**Easy codes: 10,9, 8, 6, 4B, 3, 2A, 2B, 1.  
Easy way to remember hard patterns : page 29(only if you know what has to be added in the code)**

**Practical 1:Factory method**Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

interface Shape {

void draw();

}

class Circle implements Shape {

@Override

public void draw() {

System.out.println("Draw a circle");

}

}

class Square implements Shape {

@Override

public void draw() {

System.out.println("Draw a square");

}

}

class ShapeFactory {

public Shape getShape(String shapeType) {

if (shapeType == null) {

return null;

}

if (shapeType.equalsIgnoreCase("Circle")) {

return new Circle();

} else if (shapeType.equalsIgnoreCase("Square")) {

return new Square();

}

return null;

}

}

class FactoryPatternDemo {

public static void main(String args[]) {

ShapeFactory shapeFactory = new ShapeFactory();

Shape shape1 = shapeFactory.getShape("Circle");

shape1.draw();

Shape shape2 = shapeFactory.getShape("Square");

shape2.draw();

}

}

**Practical 2A:Builder Method**   
  
public class Coffee {

private String type;

private boolean sugar;

private boolean milk;

private String size;

private Coffee(Builder builder) {

this.type = builder.type;

this.sugar = builder.sugar;

this.milk = builder.milk;

this.size = builder.size;

}

public static class Builder {

private String type;

private boolean sugar;

private boolean milk;

private String size;

public Builder(String type) {

this.type = type;

}

public Builder sugar(boolean sugar) {

this.sugar = sugar;

return this;

}

public Builder milk(boolean milk) {

this.milk = milk;

return this;

}

public Builder size(String size) {

this.size = size;

return this;

}

public Coffee build() {

return new Coffee(this);

}

}

@Override

public String toString() {

return String.format("Coffee[type=%s, sugar=%s, milk=%s, size=%s]", type, sugar, milk, size);

}

public static void main(String[] args) {

Coffee coffee1 = new Coffee.Builder("Country Bean").sugar(false).milk(true).size("big").build();

Coffee coffee2 = new Coffee.Builder("Black Coffee").build();

System.out.println(coffee1);

System.out.println(coffee2);

}

}

**Practical 2B:Singlton method**  
  
1.) SingleObject.java

class SingleObject {

private static SingleObject instance = new SingleObject();

private SingleObject() {}

public static SingleObject getInstance() {

return instance;

}

public void showMessage() {

System.out.println("Hello World");

}

}

class SingletonPatternDemo {

public static void main(String args[]) {

SingleObject object = SingleObject.getInstance();

object.showMessage();

}

}

**Practical 3:Prototype**

import java.util.\*;

interface Prototype {

Prototype getClone();

}

class EmployeeRecord implements Prototype {

private int id;

private String name, designation;

private double salary;

private String address;

public EmployeeRecord(int id, String name, String designation, double salary, String address) {

this.id = id;

this.name = name;

this.designation = designation;

this.salary = salary;

this.address = address;

}

public void showRecord() {

System.out.println(id + "\t" + name + "\t" + designation + "\t" + salary + "\t" + address);

}

@Override

public Prototype getClone() {

return new EmployeeRecord(id, name, designation, salary, address);

}

}

class PrototypeDemo {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter employee id: ");

int eid = scanner.nextInt();

System.out.print("Enter employee name: ");

String ename = scanner.next();

System.out.print("Enter employee designation: ");

String edesignation = scanner.next();

System.out.print("Enter employee address: ");

String eaddress = scanner.next();

System.out.print("Enter employee salary: ");

double esalary = scanner.nextDouble();

EmployeeRecord e1 = new EmployeeRecord(eid, ename, edesignation, esalary, eaddress);

e1.showRecord();

EmployeeRecord e2 = (EmployeeRecord) e1.getClone();

e2.showRecord();

scanner.close();

}

}

**Practical 4A:Adapter**

import java.util.\*;

interface CreditCard {

void giveBankDetails();

String getCreditCard();

}

class BankDetails {

private String bankName;

private String accHolderName;

private long accNumber;

public String getBankName() {

return bankName;

}

public void setBankName(String bankName) {

this.bankName = bankName;

}

public String getAccHolderName() {

return accHolderName;

}

public void setAccHolderName(String accHolderName) {

this.accHolderName = accHolderName;

}

public long getAccNumber() {

return accNumber;

}

public void setAccNumber(long accNumber) {

this.accNumber = accNumber;

}

}

class BankCustomer extends BankDetails implements CreditCard {

@Override

public void giveBankDetails() {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the account holder name: ");

String customerName = scanner.nextLine();

System.out.print("Enter the account number: ");

long accNumber = scanner.nextLong();

System.out.print("Enter the Bank name: ");

scanner.nextLine();

String bankName = scanner.nextLine();

setAccHolderName(customerName);

setAccNumber(accNumber);

setBankName(bankName);

scanner.close();

}

@Override

public String getCreditCard() {

long accNumber = getAccNumber();

String accHolderName = getAccHolderName();

String bankName = getBankName();

return ("The Account number " + accNumber + " of " + accHolderName + " in " + bankName

+ " bank is valid and authenticated for issuing the credit card.");

}

}

class AdapterPatternDemo {

public static void main(String[] args) {

CreditCard targetInterface = new BankCustomer();

targetInterface.giveBankDetails();

System.out.print(targetInterface.getCreditCard());

}

}

**Practical 4B:Bridge**// Vehicle abstract class

abstract class Vehicle {

protected Workshop workShop1;

protected Workshop workShop2;

protected Vehicle(Workshop workShop1, Workshop workShop2) {

this.workShop1 = workShop1;

this.workShop2 = workShop2;

}

abstract public void manufacture();

}

// Car class

class Car extends Vehicle {

public Car(Workshop workShop1, Workshop workShop2) {

super(workShop1, workShop2);

}

@Override

public void manufacture() {

System.out.print("car ");

workShop1.work();

workShop2.work();

}

}

// Bike class

class Bike extends Vehicle {

public Bike(Workshop workShop1, Workshop workShop2) {

super(workShop1, workShop2);

}

@Override

public void manufacture() {

System.out.print("bike ");

workShop1.work();

workShop2.work();

}

}

// Workshop interface

interface Workshop {

void work();

}

// Assemble class

class Assemble implements Workshop {

@Override

public void work() {

System.out.println("and Assembled");

}

}

// Produce class

class Produce implements Workshop {

@Override

public void work() {

System.out.println("produced");

}

}

// BridgePattern class

class BridgePattern {

public static void main(String[] args) {

Vehicle vehicle1 = new Car(new Produce(), new Assemble());

vehicle1.manufacture();

Vehicle vehicle2 = new Bike(new Produce(), new Assemble());

vehicle2.manufacture();

}

}

**Practical 5A:Composite**

import java.util.ArrayList;

import java.util.List;

interface Employee {

int getId();

String getName();

double getSalary();

void print();

void add(Employee employee);

void remove(Employee employee);

Employee getChild(int i);

}

class Cashier implements Employee {

private int id;

private String name;

private double salary;

public Cashier(int id, String name, double salary) {

this.id = id;

this.name = name;

this.salary = salary;

}

@Override

public void add(Employee employee) {

// Cashier does not have subordinates

throw new UnsupportedOperationException("Cannot add employee to a cashier");

}

@Override

public void remove(Employee employee) {

// Cashier does not have subordinates

throw new UnsupportedOperationException("Cannot remove employee from a cashier");

}

@Override

public Employee getChild(int i) {

// Cashier does not have subordinates

return null;

}

@Override

public int getId() {

return id;

}

@Override

public String getName() {

return name;

}

@Override

public double getSalary() {

return salary;

}

@Override

public void print() {

System.out.println("Employee Details:");

System.out.println("ID: " + getId());

System.out.println("Name: " + getName());

System.out.println("Salary: $" + getSalary());

System.out.println("Position: Cashier");

System.out.println("------------------------------");

}

}

class Accountant implements Employee {

private int id;

private String name;

private double salary;

public Accountant(int id, String name, double salary) {

this.id = id;

this.name = name;

this.salary = salary;

}

@Override

public void add(Employee employee) {

// Accountant does not have subordinates

throw new UnsupportedOperationException("Cannot add employee to an accountant");

}

@Override

public void remove(Employee employee) {

// Accountant does not have subordinates

throw new UnsupportedOperationException("Cannot remove employee from an accountant");

}

@Override

public Employee getChild(int i) {

// Accountant does not have subordinates

return null;

}

@Override

public int getId() {

return id;

}

@Override

public String getName() {

return name;

}

@Override

public double getSalary() {

return salary;

}

@Override

public void print() {

System.out.println("Employee Details:");

System.out.println("ID: " + getId());

System.out.println("Name: " + getName());

System.out.println("Salary: $" + getSalary());

System.out.println("Position: Accountant");

System.out.println("------------------------------");

}

}

class BankManager implements Employee {

private int id;

private String name;

private double salary;

private List<Employee> employees;

public BankManager(int id, String name, double salary, List<Employee> employees) {

this.id = id;

this.name = name;

this.salary = salary;

this.employees = employees;

}

@Override

public void add(Employee employee) {

employees.add(employee);

}

@Override

public void remove(Employee employee) {

employees.remove(employee);

}

@Override

public Employee getChild(int i) {

return employees.get(i);

}

@Override

public int getId() {

return id;

}

@Override

public String getName() {

return name;

}

@Override

public double getSalary() {

return salary;

}

@Override

public void print() {

System.out.println("Employee Details:");

System.out.println("ID: " + getId());

System.out.println("Name: " + getName());

System.out.println("Salary: $" + getSalary());

System.out.println("Position: Bank Manager");

System.out.println("------------------------------");

for (Employee employee : employees) {

employee.print();

}

}

}

public class CompositePatternDemo {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

Employee emp1 = new Cashier(101, "Sohan Kumar", 20000.0);

Employee emp2 = new Cashier(102, "Mohan Kumar", 25000.0);

Employee emp3 = new Accountant(103, "Seema Mahiwal", 30000.0);

employees.add(emp1);

employees.add(emp2);

employees.add(emp3);

Employee manager1 = new BankManager(100, "Ashwani Rajput", 100000.0, employees);

manager1.print();

}

}

**Practical 5B:Decorator**

// Shape.java (Can be a separate file)

public interface Shape {

void draw();

}

// Rectangle.java (Can be a separate file)

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Shape: Rectangle");

}

}

// Circle.java (Can be a separate file)

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Shape: Circle");

}

}

// ShapeDecorator.java (Can be a separate file)

public abstract class ShapeDecorator implements Shape {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape) {

this.decoratedShape = decoratedShape;

}

public void draw() {

decoratedShape.draw();

}

}

// RedShapeDecorator.java (Can be a separate file)

public class RedShapeDecorator extends ShapeDecorator {

public RedShapeDecorator(Shape decoratedShape) {

super(decoratedShape);

}

@Override

public void draw() {

decoratedShape.draw();

setRedBorder();

}

private void setRedBorder() {

System.out.println("Border Color: Red");

}

}

// DecoratorPatternDemo.java (Main class, keep in the same file)

public class DecoratorPatternDemo {

public static void main(String[] args) {

Shape circle = new Circle();

Shape redCircle = new RedShapeDecorator(new Circle());

Shape redRectangle = new RedShapeDecorator(new Rectangle());

System.out.println("Circle with normal border:");

circle.draw();

System.out.println("\nCircle of red border:");

redCircle.draw();

System.out.println("\nRectangle of red border:");

redRectangle.draw();

}

}

**Practical 6:Chain of responsibility**

import java.util.Scanner;

class Currency {

private int amount;

public Currency(int amt) {

this.amount = amt;

}

public int getAmount() {

return this.amount;

}

}

interface DispenseChain {

void setNextChain(DispenseChain nextChain);

void dispense(Currency cur);

}

class Dollar50Dispenser implements DispenseChain {

private DispenseChain chain;

@Override

public void setNextChain(DispenseChain nextChain) {

this.chain = nextChain;

}

@Override

public void dispense(Currency cur) {

if (cur.getAmount() >= 50) {

int num = cur.getAmount() / 50;

int remainder = cur.getAmount() % 50;

System.out.println("Dispensing " + num + " 50$ note");

if (remainder != 0)

this.chain.dispense(new Currency(remainder));

} else {

this.chain.dispense(cur);

}

}

}

class Dollar20Dispenser implements DispenseChain {

private DispenseChain chain;

@Override

public void setNextChain(DispenseChain nextChain) {

this.chain = nextChain;

}

@Override

public void dispense(Currency cur) {

if (cur.getAmount() >= 20) {

int num = cur.getAmount() / 20;

int remainder = cur.getAmount() % 20;

System.out.println("Dispensing " + num + " 20$ note");

if (remainder != 0)

this.chain.dispense(new Currency(remainder));

} else {

this.chain.dispense(cur);

}

}

}

class Dollar10Dispenser implements DispenseChain {

private DispenseChain chain;

@Override

public void setNextChain(DispenseChain nextChain) {

this.chain = nextChain;

}

@Override

public void dispense(Currency cur) {

if (cur.getAmount() >= 10) {

int num = cur.getAmount() / 10;

int remainder = cur.getAmount() % 10;

System.out.println("Dispensing " + num + " 10$ note");

if (remainder != 0)

this.chain.dispense(new Currency(remainder));

} else {

this.chain.dispense(cur);

}

}

}

class ATMDispenseChain {

private DispenseChain c1;

public ATMDispenseChain() {

this.c1 = new Dollar50Dispenser();

DispenseChain c2 = new Dollar20Dispenser();

DispenseChain c3 = new Dollar10Dispenser();

c1.setNextChain(c2);

c2.setNextChain(c3);

}

public static void main(String[] args) {

ATMDispenseChain atmDispenser = new ATMDispenseChain();

while (true) {

int amount = 0;

System.out.println("Enter amount to dispense");

Scanner input = new Scanner(System.in);

amount = input.nextInt();

if (amount % 10 != 0) {

System.out.println("Amount should be in multiple of 10s.");

return;

}

atmDispenser.c1.dispense(new Currency(amount));

}

}}

**Practical 7:Iterator**

interface Iterator {

boolean hasNext();

Object next();

}

interface Container {

Iterator getIterator();

}

class CollectionOfNames implements Container {

private String[] names = {"Aaaaaa", "Bbbbbb", "Cccc"};

@Override

public Iterator getIterator() {

return new CollectionOfNamesIterator();

}

private class CollectionOfNamesIterator implements Iterator {

private int index;

@Override

public boolean hasNext() {

return index < names.length;

}

@Override

public Object next() {

if (this.hasNext()) {

return names[index++];

}

return null;

}

}

}

class IteratorPatternDemo {

public static void main(String[] args) {

CollectionOfNames companyRepository = new CollectionOfNames();

for (Iterator iterator = companyRepository.getIterator(); iterator.hasNext();) {

String name = (String) iterator.next();

System.out.println("Name: " + name);

}

}}

**Practical 8:Templatee**

abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

final void play() {

initialize();

startPlay();

endPlay();

}

}

class Cricket extends Game {

@Override

void endPlay() { System.out.println("Cricket Game Finished!"); }

@Override

void initialize() { System.out.println("Cricket Game Initialized! Start playing."); }

@Override

void startPlay() { System.out.println("Cricket Game started. Enjoy the game!"); }

}

class Football extends Game {

@Override

void endPlay() { System.out.println("Football Game Finished!"); }

@Override

void initialize() { System.out.println("Football Game Initialized! Start playing."); }

@Override

void startPlay() { System.out.println("Football Game started. Enjoy the game!"); }

}

class TemplatePatternDemo {

public static void main(String[] args) {

Game cricket = new Cricket();

cricket.play();

System.out.println();

Game football = new Football();

football.play();

}

}

**Practical 9:Memento**

class MementoPatternDemo {

public static void main(String[] args) {

Originator org = new Originator();

Caretaker care = new Caretaker();

org.setState("State #1");

care.add(org.saveStateToMemento());

org.setState("State #2");

care.add(org.saveStateToMemento());

org.setState("State #3");

care.add(org.saveStateToMemento());

org.setState("State #4");

care.add(org.saveStateToMemento());

System.out.println("Current State: " + org.getState());

org.getStateFromMemento(care.get(0));

System.out.println("First saved state: " + org.getState());

org.getStateFromMemento(care.get(1));

System.out.println("Second saved State: " + org.getState());

org.getStateFromMemento(care.get(2));

System.out.println("Third saved State: " + org.getState());

}

}

class Originator {

private String state;

public void setState(String state) {

this.state = state;

}

public String getState() {

return state;

}

public Memento saveStateToMemento() {

return new Memento(state);

}

public void getStateFromMemento(Memento memento) {

state = memento.getState();

}

}

class Memento {

private String state;

public Memento(String state) {

this.state = state;

}

public String getState() {

return state;

}

}

class Caretaker {

private List<Memento> mementoList = new ArrayList<>();

public void add(Memento state) {

mementoList.add(state);

}

public Memento get(int index) {

return mementoList.get(index);

}

}

**Practical 10: Facade Design pattern**

The Facade pattern simplifies interactions with complex systems by providing a unified interface that hides the intricacies of the subsystem. It acts as a gateway, allowing clients to access the functionality they need without needing to understand the inner workings of the system. This abstraction enhances maintainability, scalability, and reusability by decoupling clients from the details of the subsystem implementation.  
import *java.util.\**;  
// Represents a mobile shop interface with methods to display model number and price.

interface MobileShop {

void modelNo();

void price();

}

// Represents an iPhone mobile model.

class Iphone implements MobileShop {

public void modelNo() {

System.out.println("Iphone 6");

}

public void price() {

System.out.println("Rs 65000.00");

}

}

// Represents a Samsung mobile model.

class Samsung implements MobileShop {

public void modelNo() {

System.out.println("Samsung Galaxy Tab 3");

}

public void price() {

System.out.println("Rs 45000.00");

}

}

// Represents a Blackberry mobile model.

class Blackberry implements MobileShop {

public void modelNo() {

System.out.println("Blackberry Z10");

}

public void price() {

System.out.println("Rs 55000.00");

}

}

// Represents a shopkeeper who manages the mobile shop.

class ShopKeeper {

private MobileShop iphone = new Iphone();

private MobileShop samsung = new Samsung();

private MobileShop blackberry = new Blackberry();

public void iphoneSale() {

iphone.modelNo();

iphone.price();

}

public void samsungSale() {

samsung.modelNo();

samsung.price();

}

public void blackberrySale() {

blackberry.modelNo();

blackberry.price();

}

}

// Represents a client class that interacts with the mobile shop through a facade pattern.

public class FacadePatternClient {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

ShopKeeper shopKeeper = new ShopKeeper();

int choice;

do {

System.out.println("========== Mobile Shop ==========");

System.out.println("1. IPHONE");

System.out.println("2. SAMSUNG");

System.out.println("3. BLACKBERRY");

System.out.println("4. Exit");

System.out.println("Enter your choice: ");

choice = scanner.nextInt();

switch (choice) {

case 1:

shopKeeper.iphoneSale();

break;

case 2:

shopKeeper.samsungSale();

break;

case 3:

shopKeeper.blackberrySale();

break;

case 4:

System.out.println("Exiting the shop. Goodbye!");

break;

default:

System.out.println("Invalid choice. Try again.");

}

} while (choice != 4);

}

}

**Easy way to remember hard patterns are as Follows:**

Prac 10 : Facade

import java.util.\*;

interface MobileShop {

void modelNo();

void price();

}

// Represents an iPhone mobile model.

//Make same class and change the name of phone and price

class Iphone implements MobileShop {

public void modelNo() {

System.out.println("Iphone 6");

}

public void price() {

System.out.println("Rs 65000.00");

}

}

// Represents a shopkeeper who manages the mobile shop.

class ShopKeeper {

//make same function and variable and change name of phones

private MobileShop iphone = new Iphone();

public void iphoneSale() {

iphone.modelNo();

iphone.price();

}

}

public class FacadePatternClient {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

ShopKeeper shopKeeper = new ShopKeeper();

int choice;

do {

System.out.println("1. IPHONE");

System.out.println("2. Exit");

System.out.println("Enter your choice: ");

choice = sc.nextInt();

switch (choice) {

case 1:

shopKeeper.iphoneSale();

break;

case 2:

System.out.println("Exiting the shop. Goodbye!");

break;

default:

System.out.println("Invalid choice. Try again.");

}

} while (choice != 2);

}

}

Prac 9 : Memento

class MementoPatternDemo {

public static void main(String[] args) {

Originator org = new Originator();

Caretaker care = new Caretaker();

org.setState("State #1");

care.add(org.saveStateToMemento());

org.setState("State #2");

care.add(org.saveStateToMemento());

System.out.println("Current State: " + org.getState());

org.getStateFromMemento(care.get(0));

System.out.println("First saved state: " + org.getState());

org.getStateFromMemento(care.get(1));

System.out.println("Second saved State: " + org.getState());

}

}

class Originator {

private String state;

public void setState(String state) {

this.state = state;

}

public String getState() {

return state;

}

public Memento saveStateToMemento() {

return new Memento(state);

}

public void getStateFromMemento(Memento memento) {

state = memento.getState();

}

}

class Memento {

private String state;

public Memento(String state) {

this.state = state;

}

public String getState() {

return state;

}

}

class Caretaker {

private List<Memento> mementoList = new ArrayList<>();

public void add(Memento state) {

mementoList.add(state);

}

public Memento get(int index) {

return mementoList.get(index);

}

}

Prac 2 : Builder

public class Coffee{

private String type;

private boolean milk;

private Coffee(Builder builder){

this.type=builder.type;

this.milk=builder.milk;

}

public static class Builder{

private String type;

private boolean milk;

public Builder(String type){

this.type=type;

}

public Builder milk(boolean milk){

this.milk=milk;

return this;

}

public Coffee build(){

return new Coffee(this);

}

}

@Override

public String toString(){

return String.format("Coffee[type=%s,milk=%s]",type,milk);

}

public static void main(String args[]){

Coffee coffee1=new Coffee.Builder("ghar ka chai").milk(true).build();

System.out.println(coffee1);

}

}