|  |  |
| --- | --- |
| abtrasfactory  public abstract class Computer {  public abstract String getRAM();  public abstract String getHDD();  public abstract String getCPU();  @Override  public String toString(){  return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+",  CPU="+this.getCPU();  }  }  public class PC extends Computer {© JOURNALDEV.COM PAGE 21 OF 132  private String ram;  private String hdd;  private String cpu;  public PC(String ram, String hdd, String cpu){  this.ram=ram;  this.hdd=hdd;  this.cpu=cpu;  }  @Override  public String getRAM() {  return this.ram;  }  @Override  public String getHDD() {  return this.hdd;  }  @Override  public String getCPU() {  return this.cpu;  }  }  package com.journaldev.design.model;  public class Server extends Computer {  private String ram;  private String hdd;  private String cpu;  public Server(String ram, String hdd, String cpu){  this.ram=ram;  this.hdd=hdd;  this.cpu=cpu;  }  @Override  public String getRAM() {  return this.ram;  }© JOURNALDEV.COM PAGE 22 OF 132  @Override  public String getHDD() {  return this.hdd;  }  @Override  public String getCPU() {  return this.cpu;  }  }  public interface ComputerAbstractFactory {  public Computer createComputer();  }  public class PCFactory implements ComputerAbstractFactory {  private String ram;  private String hdd;  private String cpu;  public PCFactory(String ram, String hdd, String cpu){  this.ram=ram;  this.hdd=hdd;  this.cpu=cpu;  }  @Override  public Computer createComputer() {  return new PC(ram,hdd,cpu);  }  }  public class ServerFactory implements ComputerAbstractFactory {  private String ram;  private String hdd;  private String cpu;  public ServerFactory(String ram, String hdd, String cpu){  this.ram=ram;  this.hdd=hdd;  this.cpu=cpu;  }  @Override  public Computer createComputer() {  return new Server(ram,hdd,cpu);  }  }  public class ComputerFactory {  public static Computer getComputer(ComputerAbstractFactory factory){  return factory.createComputer();  }  }  public class TestDesignPatterns {  public static void main(String[] args) {  testAbstractFactory();  }  private static void testAbstractFactory() {  Computer pc =  com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new  PCFactory("2 GB","500 GB","2.4 GHz"));  Computer server =  com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new  ServerFactory("16 GB","1 TB","2.9 GHz"));  System.out.println("AbstractFactory PC Config::"+pc);  System.out.println("AbstractFactory Server Config::"+server);  }  }  --Decorator  public interface Car {  public void assemble();  }  public class BasicCar implements Car {  @Override  public void assemble() {  System.out.print("Basic Car.");  }  }  public class CarDecorator implements Car {  protected Car car;  public CarDecorator(Car c){  this.car=c;  }  @Override  public void assemble() {  this.car.assemble();  }  }  public class SportsCar extends CarDecorator {  public SportsCar(Car c) {  super(c);  }  @Override  public void assemble(){  car.assemble();  System.out.print(" Adding features of Sports Car.");  }  }  public class LuxuryCar extends CarDecorator {  public LuxuryCar(Car c) {  super(c);  }  @Override  public void assemble(){  car.assemble();  System.out.print(" Adding features of Luxury Car.");  }  }  public class DecoratorPatternTest {  public static void main(String[] args) {  Car sportsCar = new SportsCar(new BasicCar());  sportsCar.assemble();  System.out.println("\n\*\*\*\*\*");  Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new  BasicCar()));  sportsLuxuryCar.assemble();  }  public class SingleObject {  //create an object of SingleObject  private static SingleObject instance = new SingleObject();  //make the constructor private so that this class cannot be  //instantiated  private SingleObject(){}  //Get the only object available  public static SingleObject getInstance(){  return instance;  }  public void showMessage(){  System.out.println("Hello World!"); | -Builder pattern  public interface Item {  public String name();  public Packing packing();  public float price();  }  public interface Packing {  public String pack();  }  public class Wrapper implements Packing {  @Override  public String pack() {  return "Wrapper";  }  }  public class Bottle implements Packing {  @Override  public String pack() {  return "Bottle";  }  }  public abstract class Burger implements Item {  @Override  public Packing packing() {  return new Wrapper();  }  @Override  public abstract float price();  }  public abstract class ColdDrink implements Item {  @Override  public Packing packing() {  return new Bottle();  }  @Override  public abstract float price();  }  public class VegBurger extends Burger {  @Override  public float price() {  return 25.0f;  }  @Override  public String name() {  return "Veg Burger";  }  }  public class ChickenBurger extends Burger {  @Override  public float price() {  return 50.5f;  }  @Override  public String name() {  return "Chicken Burger";  }  }  public class Coke extends ColdDrink {  @Override  public float price() {  return 30.0f;  }  @Override  public String name() {  return "Coke";  }  }  public class Pepsi extends ColdDrink {  @Override  public float price() {  return 35.0f;  }  @Override  public String name() {  return "Pepsi";  }  }  public class Meal {  private List<Item> items = new ArrayList<Item>();  public void addItem(Item item){  items.add(item);  }  public float getCost(){  float cost = 0.0f;    for (Item item : items) {  cost += item.price();  }  return cost;  }  public void showItems(){    for (Item item : items) {  System.out.print("Item : " + item.name());  System.out.print(", Packing : " + item.packing().pack());  System.out.println(", Price : " + item.price());  }  }  }  public class MealBuilder {  public Meal prepareVegMeal (){  Meal meal = new Meal();  meal.addItem(new VegBurger());  meal.addItem(new Coke());  return meal;  }  public Meal prepareNonVegMeal (){  Meal meal = new Meal();  meal.addItem(new ChickenBurger());  meal.addItem(new Pepsi());  return meal;  }  }  public class BuilderPatternDemo {  public static void main(String[] args) {    MealBuilder mealBuilder = new MealBuilder();  Meal vegMeal = mealBuilder.prepareVegMeal();  System.out.println("Veg Meal");  vegMeal.showItems();  System.out.println("Total Cost: " + vegMeal.getCost());  Meal nonVegMeal = mealBuilder.prepareNonVegMeal();  System.out.println("\n\nNon-Veg Meal");  nonVegMeal.showItems();  System.out.println("Total Cost: " + nonVegMeal.getCost());  }  } |