**THE MEMENTO PATTERN**

**In main.java**

// THE MEMENTO PATTERN  
 var editor = new Editor();  
 var history = new History();  
  
 editor.setContent("a");  
 history.push(editor.createState());  
  
 editor.setContent("b");  
 history.push(editor.createState());  
  
 editor.setContent("c");  
 editor.restore(history.pop());  
 editor.restore(history.pop());  
  
 System.*out*.println(editor.getContent());  
}

**In Editor (aka *Originator*).java**

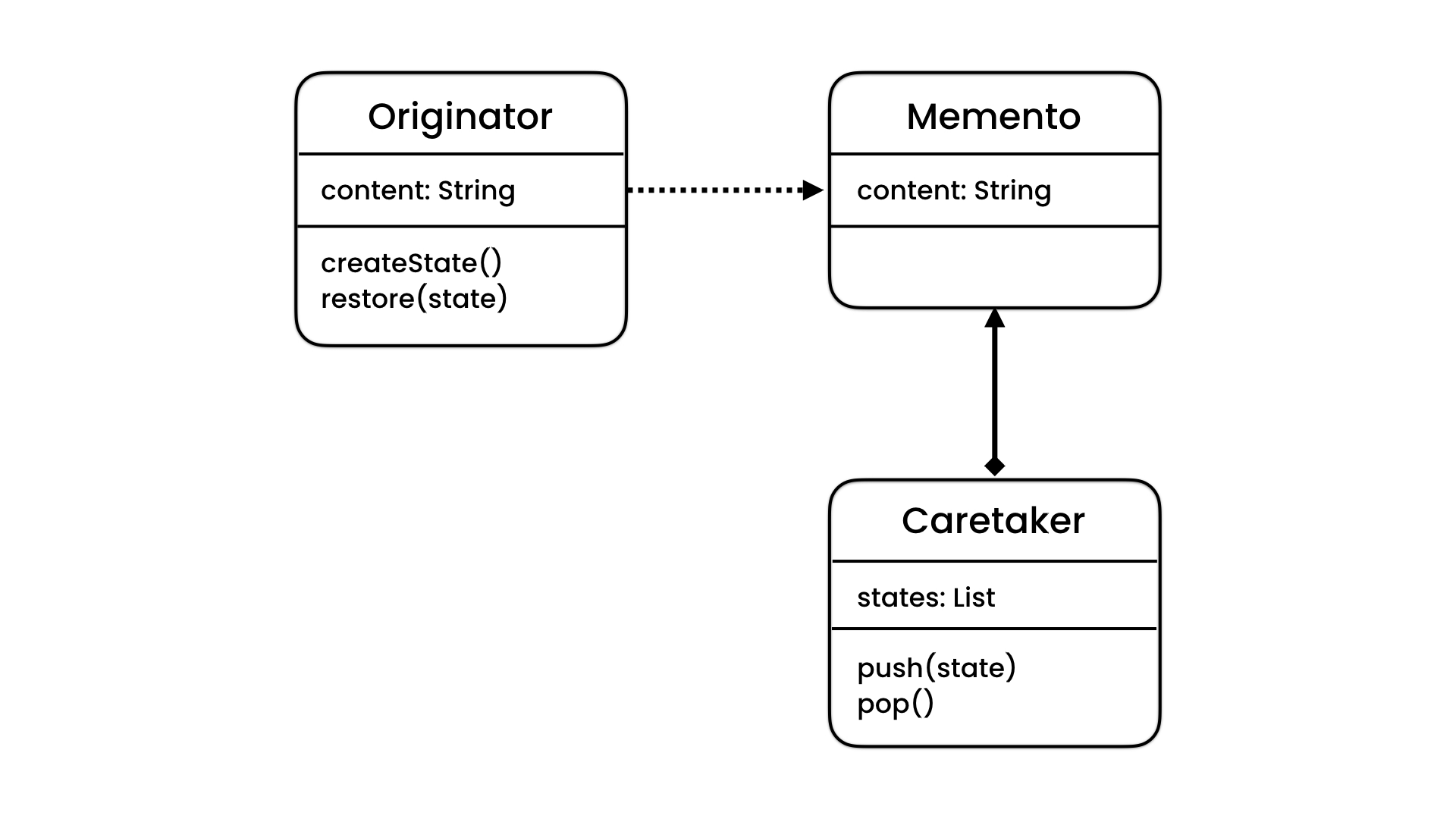
public class Editor {  
 private String content;  
  
 public EditorState createState(){  
 return new EditorState(this.content);  
 }  
  
 public void restore(EditorState state){  
 content = state.getContent();  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 }  
}

**In EditorState (aka *Memento*).java**

public class EditorState {  
 private final String content;  
  
 public EditorState(String content){  
 this.content = content;  
 }  
  
 public String getContent() {  
 return this.content;  
 }  
}

**In History (aka *Caretaker*).java**

import java.util.ArrayList;  
import java.util.List;  
  
public class History {  
 private List<EditorState> states = new ArrayList<>();  
  
 public void push(EditorState state){  
 states.add(state);  
 }  
  
 public EditorState pop(){  
 var lastIndex = states.size() -1;  
 var lastState = states.get(lastIndex);  
 states.remove(lastState);  
 return lastState;  
 }  
}

**’**

**THE ITERATOR PATTERN**

**Inside main.js** (code here will always be the same as implementation logic is abstracted away in another class)

var iteratorHistory = new BrowseHistory();  
iteratorHistory.push("a");  
iteratorHistory.push("b");  
iteratorHistory.push("c");  
  
Iterator iterator = iteratorHistory.createIterator();  
while(iterator.hasNext()){  
 System.*out*.println(iterator.current());  
 iterator.next();  
}

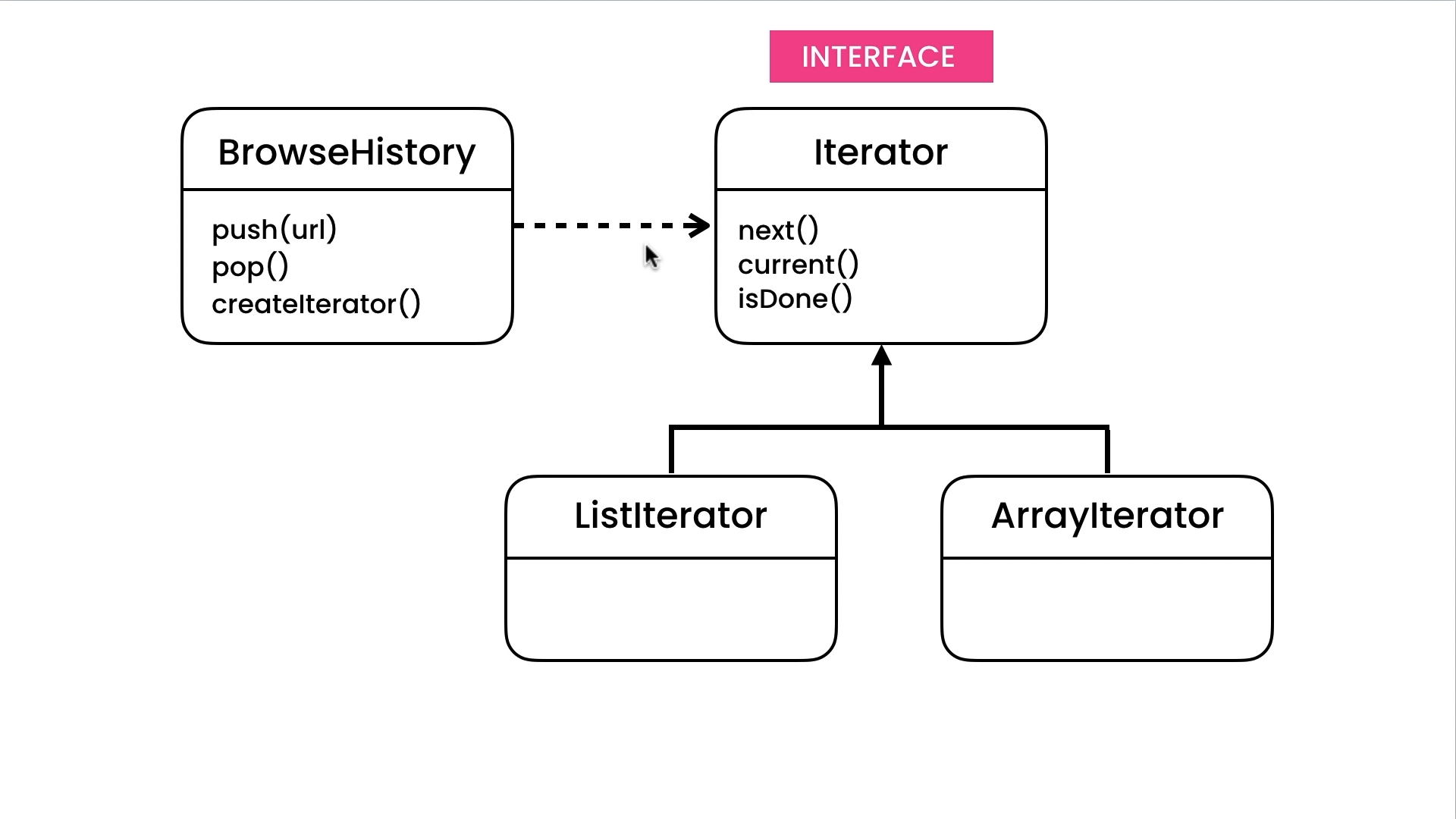
**Inside Iterator.js** (Iterator, interface)

public interface Iterator {  
 boolean hasNext();  
 String current();  
 void next();  
}

**Inside BrowseHistory.js** (when collection is a List)  
public class BrowseHistory {  
 private List<String> urls = new ArrayList<>();  
 private int index = 0;  
  
 public void push(String url){ urls.add(url); }  
  
 public String pop(){  
 var lastIndex = urls.size() -1;  
 var lastUrl = urls.get(lastIndex);  
 urls.remove(lastUrl);  
 return lastUrl;  
 }  
  
 public List<String> getUrls() { return urls; }  
  
 public void setUrls(List<String> urls) { this.urls = urls; }  
  
 public Iterator createIterator(){ return new ListIterator(this); }  
  
 public class ListIterator implements Iterator{  
 public BrowseHistory history;  
  
 public ListIterator(BrowseHistory history){ this.history = history; }  
  
 @Override  
 public boolean hasNext() { return (index < history.getUrls().size()); };  
  
 public String current() { return history.getUrls().get(index); }  
  
 public void next() { index++; }  
 }

}

**Inside BrowseHistory.js** (when collection is a Stack aka array of fixed length)  
public class BrowseHistory {  
 private String[] urls = new String[10];  
 private int index = 0;  
 private int count = 0;  
  
 public void push(String url){ urls[count++] = url; }  
  
 public String pop(){ return urls[--count]; }  
  
 public String[] getUrls() { return urls; }  
  
 public void setUrls(String[] urls) { this.urls = urls; }  
  
 public Iterator createIterator(){ return new ArrayIterator(this); }  
  
  
 public class ArrayIterator implements Iterator{  
 public BrowseHistory history;  
  
 public ArrayIterator(BrowseHistory history){ this.history = history; }  
  
 public boolean hasNext() { return (index < history.count); }  
  
 public String current() { return history.urls[index]; }  
  
 public void next() { index++; }  
 }  
}



**THE STRATEGY PATTERN**

**Inside main.java**

var imageStorage = new ImageStorage();  
imageStorage.store("image1",new JpegCompressor(), new BandWFilter());  
imageStorage.store("image1",new PngCompressor(), new BandWFilter());

**Inside ImageStorage.java** (the Context)

var imageStorage = new ImageStorage();  
imageStorage.store("image1",new JpegCompressor(), new BandWFilter());  
imageStorage.store("image1",new PngCompressor(), new BandWFilter());

**Inside Compressor.java** (interface 1)

public interface Compressor {  
// byte[] compress(byte[] image)  
 void compress(String fileName);  
}

**Inside Filter.java** (interface 2)  
public interface Filter {  
 void apply(String fileName);  
}

**Inside JpegCompressor.java** (1st strategy for interface 1)

public interface Filter {  
 void apply(String fileName);  
}

**Inside PngCompressor.java** (2nd strategy for interface 1)

public class PngCompressor implements Compressor{  
 @Override  
 public void compress(String fileName) {  
 System.*out*.println("Compressing "+ fileName +" using PNG format");  
 }  
}

**Inside BlackAndWhiteFilter.java** (1st strategy for interface 2)

public class BandWFilter implements Filter{  
 @Override  
 public void apply(String fileName) {  
 System.*out*.println("Filtering " + fileName + " using the Black and White filter");  
 }  
}

**Inside RGBFilter.java** (2nd strategy for interface 2)

public class RGBFilter implements Filter{  
 @Override  
 public void apply(String fileName) {  
 System.*out*.println("Filtering " + fileName + " using the RGB filter");  
 }  
}

**THE TEMPLATE METHOD PATTERN**

**Inside main.java**

var task = new TransferMoneyTask();  
task.execute();  
var task2 = new GenerateReportTask();  
task.execute();

**Inside Auditrail.java** (Just a prereq method that any task we define must/needs to use)

public class AuditTrail {  
 public void record(){  
 System.*out*.println("Audit recorded !");  
 }  
}

**Inside Task.java** (the common code containing parent abstract class)

package com.ankit.templateMethod;  
  
public abstract class Task {  
 public AuditTrail auditTrail;  
  
 public Task(){  
 this.auditTrail = new AuditTrail();  
 }  
  
 public Task(AuditTrail auditTrail){  
 this.auditTrail = auditTrail;  
 }  
  
 public void execute(){  
 auditTrail.record();  
 System.*out*.println("Starting execution of transaction......");  
  
 doExecute();  
 }  
  
 protected abstract void doExecute();  
}

**Inside MoneyTransfer.java** (A task (defined by us) which **extends** the parent abstract class)

public class TransferMoneyTask extends Task{  
// public TransferMoneyTask(AuditTrail auditTrail){  
// super(auditTrail);  
// }  
 @Override  
 protected void doExecute() {  
 System.*out*.println("Your money has been successfully transferred !");  
 }  
}

**Inside GenerateReport.java** (A task (defined by us) which **extends** the parent class)

public class GenerateReportTask extends Task{  
 @Override  
 protected void doExecute() {  
 System.*out*.println("Your report has been successfully generated !");  
 }  
}

**THE COMMAND PATTERN**

**Inside fx/Button.java** (Invoker)

public class Button {  
 private String label;  
 private Command command;  
  
 public Button(Command command) {  
 this.command = command;  
 }  
  
 public void click() {  
 command.execute();  
 }  
  
 public String getLabel() {  
 return label;  
 }  
  
 public void setLabel(String label) {  
 this.label = label;  
 }  
}

**Inside fx/Command.java** (Command Interface)

public interface Command {  
 void execute();  
}

**Inside AddCustomerCommand.java** (ConcreteCommandA)

public class AddCustomerCommand implements Command {  
 private CustomerService customerService;  
  
 public AddCustomerCommand(CustomerService customerService) {  
 this.customerService = customerService;  
 }  
  
 @Override  
 public void execute() {  
 customerService.addCustomer();  
 }  
}

**Inside CustomerService.java** (Receiver)

public class CustomerService {  
 public void addCustomer(){  
 System.*out*.println("Adding a customer.....");  
 }  
}

**Inside main.java**

var service = new CustomerService();  
var command = new AddCustomerCommand(service);  
var button = new Button(command);  
button.click();

**IMPLEMENTING COMPOSITE COMMAND FUNCTIONALITY**

**Inside CompositeCommand.java**

public class CompositeCommand implements Command {  
 public List<Command> commands = new ArrayList<>();  
  
 public void addCommand(Command command){  
 commands.add(command);  
 }  
  
 @Override  
 public void execute() {  
 for(Command command : commands){  
 command.execute();  
 }  
 }  
}

**Inside BlackAndWhite.java**

public class BlackAndWhite implements Command {  
 @Override  
 public void execute() {  
 System.*out*.println("Black and white filter applied !");  
 }  
}

**Inside WoodenFrame.java**

public class WoodenFrame implements Command {  
 @Override  
 public void execute() {  
 System.*out*.println("Wooden frame applied !");  
 }  
}

**IMPLEMENTING THE UNDO MECHANISM**

**Inside Command.java** (top level (parent) interface)

public interface Command {  
 void execute();  
}

**Inside UndoableCommand.java** (child interface)

public interface Command {  
 void execute();  
}

**Inside HtmlDoc.java** (kind of like a state with its predef methods to restore state so we don’t have to store entire state snapshot but just a snapshot of its properties)

public class HtmlDoc {  
 public String content;  
  
 public void makeBold(){  
 this.content = "<b>" + this.content + "</b>";  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 }  
}

**Inside BoldCommand.java** (implements the undo mechanism)

public class BoldCommand implements UndoableCommand {  
 public HtmlDoc document;  
 public String prevContent;  
 public History history;  
  
 public BoldCommand(HtmlDoc document, History history) {  
 this.document = document;  
 this.history = history;  
 }  
  
 @Override  
 public void execute() {  
 prevContent = document.getContent();  
 document.makeBold();  
 history.push(this);  
 }  
  
 @Override  
 public void unexecute() {  
 document.setContent(prevContent);  
 }  
}

**Inside UndoCommand.java** (just provides a cleaner way of directly accessing history and undoing instead of calling unexecuted() on a command)

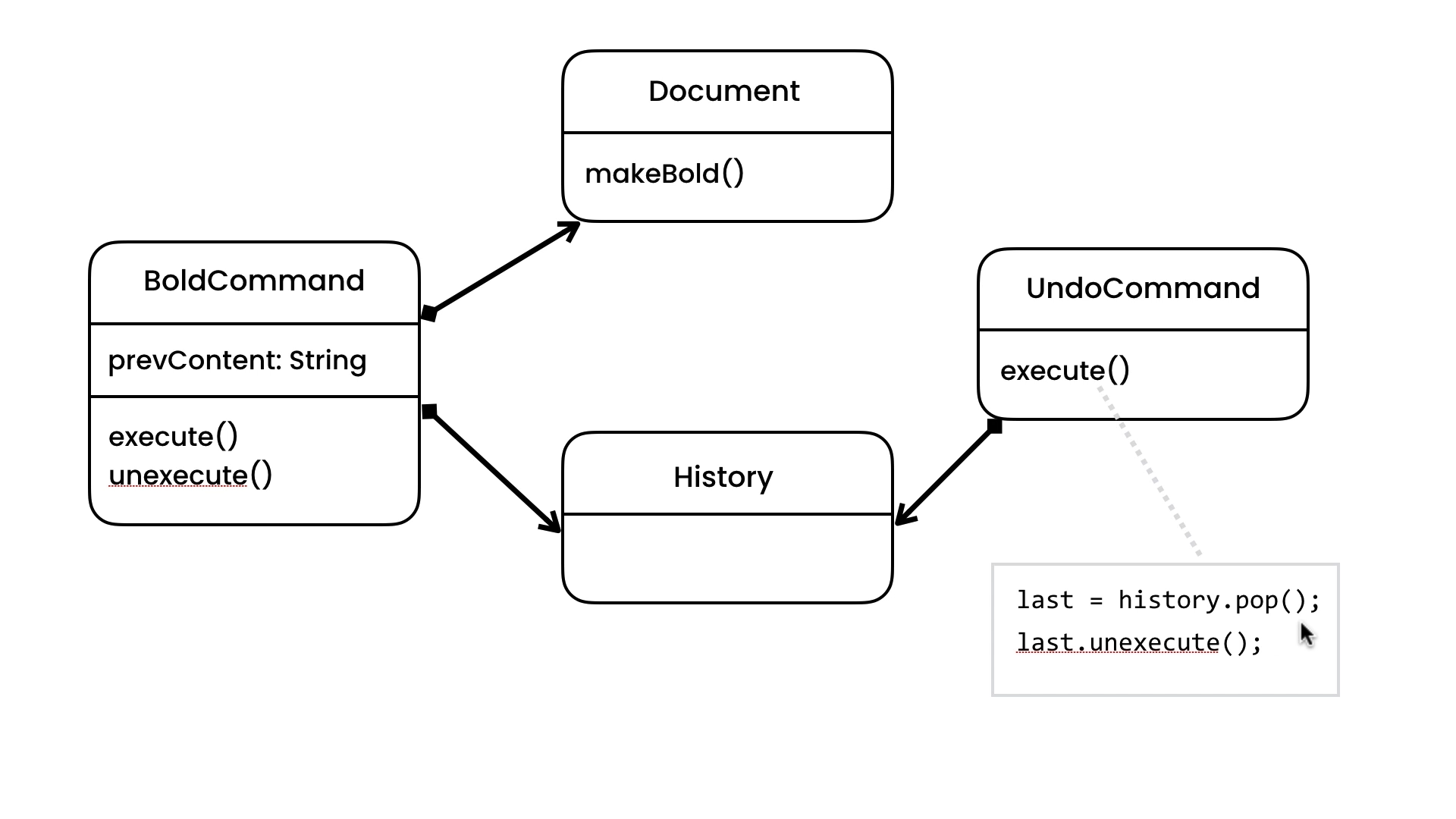
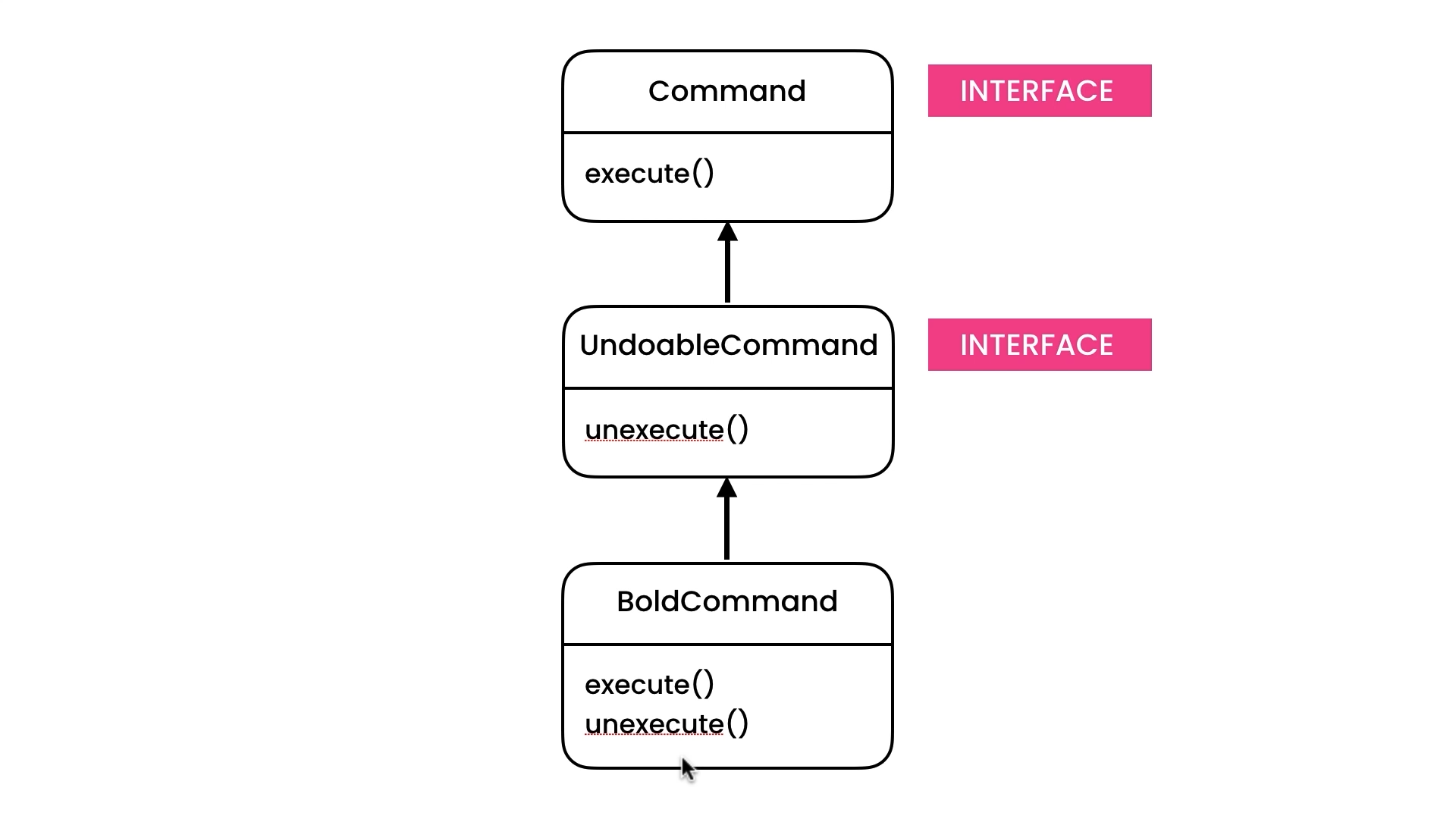
public class UndoCommand implements Command{  
 private History history;  
  
 public UndoCommand(History history) {  
 this.history = history;  
 }  
  
 @Override  
 public void execute() {  
 if (history.size()>0){  
 history.pop().unexecute();  
 }  
 }  
}

**Inside History.java** (stores and adds/deletes list of commands)

public class History {  
 private Deque<UndoableCommand> commands = new ArrayDeque<>();  
  
 public void push(UndoableCommand command){  
 commands.add(command);  
 }  
  
 public UndoableCommand pop(){  
 return commands.remove();  
 }  
  
 public Integer size(){  
 return commands.size();  
 }  
}

**Inside main.java**

var document = new HtmlDoc();  
var historyOfCmds = new com.ankit.command.undo.History();  
var boldCommand = new BoldCommand(document,historyOfCmds);  
document.setContent("Namaskar !");  
System.*out*.println(document.getContent());  
  
boldCommand.execute();  
System.*out*.println(document.getContent());  
  
var undoCommand = new UndoCommand(historyOfCmds);  
undoCommand.execute();



**THE OBSERVER PATTERN**

1. **PULL STYLE**

**Inside Subject.java** (the subject i.e. abstract parent class off of which observables are based)

public class Subject {  
 public List<Observer> observers = new ArrayList<>();  
  
 public void addObserver(Observer observer){  
 observers.add(observer);  
 }  
  
 public void removeObserver(Observer observer){  
 observers.remove(observer);  
 }  
  
 public void notifyObservers(){  
 for(var observer : observers){  
 observer.update();  
 }  
 }  
  
}

**Inside DataSrc.java** (an “observable”)

public class DataSource extends Subject {  
 private Integer value;  
  
 public Integer getValue() {  
 return value;  
 }  
  
 public void setValue(Integer value) {  
 this.value = value;  
 notifyObservers();  
 }  
}

**Inside Observer.java** (the interface off of which observers are based)

public interface Observer {  
 void update();  
}

**Inside Spreadsheet.java** (an “observer”)

public class Spreadsheet implements Observer{  
 private DataSource dataSource;  
  
 public Spreadsheet(DataSource dataSource) {  
 this.dataSource = dataSource;  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println("Spreadsheet was just updated with value :" + dataSource.getValue());  
 }  
}

**Inside Chart.java** (an “observer”)

public class Chart implements Observer{  
 private DataSource dataSource;  
  
 public Chart(DataSource dataSource) {  
 this.dataSource = dataSource;  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println("Chart was just updated with value :" + dataSource.getValue());  
 }  
}

**Inside main.java**

var dataSrc = new DataSource();  
var sheet1 = new Spreadsheet(dataSrc);  
var sheet2 = new Spreadsheet(dataSrc);  
var chart = new Chart(dataSrc);  
  
dataSrc.addObserver(sheet1);  
dataSrc.addObserver(sheet2);  
dataSrc.addObserver(chart);  
  
dataSrc.setValue(1);

1. **PUSH STYLE**

**Inside Subject.java** (the subject i.e. abstract parent class off of which observables are based)

public class Subject {  
 public List<Observer> observers = new ArrayList<>();  
  
 public void addObserver(Observer observer){  
 observers.add(observer);  
 }  
  
 public void removeObserver(Observer observer){  
 observers.remove(observer);  
 }  
  
 public void notifyObservers(int value){  
 for(var observer : observers){  
 observer.update(value);  
 }  
 }  
}

**Inside DataSrc.java** (an “observable”)

public class DataSource extends Subject {  
 private Integer value;  
  
 public Integer getValue() {  
 return value;  
 }  
  
 public void setValue(Integer value) {  
 this.value = value;  
 notifyObservers(value);  
 }  
}

**Inside Observer.java** (the interface off of which observers are based)

public interface Observer {  
 void update(int value);  
}

**Inside Spreadsheet.java** (an “observer”)

public class Spreadsheet implements Observer{  
 @Override  
 public void update(int value) {  
 System.*out*.println("Spreadsheet was just updated with value :" + value);  
 }  
}

**Inside Chart.java** (an “observer”)

public class Chart implements Observer{  
 @Override  
 public void update(int value) {  
 System.*out*.println("Chart was just updated with value :" + value);  
 }  
}

**Inside main.java**

var dataSrc = new DataSource();  
var sheet1 = new Spreadsheet();  
var sheet2 = new Spreadsheet();  
var chart = new Chart();  
  
dataSrc.addObserver(sheet1);  
dataSrc.addObserver(sheet2);  
dataSrc.addObserver(chart);  
  
dataSrc.setValue(1);

**THE MEDIATOR PATTERN**

**Inside DialogBox.java** (the abstract parent class of all mediators)

public abstract class DialogBox {  
 public abstract void changed(UIControl control);  
}

**Inside UIControl.java** (the abstract parent class of a number of UIControls that are grouped together to be monitored and operated upon by the same mediator)

public class UIControl {  
 protected DialogBox owner;  
  
 public UIControl(DialogBox owner){  
 this.owner = owner;  
 }  
}

**Inside ListBox.java** (an object/array/list that we want our UI Controls to communicate with)

public class ListBox extends UIControl{  
 private String selection;  
  
 public ListBox(DialogBox owner) { super(owner); }  
  
 public String getSelection() { return selection; }  
  
 public void setSelection(String selection) {  
 this.selection = selection;  
 owner.changed(this);  
 }  
}

**Inside TextBox.java** (a UIControl)

public class TextBox extends UIControl{  
 private String content;  
  
 public TextBox(DialogBox owner) {  
 super(owner);  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 owner.changed(this);  
 }  
}

**Inside Button.java** (a UIControl)

public class Button extends UIControl{  
 private Boolean isEnabled;  
  
 public Button(DialogBox owner) {  
 super(owner);  
 }  
  
 public Boolean getEnabled() {  
 return isEnabled;  
 }  
  
 public void setEnabled(Boolean enabled) {  
 isEnabled = enabled;  
 owner.changed(this);  
 }  
}

**Inside ArticleListBox.java** (a mediator)

public class ArticlesDialogBox extends DialogBox{  
 private ListBox articlesListBox = new ListBox(this);  
 private TextBox articlesTextBox = new TextBox(this);  
 private Button saveButton = new Button(this);  
  
 public void simulateUserInteraction(){  
 articlesListBox.setSelection("Article #1");  
// articlesTextBox.setContent("");  
// articlesListBox.setSelection("Article #2");  
 System.*out*.println("Text title : " + articlesTextBox.getContent());  
 System.*out*.println("Button Status : " + saveButton.getEnabled());  
 }  
  
 @Override  
 public void changed(UIControl control) {  
 if(control == articlesListBox){  
 articleListBoxSelected();  
 }else if(control == articlesTextBox){  
 articleTextBoxSelected();  
 }  
 }  
  
 public void articleListBoxSelected(){  
 articlesTextBox.setContent(articlesListBox.getSelection());  
 saveButton.setEnabled(true);  
 }  
  
 public void articleTextBoxSelected(){  
 var content = articlesTextBox.getContent();  
 var isEmpty = (content == null || content.isEmpty());  
 saveButton.setEnabled(!isEmpty);  
 }  
}

**Inside main.java**

var articleDialogBox = new ArticlesDialogBox();  
articleDialogBox.simulateUserInteraction();

**THE CHAIN OF RESPONSIBILITY PATTERN**

**Inside HttpRequest.java** (just your regular request object)

public class HttpRequest {  
 private String username;  
 private String password;  
  
 public HttpRequest(String username, String password) {  
 this.username = username;  
 this.password = password;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public String getUsername() {  
 return username;  
 }  
}

**Inside Handler.java** (Linked List that handles middleware chronology logic i.e. each handler calls the next one)

public abstract class Handler {  
 private Handler next;  
  
 public Handler(Handler next){  
 this.next = next;  
 }  
  
 public void handle(HttpRequest request){  
 if(doHandle(request)){  
 return;  
 }  
  
 if(next!=null){  
 next.handle(request);  
 }  
 }  
  
 public abstract boolean doHandle(HttpRequest request);  
}

**Inside WebServer.java** (Calls the handlers)

public class WebServer {  
 private Handler handler;  
  
 public WebServer(Handler handler) {  
 this.handler = handler;  
 }  
  
 public void handle(HttpRequest request){  
 handler.handle(request);  
 }  
}

**Inside Authenticator.java** (Handler #1)

public class Authenticator extends Handler{  
 public Authenticator(Handler next) {  
 super(next);  
 }  
  
 @Override  
 public boolean doHandle(HttpRequest request) {  
 System.*out*.println("Authenticating...");  
 if(request.getUsername() == "admin" && request.getPassword() == "admin1234"){  
 System.*out*.println("Authentication passed ✔");  
 return false;  
 }  
 else  
 System.*out*.println("Authentication failed ❌");  
 return true;  
 }  
}

**Inside Logger.java** (Handler #2)

public class Logger extends Handler{  
 public Logger(Handler next) {  
 super(next);  
 }  
  
 @Override  
 public boolean doHandle(HttpRequest request) {  
 System.*out*.println("Logging....");  
 return false;  
 }  
}

**Inside Compressor.java** (Handler #3)

public class Compressor extends Handler{  
 public Compressor(Handler next) {  
 super(next);  
 }  
  
 @Override  
 public boolean doHandle(HttpRequest request) {  
 System.*out*.println("Compressing....");  
 return false;  
 }  
}

**Inside main.java**

// authenticator -> compressor -> logger

var logger = new Logger(null);  
var compressor = new Compressor(logger);  
var authenticator = new Authenticator(compressor);  
var webServer = new WebServer(authenticator);  
  
var request = new HttpRequest("admin", "admin1234");  
webServer.handle(request);

**THE VISITOR PATTERN**

**Inside Operation.java** (the visitor interface)

public interface Operation {  
 void apply(HeadingNode headingNode);  
 void apply(AnchorTagNode anchorTagNode);  
}

**Inside HighlightOperation.java** (a visitor)

public class HighlightOperation implements Operation{  
 @Override  
 public void apply(HeadingNode headingNode) {  
 System.*out*.println("highlight-heading");  
 }  
  
 @Override  
 public void apply(AnchorTagNode anchorTagNode) {  
 System.*out*.println("highlight-anchor");  
 }  
}

**Inside PlainTextOperation.java** (a visitor)

public class PlainTextOperation implements Operation{  
 @Override  
 public void apply(HeadingNode headingNode) {  
 System.*out*.println("Plain text from heading node");  
 }  
  
 @Override  
 public void apply(AnchorTagNode anchorTagNode) {  
 System.*out*.println("Plain text from anchor node");  
 }  
}

**Inside HtmlNode.java** (the element interface)

public interface HtmlNode {  
 void execute(Operation operation);  
}

**Inside HeadingNode.java** (an element)

public class HeadingNode implements HtmlNode{  
 @Override  
 public void execute(Operation operation) {  
 operation.apply(this);  
 }  
}

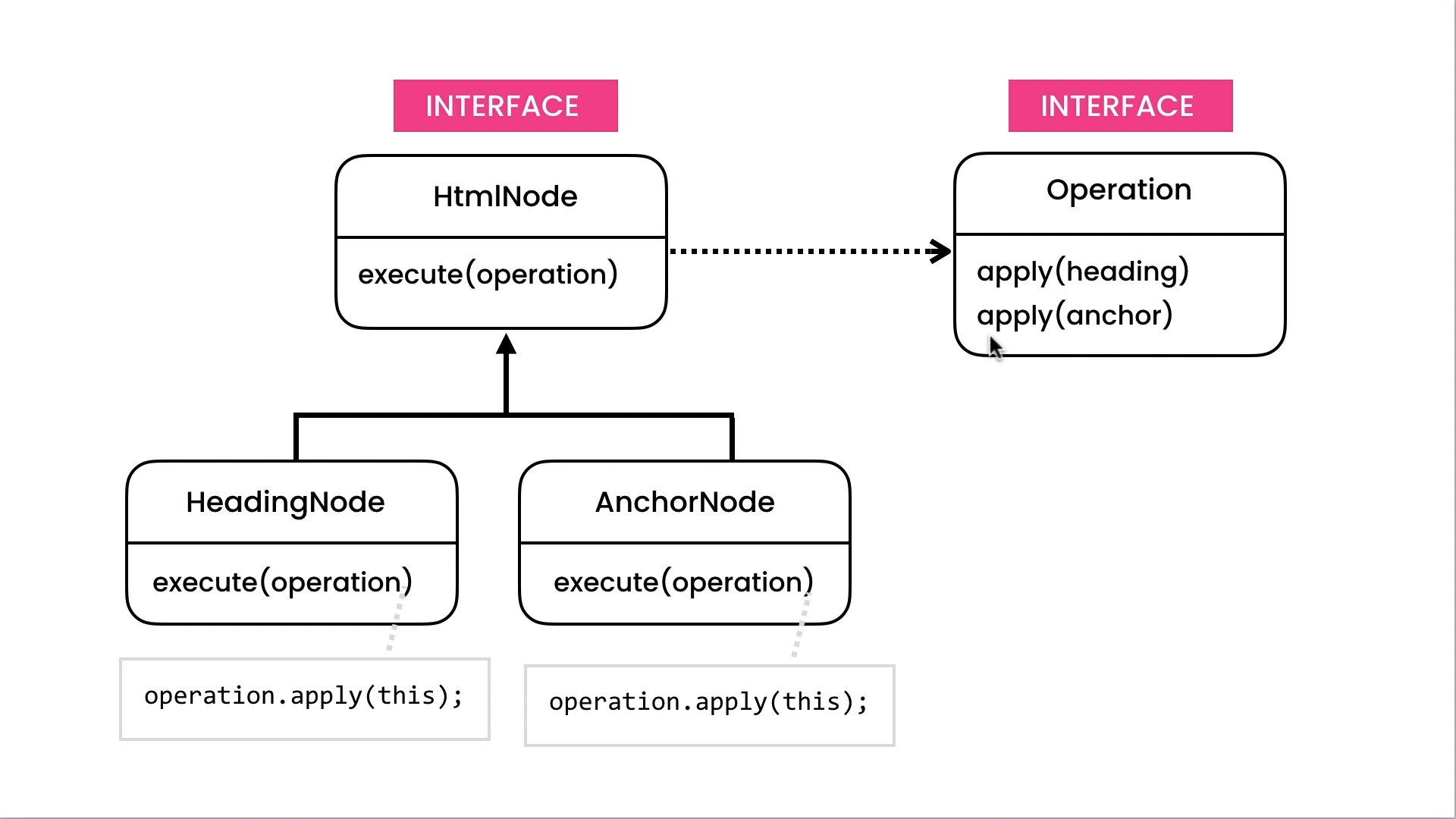
**Inside AnchorTagNode.java** (an element)

public class AnchorTagNode implements HtmlNode{  
 @Override  
 public void execute(Operation operation) {  
 operation.apply(this);  
 }  
}

**Inside HtmlDocument.java** (just an extra class to add new operations and call operations on all elements together)

**Inside Main.java**

var htmlDocument = new HtmlDocument();  
var headingNode= new HeadingNode();  
var anchorNode = new AnchorTagNode();  
htmlDocument.add(headingNode);  
htmlDocument.add(anchorNode);  
System.*out*.println("Extracting plain text from html tags....");  
htmlDocument.execute(new PlainTextOperation());  
System.*out*.println("Highlighting html tags.......");  
htmlDocument.execute(new HighlightOperation());



**THE COMPOSITE PATTERN**

**Inside Component.java**

public interface Component {  
 void render();  
 void move();  
}

**Inside Shape.java**

public class Shape implements Component{  
 @Override  
 public void render(){ System.*out*.println("Rendered !");}  
  
 @Override  
 public void move(){ System.*out*.println("Moved !"); }  
}

**Inside Group.java**

public class Group implements Component  
{  
 private List<Component> components = new ArrayList<>();  
  
 public void add(Component cmp){ components.add(cmp);}  
  
 @Override  
 public void render() {  
 for(var cmp : components){  
 cmp.render();  
 }  
 }  
  
 @Override  
 public void move() {  
 for(var cmp : components){  
 cmp.move();  
 }  
 }  
}

**Inside main.java**

var shape1 = new Shape();  
var shape2 = new Shape();  
var shape3 = new Shape();  
var shape4 = new Shape();  
var group1 = new Group();  
var group2 = new Group();  
var group = new Group();  
group1.add(shape1);  
group1.add(shape2);  
group2.add(shape3);  
group2.add(shape4);  
group.add(group1);  
group.add(group2);  
group.render();  
group.move();

**THE ADAPTER PATTERN**

**Inside Image.java** (unimportant class)

public class Image {  
}

**Inside Filter.java** (the interface we want all filters to follow)

public interface Filter {  
 void apply(Image image);  
}

**Inside ImageView.java** (just a consumer of any class following the Filter interface)

public interface Filter {  
 void apply(Image image);  
}

**Inside VividFilter.java** (class that, by definition follows Filter interface)

public class VividFilter implements Filter{  
 @Override  
 public void apply(Image image){  
 System.*out*.println("Applied custom made vivid filter");  
 }  
}

**Inside Caramel Filter.java** ( **adaptee** i.e. class that, by definition, does not follow the Filter interface so we gotta apply an adapter to this to use it in any consumers of the Filter interface like ImageView)

public class VividFilter implements Filter{  
 @Override  
 public void apply(Image image){  
 System.*out*.println("Applied custom made vivid filter");  
 }  
}

**Inside CaramelFilter.java**

public class CaramelFilter {  
 public void init(){  
 System.*out*.println("Initialising 3rd party package....");  
 }  
  
 public void render(){  
 System.*out*.println("Caramel filter rendered on image");  
 }  
}

**Inside CaramelAdapterOne** (**adapter**, implemented via the Composition principle)

public class CaramelFilter {  
 public void init(){  
 System.*out*.println("Initialising 3rd party package....");  
 }  
  
 public void render(){  
 System.*out*.println("Caramel filter rendered on image");  
 }  
}

**Inside CaramelAdapterTwo** (**adapter,** implemented via the Inheritance principle)

public class CaramelAdapterTwo extends CaramelFilter implements Filter {  
 @Override  
 public void apply(Image image) {  
 init();  
 System.*out*.println("Using inheritance to create adapter");  
 render();  
 }  
}

**Inside main.java**

var image = new Image();  
var imageView = new ImageView(image);  
imageView.apply(new VividFilter());  
imageView.apply(new CaramelAdapterOne(new CaramelFilter()));  
imageView.apply(new CaramelAdapterTwo());