Design Pattern in Java

A design pattern is a **well-proved solution** for solving the specific problem/task.

Now, a question will be arising in your mind what kind of specific problem? Let me explain by taking an example.

**Problem Given:**  
Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

**Solution:**  
**Singleton design pattern** is the best solution of above specific problem. So, every design pattern has **some specification or set of rules** for solving the problems. What are those specifications, you will see later in the types of design patterns.

But remember one-thing, design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.

To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

Advantage of design pattern:

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They capture the software engineering experiences.
4. They provide transparency to the design of an application.
5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6. Design patterns don?t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

When should we use the design patterns?

We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

**Categorization of design patterns:**

Basically, design patterns are categorized into two parts:

1. Core java (or JSE) Design Patterns.
2. JEE Design Patterns.

Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

1. Creational Design Pattern
   1. Factory Pattern
   2. Abstract Factory Pattern
   3. Singleton Pattern
   4. Prototype Pattern
   5. Builder Pattern.
2. Structural Design Pattern
3. Adapter Pattern
4. Bridge Pattern
5. Composite Pattern
6. Decorator Pattern
7. Facade Pattern
8. Flyweight Pattern
9. Proxy Pattern
10. Behavioral Design Pattern
11. Chain Of Responsibility Pattern
12. Command Pattern
13. Interpreter Pattern
14. Iterator Pattern
15. Mediator Pattern
16. Memento Pattern
17. Observer Pattern
18. State Pattern
19. Strategy Pattern
20. Template Pattern
21. Visitor Pattern

Do you know?

* **Christopher Alexander** was the first person who invented all the above Design Patterns in 1977.
* But later the **Gang of Four - Design patterns, elements of reusable object-oriented software** book was written by a group of four persons named as Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in 1995.
* That's why all the above 23 Design Patterns are known as **Gang of Four (GoF) Design Patterns.**

Creational design patterns

Creational design patterns are concerned with**the way of creating objects.** These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

But everyone knows an object is created by using new keyword in java. For example:

StudentRecord s1=**new** StudentRecord();

Hard-Coded code is not the good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

Types of creational design patterns

There are following 6 types of creational design patterns.

1. [Factory Method Pattern](http://www.javatpoint.com/factory-method-design-pattern)
2. [Abstract Factory Pattern](http://www.javatpoint.com/abstract-factory-pattern)
3. [Singleton Pattern](http://www.javatpoint.com/singleton-design-pattern-in-java)
4. [Prototype Pattern](http://www.javatpoint.com/prototype-design-pattern)
5. [Builder Pattern](http://www.javatpoint.com/builder-design-pattern)
6. [Object Pool Pattern](http://www.javatpoint.com/object-pool-pattern)

Factory Method Pattern

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

**Advantage of Factory Design Pattern**

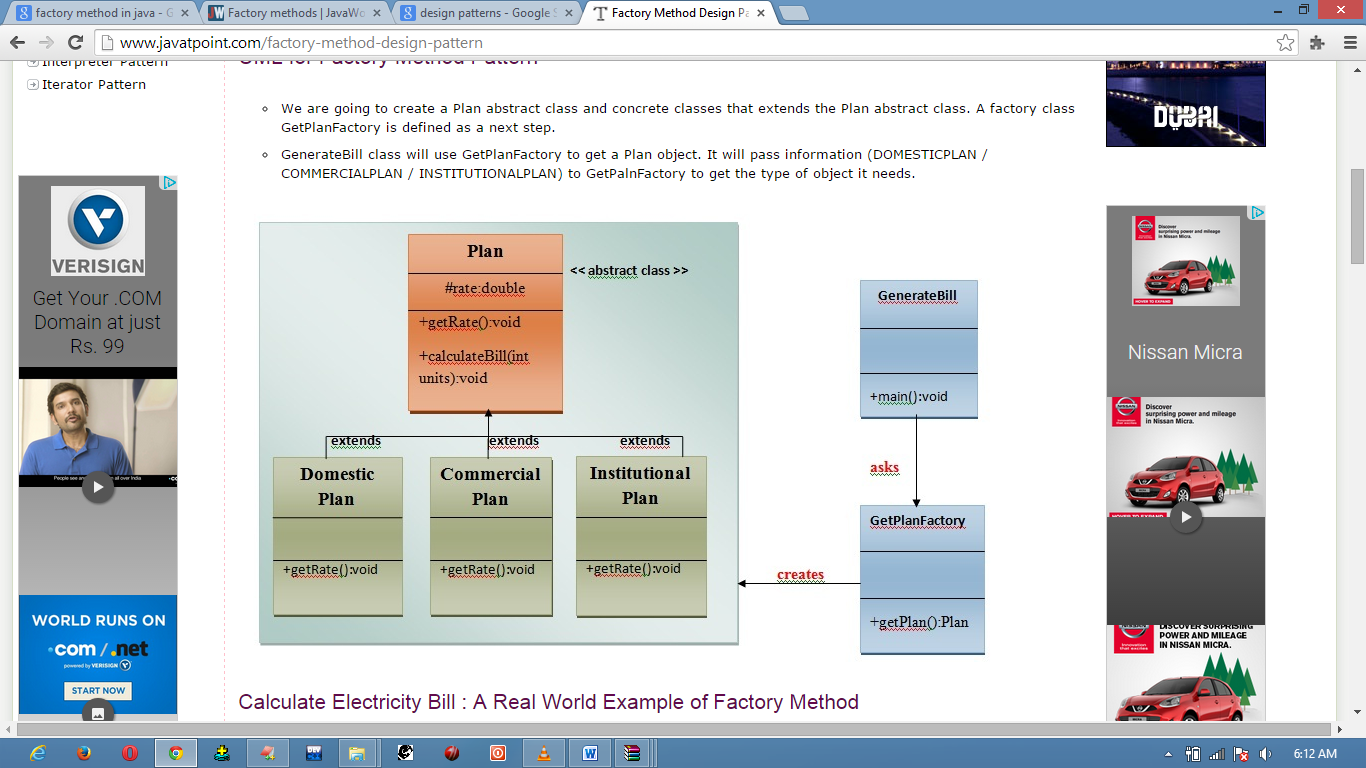
* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

**Usage of Factory Design Pattern**

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

UML for Factory Method Pattern

* We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.
* GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.



//Calculate Electricity bill a real world Example.

// Step1

//We are going to create a Plan abstract class.

import java.io.\*; /\*we are using io concept in our class that's why we are importing io pacakage.\*/

abstract class Plan

{

protected double rate;

abstract void getRate();

public void calculateBill(int units)

{

System.out.println(units\*rate);

}

}//end of Plan class.

// Step 2

//We are going to create a Concrete classes that extends Plan abstract class.

class DomesticPlan extends Plan

{

//@override

public void getRate()

{

rate=3.50;

}

}//end of DomesticPlan class.

class CommercialPlan extends Plan

{

//@override

public void getRate()

{

rate=7.50;

}

}//end of CommercialPlan class.

class InstitutionalPlan extends Plan

{

//@override

public void getRate()

{

rate=5.50;

}

}//end of InstitutionalPlan class.

// Step 3

//Create a GetPlanFactory to generate object of concrete classes based on given information.

class GetPlanFactory

{

//use getPlan method to get object of type Plan

public Plan getPlan(String planType)

{

if(planType == null)

{

return null;

}

if(planType.equalsIgnoreCase("DOMESTICPLAN"))

{

return new DomesticPlan();

}

else if(planType.equalsIgnoreCase("COMMERCIALPLAN"))

{

return new CommercialPlan();

}

else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN"))

{

return new InstitutionalPlan();

}

return null;

}

}//end of GetPlanFactory class.

// Step 4

//Use the GetPlanFactory to get the object of concrete classes by passing an information such as type(DOMESTICPLAN/COMMERCIALPLAN/INSTITUTIONALPLAN).

class GenerateBill

{

public static void main(String args[])throws IOException

{

GetPlanFactory planFactory = new GetPlanFactory();

//get an object of DomesticPaln and call its getPlan()method.But we want to calculate the bill for one plan at time not all.for this we IO concept.

System.out.print("Enter the name of plan for which the bill will be generated: ");

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

String planName=br.readLine();

System.out.print("Enter the number of units for bill will be calculated: ");

int units=Integer.parseInt(br.readLine());

Plan p = planFactory.getPlan(planName);

//call getRate() method and calculateBill()method of DomesticPaln.

System.out.print("Bill amount for "+planName+" of "+units+" units is: ");

p.getRate();

p.calculateBill(units);

}

}//end of GenerateBill class.

Abstract Factory Pattern

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

**Advantage of Abstract Factory Pattern**

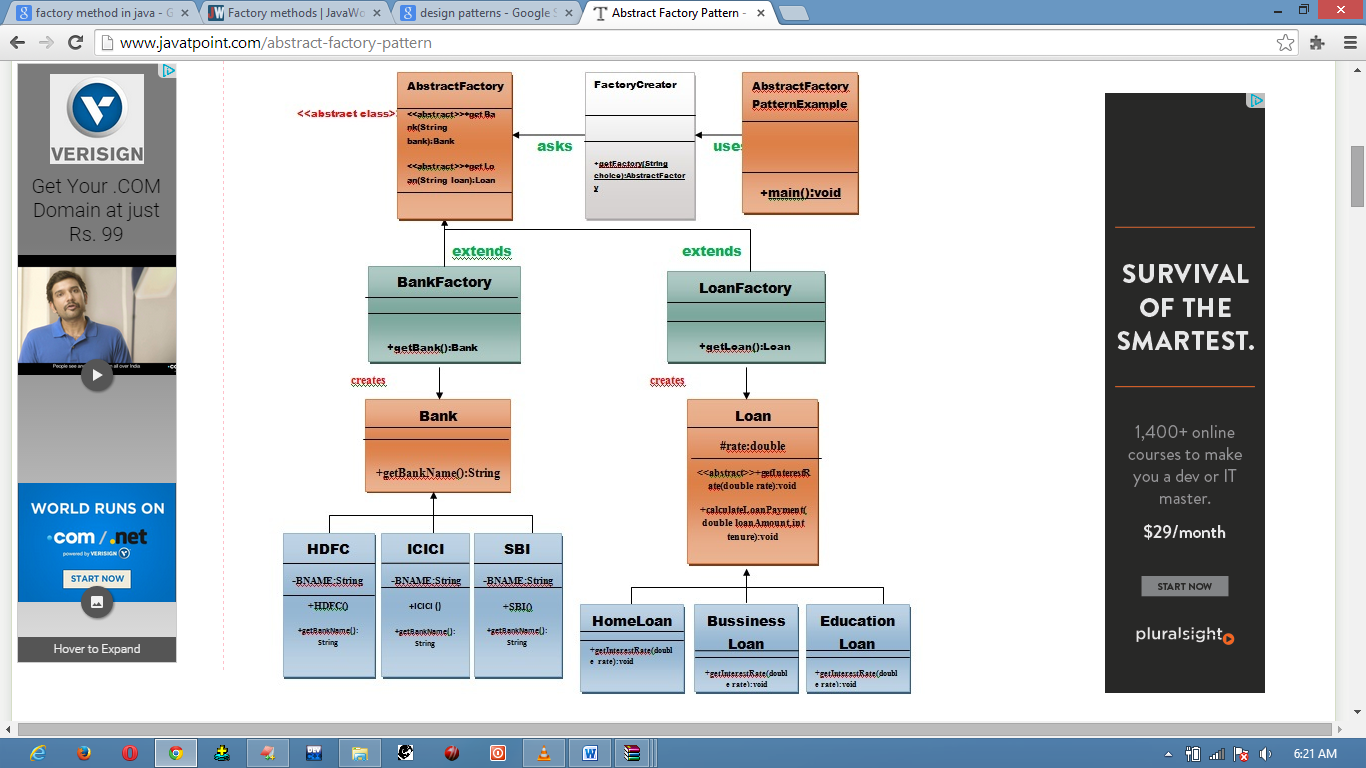
* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.

**Usage of Abstract Factory Pattern**

* When the system needs to be independent of how its object are created, composed, and represented.
* When the family of related objects has to be used together, then this constraint needs to be enforced.
* When you want to provide a library of objects that does not show implementations and only reveals interfaces.
* When the system needs to be configured with one of a multiple family of objects.

UML for Abstract Factory Pattern

* We are going to create a **Bank interface** and a **Loan abstract class** as well as their sub-classes.
* Then we will create **AbstractFactory** class as next step.
* Then after we will create concrete classes, **BankFactory,** and **LoanFactory** that will extends **AbstractFactory class**
* After that, **AbstractFactoryPatternExample** class uses the **FactoryCreator** to get an object of **AbstractFactory** class.
* See the diagram carefully which is given below:



// calculate the loan payment for different banks like HDFC, ICICI, SBI etc. a real world example.

// Step 1

//We are going to create a Bank interface.

import java.io.\*; /\*we are using io concept in our class that's why we are importing io pacakage.\*/

interface Bank

{

String getBankName();

}

// Step 2

//Create concrete classes that will implement the Bank interface.

class HDFC implements Bank

{

private final String BNAME;

public HDFC()

{

BNAME="HDFC BANK";

}

// @override

public String getBankName()

{

return BNAME;

}

}

class ICICI implements Bank

{

private final String BNAME;

public ICICI()

{

BNAME="ICICI BANK";

}

// @override

public String getBankName()

{

return BNAME;

}

}

class SBI implements Bank

{

private final String BNAME;

public SBI()

{

BNAME="SBI BANK";

}

// @override

public String getBankName()

{

return BNAME;

}

}

// Step 3

//Create a loan abstract class.

abstract class Loan

{

protected double rate;

abstract void getInterestRate(double rate);

public void calculateLoanPayment(double loanamount, int years)

{

/\*

to calculate the monthly loan payment i.e. EMI

rate=annual interest rate/12\*100;

n=number of monthly installments;

1year=12 months.

so, n=years\*12;

\*/

double EMI;

int n;

n=years\*12;

rate=rate/1200;

EMI=((rate\*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))\*loanamount;

System.out.println("your's monthly EMI is "+ EMI +" for the amount "+loanamount+" you have borrowed " );

}

}// end of the Loan abstract class.

// Step 4

//Create concrete classes that will extends the Loan abstract class.

class HomeLoan extends Loan

{

public void getInterestRate(double r)

{

rate=r;

}

}//End of the HomeLoan class.

class BussinessLoan extends Loan

{

public void getInterestRate(double r)

{

rate=r;

}

}//End of the BusssinessLoan class.

class EducationLoan extends Loan

{

public void getInterestRate(double r)

{

rate=r;

}

}//End of the EducationLoan class.

// Step 5

//Create an abstract class (i.e AbstractFactory) to get the factories for Bank and Loan Objects.

abstract class AbstractFactory

{

public abstract Bank getBank(String bank);

public abstract Loan getLoan(String loan);

}//End of the AbstractFactory.

// Step 6

//Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

class BankFactory extends AbstractFactory

{

//@override

public Bank getBank(String bank){

if(bank == null){

return null;

}

if(bank.equalsIgnoreCase("HDFC")){

return new HDFC();

} else if(bank.equalsIgnoreCase("ICICI")){

return new ICICI();

} else if(bank.equalsIgnoreCase("SBI")){

return new SBI();

}

return null;

}

//@Override

public Loan getLoan(String loan) {

return null;

}

}//End of the BankFactory class.

class LoanFactory extends AbstractFactory

{

//@Override

public Bank getBank(String bank)

{

return null;

}

//@Override

public Loan getLoan(String loan){

if(loan == null){

return null;

}

if(loan.equalsIgnoreCase("Home")){

return new HomeLoan();

} else if(loan.equalsIgnoreCase("Business")){

return new BussinessLoan();

} else if(loan.equalsIgnoreCase("Education")){

return new EducationLoan();

}

return null;

}

}//End of the LoanFactory class.

// Step 7

//Create a FactoryCreator class to get the factories by passing an information such as Bank or Loan.

class FactoryCreator {

public static AbstractFactory getFactory(String choice){

if(choice.equalsIgnoreCase("Bank")){

return new BankFactory();

} else if(choice.equalsIgnoreCase("Loan")){

return new LoanFactory();

}

return null;

}

}//End of the FactoryCreator.

// Step 8

//Use the FactoryCreator to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

class AbstractFactoryPatternExample

{

public static void main(String args[])throws IOException

{

//get Bank factory

//get an object of Bank and call its method getBank().But we want to get one bank at time not all.for this we IO concept.

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

System.out.print("Enter the name of Bank from where you want to take loan amount: ");

String bankName=br.readLine();

System.out.print("\n");

System.out.print("Enter the type of loan you want to take, like home loan or bussiness loan or education loan : ");

String loanName=br.readLine();

AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");

Bank b=bankFactory.getBank(bankName);

System.out.print("\n");

System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

double rate=Double.parseDouble(br.readLine());

System.out.print("\n");

System.out.print("Enter the loan amount you want to take: ");

double loanAmount=Double.parseDouble(br.readLine());

System.out.print("\n");

System.out.print("Enter the number of years to pay your entire loan amount: ");

int years=Integer.parseInt(br.readLine());

System.out.print("\n");

System.out.println("you are taking the loan from "+ b.getBankName());

AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");

Loan l=loanFactory.getLoan(loanName);

l.getInterestRate(rate);

l.calculateLoanPayment(loanAmount,years);

}

}//End of the AbstractFactoryPatternExample

Prototype Design Pattern

Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

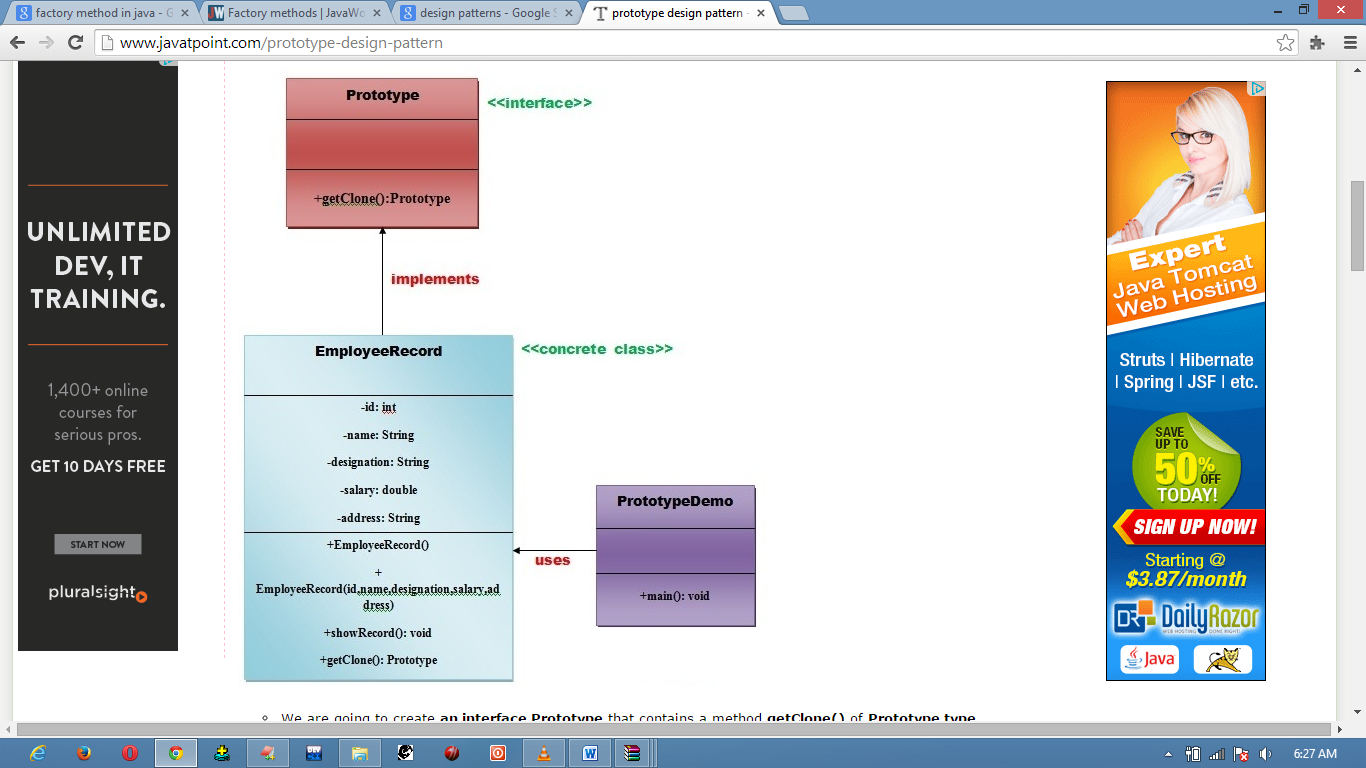
**Advantage of Prototype Pattern**

The main advantages of prototype pattern are as follows:

* It reduces the need of sub-classing.
* It hides complexities of creating objects.
* The clients can get new objects without knowing which type of object it will be.
* It lets you add or remove objects at runtime.

**Usage of Prototype Pattern**

* When the classes are instantiated at runtime.
* When the cost of creating an object is expensive or complicated.
* When you want to keep the number of classes in an application minimum.
* When the client application needs to be unaware of object creation and representation.



// Employee records of oracle corporation.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

interface Prototype {

public Prototype getClone();

}//End of Prototype interface.

class EmployeeRecord implements Prototype{

private int id;

private String name, designation;

private double salary;

private String address;

public EmployeeRecord(){

System.out.println(" Employee Records of Oracle Corporation ");

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

System.out.println("Eid"+"\t"+"Ename"+"\t"+"Edesignation"+"\t"+"Esalary"+"\t\t"+"Eaddress");

}

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

this();

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);

}

}//End of EmployeeRecord class.

class PrototypeDemo{

public static void main(String[] args) throws IOException {

BufferedReader br =new BufferedReader(new InputStreamReader(System.in));

System.out.print("Enter Employee Id: ");

int eid=Integer.parseInt(br.readLine());

System.out.print("\n");

System.out.print("Enter Employee Name: ");

String ename=br.readLine();

System.out.print("\n");

System.out.print("Enter Employee Designation: ");

String edesignation=br.readLine();

System.out.print("\n");

System.out.print("Enter Employee Address: ");

String eaddress=br.readLine();

System.out.print("\n");

System.out.print("Enter Employee Salary: ");

double esalary= Double.parseDouble(br.readLine());

System.out.print("\n");

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

e1.showRecord();

System.out.println("\n");

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

e2.showRecord();

}

}//End of the ProtoypeDemo class.

Builder Design Pattern

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

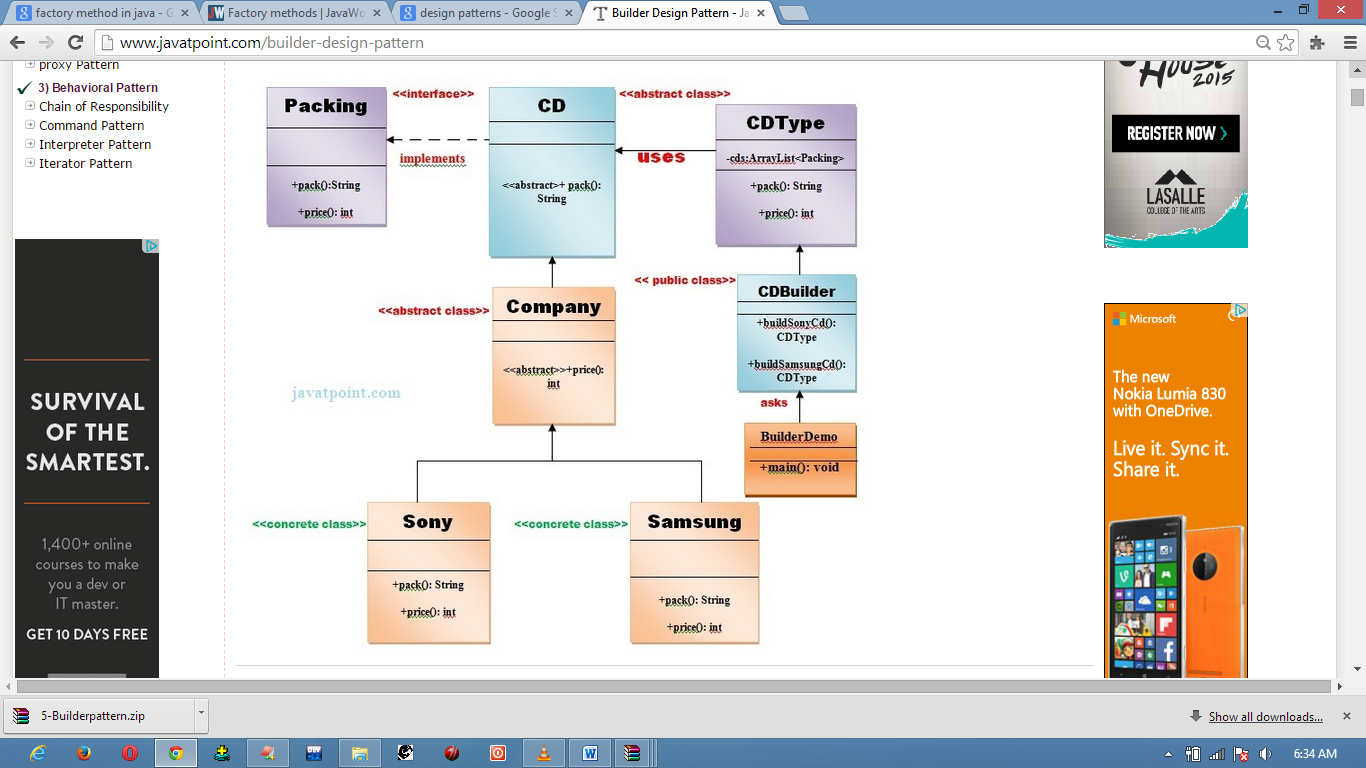
It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

**Advantage of Builder Design Pattern**

The main advantages of Builder Pattern are as follows:

* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects.

**UML for Builder Pattern Example**



Example of Builder Design Pattern

To create simple example of builder design pattern, you need to follow 6 following steps.

1. Create Packing interface
2. Create 2 abstract classes CD and Company
3. Create 2 implementation classes of Company: Sony and Samsung
4. Create the CDType class
5. Create the CDBuilder class
6. Create the BuilderDemo class

public interface Packing {

public String pack();

public int price();

}// End of Packing interface.

public abstract class CD implements Packing{

public abstract String pack();

}//End of the CD class.

public abstract class Company extends CD{

public abstract int price();

}//End of the Company class.

public class Sony extends Company{

@Override

public int price(){

return 20;

}

@Override

public String pack(){

return "Sony CD";

}

}//End of the Sony class.

public class Samsung extends Company {

@Override

public int price(){

return 15;

}

@Override

public String pack(){

return "Samsung CD";

}

}//End of the Samsung class.

import java.util.ArrayList;

import java.util.List;

public class CDType

{

private List<Packing> items=new ArrayList<Packing>();

public void addItem(Packing packs)

{

items.add(packs);

}

public void getCost()

{

for (Packing packs : items)

{

packs.price();

}

}

public void showItems()

{

for (Packing packing : items)

{

System.out.print("CD name : "+packing.pack());

System.out.println(", Price : "+packing.price());

}

}

}//End of the CDType class.

public class CDBuilder

{

public CDType buildSonyCD()

{

CDType cds=new CDType();

cds.addItem(new Sony());

return cds;

}

public CDType buildSamsungCD()

{

CDType cds=new CDType();

cds.addItem(new Samsung());

return cds;

}

}// End of the CDBuilder class.

public class BuilderDemo{

public static void main(String args[]){

CDBuilder cdBuilder=new CDBuilder();

CDType cdType1=cdBuilder.buildSonyCD();

cdType1.showItems();

CDType cdType2=cdBuilder.buildSamsungCD();

cdType2.showItems();

}

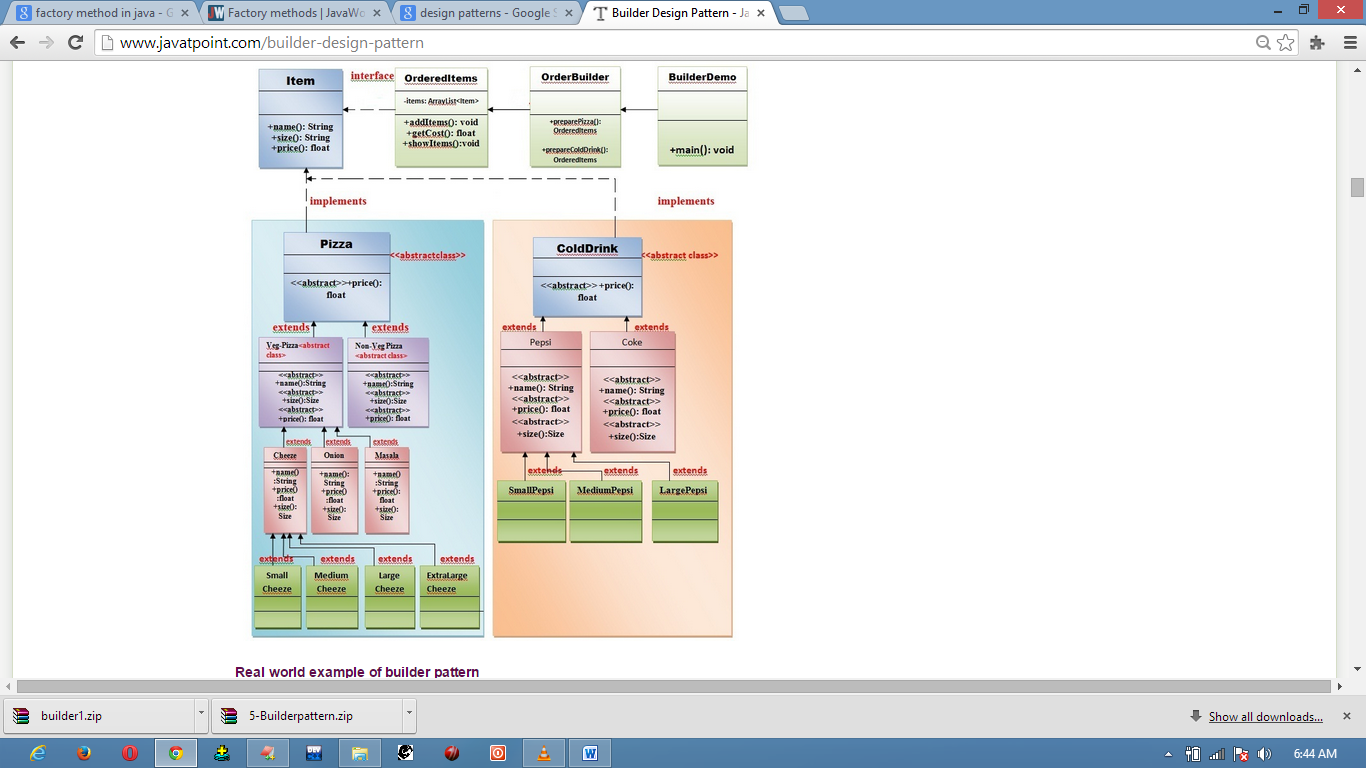
}

### Another Real world example of Builder Pattern

#### UML for Builder Pattern:

We are considering a business case of **pizza-hut** where we can get different varieties of pizza and cold-drink.

**Pizza** can be either a Veg pizza or Non-Veg pizza of several types (like cheese pizza, onion pizza, masala-pizza etc) and will be of 4 sizes i.e. small, medium, large, extra-large.

**Cold-drink** can be of several types (like Pepsi, Coke, Dew, Sprite, Fanta, Maaza, Limca, Thums-up etc.) and will be of 3 sizes small, medium, large.

**//Create an interface Item that represents the Pizza and Cold-drink.**

public interface Item

{

public String name();

public String size();

public float price();

}// End of the interface Item.

**//Create an abstract class Pizza that will implement to the interface Item.**

public abstract class Pizza implements Item

{

@Override

public abstract float price();

}

**//Create an abstract class ColdDrink that will implement to the interface Item.**

public abstract class ColdDrink implements Item

{

@Override

public abstract float price();

}

**//Create an abstract class VegPizza that will extend to the abstract class Pizza.**

public abstract class VegPizza extends Pizza

{

@Override

public abstract float price();

@Override

public abstract String name();

@Override

public abstract String size();

}// End of the abstract class VegPizza.

**//Create an abstract class NonVegPizza that will extend to the abstract class //Pizza.**

public abstract class NonVegPizza extends Pizza

{

@Override

public abstract float price();

@Override

public abstract String name();

@Override

public abstract String size();

}// End of the abstract class NonVegPizza.

**//Now, create concrete sub-classes SmallCheezePizza, MediumCheezePizza, //LargeCheezePizza, ExtraLargeCheezePizza that will extend to the abstract class //VegPizza.**

public class SmallCheezePizza extends VegPizza

{

@Override

public float price()

{

return 170.0f;

}

@Override

public String name()

{

return "Cheeze Pizza";

}

@Override

public String size()

{

return "Small size";

}

}// End of the SmallCheezePizza class.

public class MediumCheezePizza extends VegPizza

{

@Override

public float price()

{

return 220.f;

}

@Override

public String name()

{

return "Cheeze Pizza";

}

@Override

public String size()

{

return "Medium Size";

}

}// End of the MediumCheezePizza class.

public class LargeCheezePizza extends VegPizza

{

@Override

public float price()

{

return 260.0f;

}

@Override

public String name()

{

return "Cheeze Pizza";

}

@Override

public String size()

{

return "Large Size";

}

}// End of the LargeCheezePizza class.

public class ExtraLargeCheezePizza extends VegPizza

{

@Override

public float price()

{

return 300.f;

}

@Override

public String name()

{

return "Cheeze Pizza";

}

@Override

public String size()

{

return "Extra-Large Size";

}

}// End of the ExtraLargeCheezePizza class.

**//Now, similarly create concrete sub-classes SmallOnionPizza, //MediumOnionPizza, LargeOnionPizza, ExtraLargeOnionPizza that will extend to //the abstract class VegPizza.**

public class SmallOnionPizza extends VegPizza

{

@Override

public float price()

{

return 120.0f;

}

@Override

public String name()

{

return "Onion Pizza";

}

@Override

public String size()

{

return "Small Size";

}

}// End of the SmallOnionPizza class.

public class MediumOnionPizza extends VegPizza

{

@Override

public float price()

{

return 150.0f;

}

@Override

public String name()

{

return "Onion Pizza";

}

@Override

public String size()

{

return "Medium Size";

}

}// End of the MediumOnionPizza class.

public class LargeOnionPizza extends VegPizza

{

@Override

public float price()

{

return 180.0f;

}

@Override

public String name()

{

return "Onion Pizza";

}

@Override

public String size()

{

return "Large size";

}

}// End of the LargeOnionPizza class.

public class ExtraLargeOnionPizza extends VegPizza

{

@Override

public float price()

{

return 200.0f;

}

@Override

public String name()

{

return "Onion Pizza";

}

@Override

public String size()

{

return "Extra-Large Size";

}

}// End of the ExtraLargeOnionPizza class

**//Now, similarly create concrete sub-classes SmallMasalaPizza, //MediumMasalaPizza, LargeMasalaPizza, ExtraLargeMasalaPizza that will extend //to the abstract class VegPizza.**

public class SmallMasalaPizza extends VegPizza

{

@Override

public float price()

{

return 100.0f;

}

@Override

public String name()

{

return "Masala Pizza";

}

@Override

public String size()

{

return "Samll Size";

}

}// End of the SmallMasalaPizza class

public class MediumMasalaPizza extends VegPizza

{

@Override

public float price()

{

return 120.0f;

}

@Override

public String name()

{

return "Masala Pizza";

}

@Override

public String size()

{

return "Medium Size";

}

public class LargeMasalaPizza extends VegPizza

{

@Override

public float price()

{

return 150.0f;

}

@Override

public String name()

{

return "Masala Pizza";

}

@Override

public String size()

{

return "Large Size";

}

} //End of the LargeMasalaPizza class

public class ExtraLargeMasalaPizza extends VegPizza

{

@Override

public float price()

{

return 180.0f;

}

@Override

public String name()

{

return "Masala Pizza";

}

@Override

public String size()

{

return "Extra-Large Size";

}

}// End of the ExtraLargeMasalaPizza class

**//Now, create concrete sub-classes SmallNonVegPizza, MediumNonVegPizza, //LargeNonVegPizza, ExtraLargeNonVegPizza that will extend to the abstract //class NonVegPizza.**

public class SmallNonVegPizza extends NonVegPizza

{

@Override

public float price()

{

return 180.0f;

}

@Override

public String name()

{

return "Non-Veg Pizza";

}

@Override

public String size()

{

return "Samll Size";

}

}// End of the SmallNonVegPizza class

public class MediumNonVegPizza extends NonVegPizza

{

@Override

public float price()

{

return 200.0f;

}

@Override

public String name()

{

return "Non-Veg Pizza";

}

@Override

public String size()

{

return "Medium Size";

}

public class LargeNonVegPizza extends NonVegPizza

{

@Override

public float price()

{

return 220.0f;

}

@Override

public String name()

{

return "Non-Veg Pizza";

}

@Override

public String size()

{

return "Large Size";

}

}// End of the LargeNonVegPizza class

public class ExtraLargeNonVegPizza extends NonVegPizza

{

@Override

public float price()

{

return 250.0f;

}

@Override

public String name()

{

return "Non-Veg Pizza";

}

@Override

public String size()

{

return "Extra-Large Size";

}

} // End of the ExtraLargeNonVegPizza class

**//Now, create two abstract classes Pepsi and Coke that will extend abstract class //ColdDrink.**

public abstract class Pepsi extends ColdDrink

{

@Override

public abstract String name();

@Override

public abstract String size();

@Override

public abstract float price();

}// End of the Pepsi class

public abstract class Coke extends ColdDrink

{

@Override

public abstract String name();

@Override

public abstract String size();

@Override

public abstract float price();

}// End of the Coke class

public class SmallPepsi extends Pepsi

{

@Override

public String name()

{

return "300 ml Pepsi";

}

@Override

public float price()

{

return 25.0f;

}

@Override

public String size()

{

return "Small Size";

}

}// End of the SmallPepsi class

public class MediumPepsi extends Pepsi

{

@Override

public String name()

{

return "500 ml Pepsi";

}

@Override

public String size()

{

return "Medium Size";

}

@Override

public float price()

{

return 35.0f;

}

}// End of the MediumPepsi class

public class LargePepsi extends Pepsi

{

@Override

public String name()

{

return "750 ml Pepsi";

}

@Override

public String size()

{

return "Large Size";

}

@Override

public float price()

{

return 50.0f;

}

}// End of the LargePepsi class

**//Now, create concrete sub-classes SmallCoke, MediumCoke, LargeCoke that will //extend to the abstract class Coke.**

public class SmallCoke extends Coke

{

@Override

public String name()

{

return "300 ml Coke";

}

@Override

public String size()

{

return "Small Size";

}

@Override

public float price()

{

return 25.0f;

}

}// End of the SmallCoke class

public class MediumCoke extends Coke

{

@Override

public String name()

{

return "500 ml Coke";

}

@Override

public String size()

{

return "Medium Size";

}

@Override

public float price()

{

return 35.0f;

}

}// End of the MediumCoke class

public class LargeCoke extends Coke

{

@Override

public String name()

{

return "750 ml Coke";

}

@Override

public String size()

{

return "Large Size";

}

@Override

public float price()

{

return 50.0f;

}

}// End of the LargeCoke class

import java.util.ArrayList;

import java.util.List;

public class OrderedItems

{

List<Item> items=new ArrayList<Item>();

public void addItems(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.println("Item is:" +item.name());

System.out.println("Size is:" +item.size());

System.out.println("Price is: " +item.price());

}

}

}// End of the OrderedItems class

**//Create an OrderBuilder class that will be responsible to create the objects of //OrderedItems class.**

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class OrderBuilder

{

public OrderedItems preparePizza() throws IOException

{

OrderedItems itemsOrder=new OrderedItems();

BufferedReader br =new BufferedReader(new InputStreamReader(System.in));

System.out.println(" Enter the choice of Pizza ");

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

System.out.println(" 1. Veg-Pizza ");

System.out.println(" 2. Non-Veg Pizza ");

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

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

int pizzaandcolddrinkchoice=Integer.parseInt(br.readLine());

switch(pizzaandcolddrinkchoice)

{

case 1:

{

System.out.println("You ordered Veg Pizza");

System.out.println("\n\n");

System.out.println(" Enter the types of Veg-Pizza ");

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

System.out.println(" 1.Cheeze Pizza ");

System.out.println(" 2.Onion Pizza ");

System.out.println(" 3.Masala Pizza ");

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

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

int vegpizzachoice=Integer.parseInt(br.readLine());

switch(vegpizzachoice)

{

case 1:

{

System.out.println("You ordered Cheeze Pizza");

System.out.println("Enter the cheeze pizza size");

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

System.out.println(" 1. Small Cheeze Pizza ");

System.out.println(" 2. Medium Cheeze Pizza ");

System.out.println(" 3. Large Cheeze Pizza ");

System.out.println(" 4. Extra-Large Cheeze Pizza ");

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

int cheezepizzasize=Integer.parseInt(br.readLine());

switch(cheezepizzasize)

{

case 1:

itemsOrder.addItems(new SmallCheezePizza());

break;

case 2:

itemsOrder.addItems(new MediumCheezePizza());

break;

case 3:

itemsOrder.addItems(new LargeCheezePizza());

break;

case 4:

itemsOrder.addItems(new ExtraLargeCheezePizza());

break;

case 2:

{

System.out.println("You ordered Onion Pizza");

System.out.println("Enter the Onion pizza size");

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

System.out.println(" 1. Small Onion Pizza ");

System.out.println(" 2. Medium Onion Pizza ");

System.out.println(" 3. Large Onion Pizza ");

System.out.println(" 4. Extra-Large Onion Pizza ");

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

int onionpizzasize=Integer.parseInt(br.readLine());

switch(onionpizzasize)

{

case 1:

itemsOrder.addItems(new SmallOnionPizza());

break;

case 2:

itemsOrder.addItems(new MediumOnionPizza());

break;

case 3:

itemsOrder.addItems(new LargeOnionPizza());

break;

case 4:

itemsOrder.addItems(new ExtraLargeOnionPizza());

break;

}

}

break;

case 3:

{

System.out.println("You ordered Masala Pizza");

System.out.println("Enter the Masala pizza size");

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

System.out.println(" 1. Small Masala Pizza ");

System.out.println(" 2. Medium Masala Pizza ");

System.out.println(" 3. Large Masala Pizza ");

System.out.println(" 4. Extra-Large Masala Pizza ");

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

int masalapizzasize=Integer.parseInt(br.readLine());

switch(masalapizzasize)

{

case 1:

itemsOrder.addItems(new SmallMasalaPizza());

break;

case 2:

itemsOrder.addItems(new MediumMasalaPizza());

break;

case 3:

itemsOrder.addItems(new LargeMasalaPizza());

break;

case 4:

itemsOrder.addItems(new ExtraLargeMasalaPizza());

break;

}

}

break;

}

}

break;// Veg- pizza choice completed.

case 2:

{

System.out.println("You ordered Non-Veg Pizza");

System.out.println("\n\n");

System.out.println("Enter the Non-Veg pizza size");

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

System.out.println(" 1. Small Non-Veg Pizza ");

System.out.println(" 2. Medium Non-Veg Pizza ");

System.out.println(" 3. Large Non-Veg Pizza ");

System.out.println(" 4. Extra-Large Non-Veg Pizza ");

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

int nonvegpizzasize=Integer.parseInt(br.readLine());

switch(nonvegpizzasize)

{

case 1:

itemsOrder.addItems(new SmallNonVegPizza());

break;

case 2:

itemsOrder.addItems(new MediumNonVegPizza());

break;

case 3:

itemsOrder.addItems(new LargeNonVegPizza());

break;

case 4:

itemsOrder.addItems(new ExtraLargeNonVegPizza());

break;

}

}

break;

default:

{

break;

}

}//end of main Switch

//continued?..

System.out.println(" Enter the choice of ColdDrink ");

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

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

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

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

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

int coldDrink=Integer.parseInt(br.readLine());

switch (coldDrink)

{

case 1:

{

System.out.println("You ordered Pepsi ");

System.out.println("Enter the Pepsi Size ");

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

System.out.println(" 1. Small Pepsi ");

System.out.println(" 2. Medium Pepsi ");

System.out.println(" 3. Large Pepsi ");

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

int pepsisize=Integer.parseInt(br.readLine());

switch(pepsisize)

{

case 1:

itemsOrder.addItems(new SmallPepsi());

break;

case 2:

itemsOrder.addItems(new MediumPepsi());

break;

case 3:

itemsOrder.addItems(new LargePepsi());

break;

}

}

break;

case 2:

{

System.out.println("You ordered Coke");

System.out.println("Enter the Coke Size");

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

System.out.println(" 1. Small Coke ");

System.out.println(" 2. Medium Coke ");

System.out.println(" 3. Large Coke ");

System.out.println(" 4. Extra-Large Coke ");

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

int cokesize=Integer.parseInt(br.readLine());

switch(cokesize)

{

case 1:

itemsOrder.addItems(new SmallCoke());

break;

case 2:

itemsOrder.addItems(new MediumCoke());

break;

case 3:

itemsOrder.addItems(new LargeCoke());

break;

}

}

break;

default:

{

break;

}

}//End of the Cold-Drink switch

return itemsOrder;

} //End of the preparePizza() method

**//Create a BuilderDemo class that will use the OrderBuilder class.**

import java.io.IOException;

public class BuilderDemo

{

public static void main(String[] args) throws IOException

{

// TODO code application logic here

OrderBuilder builder=new OrderBuilder();

OrderedItems orderedItems=builder.preparePizza();

orderedItems.showItems();

System.out.println("\n");

System.out.println("Total Cost : "+ orderedItems.getCost());

}

}// End of the BuilderDemo class