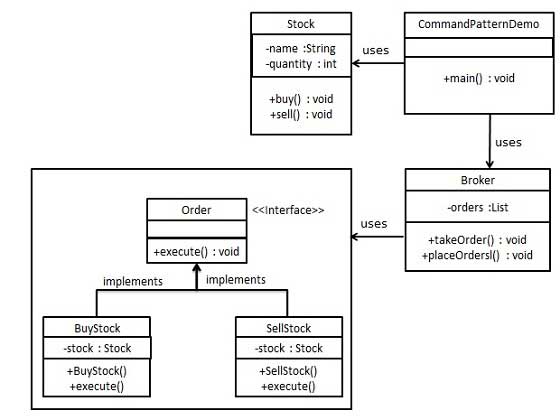
Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

## Implementation

We have created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders.

*Broker* object uses command pattern to identify which object will execute which command based on the type of command. *CommandPatternDemo*, our demo class, will use *Broker*class to demonstrate command pattern.



\* broker decides the action , as buystock , sellStock cannot buy/sell themselves.

Same as textbox widget cannot decide what should be done on content, content will be passed to next application object.

import java.util.ArrayList;

import java.util.List;

/\*

\* step 1

\* create a command interface

\* Order.java

\*/

interface Order{

void execute();

}

/\*

\* step 2

\* create a request class

\* Stock.java

\*/

class Stock{

private String name = "ABC";//data supposed to come from database

private int quantity = 10;

public void buy() {

System.out.println("Stock [ Name: "+name+", Quantity: "+ quantity +"] bought ");

}

public void sell() {

System.out.println("Stock [ Name: "+name+", Quantity: "+ quantity +"] sold ");

}

}

/\*

\* step 3

\* create concrete classes implementing the order interface

\* BuyStock

\*/

class BuyStock implements Order{

private Stock abcStock;

public BuyStock(Stock abcStock) {

this.abcStock = abcStock;

}

public void execute() {

abcStock.buy();

}

}

/\*

\* SellStock.java

\*/

class SellStock implements Order{

private Stock abcStock;

public SellStock(Stock abcStock) {

this.abcStock = abcStock;

}

@Override

public void execute() {

abcStock.sell();

}

}

/\*

\* step 4

\* create command invoker class

\* Broker

\*/

class Broker{

private List<Order> orderList = new ArrayList<>();

public void takeOrder(Order order) {

orderList.add(order);

}

public void placeOrders() {

for(Order order : orderList) {

order.execute();

}

}

}

/\*

\* step 5

\* use broker class to take and execute commands

\*/

public class TestCommandDemo {

public static void main(String[] args) {

Stock abcStock = new Stock();

BuyStock buyStockOrder = new BuyStock(abcStock);

SellStock sellStockOrder = new SellStock(abcStock);

Broker broker = new Broker();

broker.takeOrder(buyStockOrder);

broker.takeOrder(sellStockOrder);

broker.placeOrders();

}

}