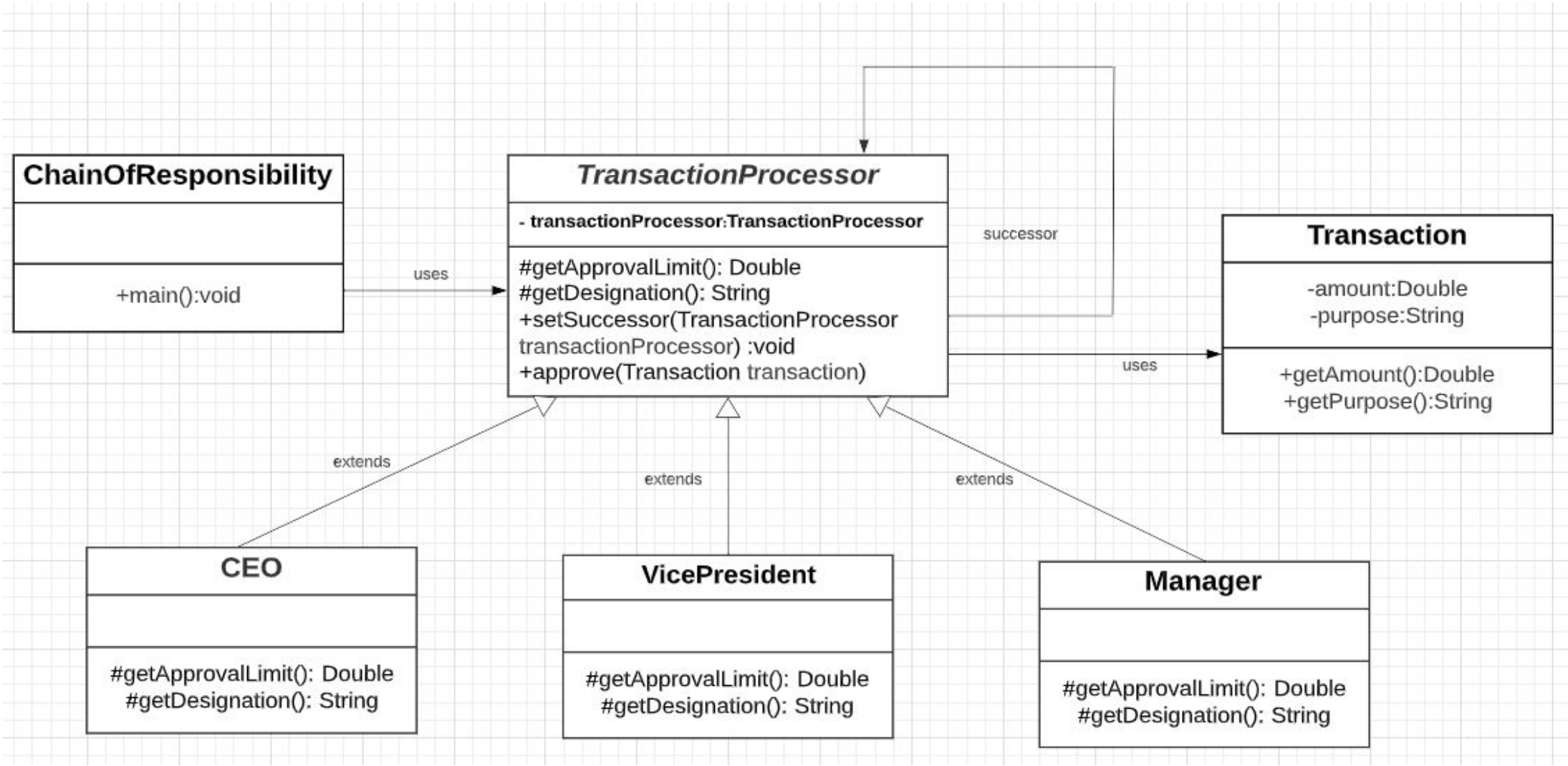
In the transaction processing application, approval process acts like a chain of responsibilities.

First, manager acts on the transaction, if the amount is higher than his approval limit then the transaction is transferred to vice president and so on.

We need to design processing and command objects. Based on the above example, processing objects are employees like Manager, Vice President because they process the transaction by approving and command object is the transaction. Once processing objects are created then we need to chain them together like manager -> vice president -> CEO and pass the transaction at the beginning of the chain. Transaction continues to flow in the chain until it reaches the employee, who's limit allows to approve it.

Object Oriented Analysis and Design with Java

UML Class Diagram



First create an abstract class for processing the transactions.

If the transaction can be processed then it should be passed to its successor so this class store

its next successor object.

If transaction is with in the limit then it prints the message.

If it is out of range then passes it to next processor by calling ‘approve’ method,

before calling approve check for null if processor request is last in the chain.


```
abstract class TransactionProcessor
{
    private TransactionProcessor transactionProcessor;
    abstract protected Double getApprovalLimit();
    abstract protected String getDesignation();
    public void setSuccessor(TransactionProcessor transactionProcessor)
    {
        this.transactionProcessor = transactionProcessor;
    }
    public void approve(Transaction transaction)
    {
       if(transaction.getAmount() <=0.0 )
        {
            System.out.println("Invalid Amount. Amount should be > 0");
            return;
        }
        if(transaction.getAmount() <= getApprovalLimit())
        {
            System.out.println("Transaction for amount "+transaction.getAmount()+ " approved by "+getDesignation());
        }
        else
        {
            if(transactionProcessor == null)
             {
                System.out.println("Invalid Amount. Amount should not exceed 25lacs!");
                return;
              }
              transactionProcessor.approve(transaction);
```

Now we need to create concrete class for each approver (Manager, Vice President, CEO) and this class will extend TransactionProcessor.

Each processing class sets its limit and designation.

```
class Manager extends TransactionProcessor{

    @Override
    protected Double getApprovalLimit() {
        return 100000.0;
    }

    @Override
    protected String getDesignation() {
        return "manager";
    }
}
```


```
class VicePresident extends TransactionProcessor{

    @Override
    protected Double getApprovalLimit() {
        return 1000000.0;
    }

    @Override
    protected String getDesignation() {
        return “Vice President";
    }
}
```


```
class CEO extends TransactionProcessor{

    @Override
    protected Double getApprovalLimit() {
        return 2500000.0;
    }

    @Override
    protected String getDesignation() {
        return "CEO";
    }
}
```

Next Create a transaction class.

```
class Transaction{
    private Double amount;
    private String purpose;

    Transaction(Double amount, String purpose){
        this.amount = amount;
        this.purpose = purpose;
    }
    public Double getAmount() {
        return amount;
    }
    public String getPurpose() {
        return purpose;
    }
}
```

Now let us glue all these classes together to form chain of responsibility pattern using ChainOfResponsibility class.

- Create processing objects. We have three processing objects in the chain.
- Chain together all the processing objects. Order of the objects is important because manager handover transaction to vice president.
- Because the chain starts from the manager so transactions are pushed using the manager instance.

```
public class ChainOfResponsibility {
public static void main(String[] args) {
        Manager manager = new Manager();
        VicePresident vicePresident = new VicePresident();
        CEO ceo = new CEO();
        manager.setSuccessor(vicePresident);
        vicePresident.setSuccessor(ceo);
        manager.approve(new Transaction(2600000.0, "general"));
        manager.approve(new Transaction(50000.0, "general"));
        manager.approve(new Transaction(120000.0, "general"));
        manager.approve(new Transaction(1500000.0, "general"));
        manager.approve(new Transaction(0.0, "general"));
}
}
```

Creating Chain

manager class returns its approval limit amount. TransactionProcessor class approve method checks whether the given amount is <= approval limit. If it returns false it propagates the request to the successor VicePresident class to get its approval limit and so on. In case the request can not be serviced then null object will be return for the amount exceeding 25lacs.

Output

```
Invalid Amount. Amount should not exceed 25lacs!
Transaction for amount 50000.0 approved by manager
Transaction for amount 120000.0 approved by vice president
Transaction for amount 1500000.0 approved by CEO
Invalid Amount. Amount should be > 0
```

Benefits, Uses and Drawbacks

Benefits:

- Decouples the sender of the request and its receivers,
- Simplifies your object because it doesn't have to know the chain's structure and keep direct reference to its members,
- Allows you to add or remove responsibilities dynamically by changing the members or the order of the chain.

Uses:

- Commonly used in Windows systems to handle events like mouse clicks and keyboard events.
- In java it is used in handling chain of Exceptions

Drawbacks:

- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage),
- Can be hard to observe the runtime characteristics and debug.

References

Text Reference

Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

https://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP17-ChainOfResponsibility.html
https://refactoring.guru/design-patterns
https://thetechstack.net/chain-of-responsibility-design-pattern/

THANK YOU

J.Ruby Dinakar

Department of Computer Science and Engineering

rubydinakar@pes.edu