The chain of responsibility pattern is a pattern with different people who have different authority or power, for example imagine you have a group of friends. You ask the first one for help. If they can't help you then they pass your request to the next friend until someone actually can do it. It's like a team of friends who work together to solve a problem.

```
interface ExpensivePizzaHandler{
    void handleRequest(double money);
}
```

I have declared an interface ExpensivePizzaHandler which has a method handlerequest. The concrete handlers need to implement this method. I have created this method to handle purchase requests.

```
class Manager implements ExpensivePizzaHandler{
    private static final double APPROVAL_LIMIT = 1000;

    @Override
    public void handleRequest(double amount){
        if (amount<=APPROVAL_LIMIT){
            System.out.println("Manager appproves the purchase of 50 pizza.");
        }
        else{
            System.out.println("Manager can not approve the request. Asking the Director.");
            new Director().handleRequest(amount);
        }
    }

}
```

```
class Director implements ExpensivePizzaHandler{
    private static final double APPROVAL_LIMIT = 4000;

    @Override
    public void handleRequest(double amount){
        if (amount <= APPROVAL_LIMIT){
            System.out.println("Director can approve request for the 1000.");
        }
        else{
            System.out.println("Director can not approve request. Asking the President");
            new President().handleRequest(amount);
        }
    }
}

class President implements ExpensivePizzaHandler{
    private static final double APPROVAL_LIMIT = 100000;

    @Override
    public void handleRequest(double amount){
        if (amount <= APPROVAL_LIMIT){
            System.out.println("The president approves the request for the 1000 pizzas.");
        }
        else{
            System.out.println("The request has been denied by all authorities of the company");
        }
    }
}
```

Here I have created three concrete handlers. Manager,Director and President. They all implement the ExpensivePizzaHandler interface.

Private static final double APPROVAL_LIMIT: This line means that it is a private constant which can not be changed and it is shared among all instances of the class. So what happens in these concrete handlers, when the first handler can approve the request then it will approve it otherwise it will pass the request to the next handler. Also to mention in the else statement I have a line like for the first handler its new President().handleRequest(amount) : This line is responsible for creating a new instance using the new keyword it invokes the constructor to initialize the object. The .handleRequest(amount) : this line will immediately call the method passing the amount as an argument. The handleRequest is a part of the interface and each concrete class will have their own implementation on how to use it. In simple terms. It will pass the request to the next handler for approval.

There are many benefits of using the chain of command pattern like for my case Flexibility and Extensibility. This means that I can modify the handlers without modifying the client code or existing handlers. For example in my case, I could introduce a new authority with a different approval rate by creating a new class. If i were not to use it then I would need to deal with if-else or switch statements in the client code to determine which authority should handle a specific purchase request, this approach could lead to a code which is harder to read, maintain and extend.
Dynamic handling and Scalability, which means that requests are automatically passed through a chain until handled or denied which allows dynamic handling. The approval criteria can also increase without changing the client code. Without this I would need to hardcode the logic which means that the client code could result in a less scalable and rigid system. With the chain of responsibility the client code is decoupled from the details of how requests are processed and who handles them. Changes to the approval process can be localized within the appropriate handler class which could promote maintainability.
Without this direct control could lead to tight coupling with the approval logic. Modifications to the approval process might require changes scattered throughout the client code making maintenance more error-prone.