**Singleton pattern** is one of the simplest 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.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

*SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



Step 1

Create a Singleton Class.

*SingleObject.java*

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!");

}

}

Step 2

Get the only object from the singleton class.

*SingletonPatternDemo.java*

public class SingletonPatternDemo {

public static void main(String[] args) {

//illegal construct

//Compile Time Error: The constructor SingleObject() is not visible

//SingleObject object = new SingleObject();

//Get the only object available

SingleObject object = SingleObject.getInstance();

//show the message

object.showMessage();

}

}

**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.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

*FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



Step 1

Create an interface.

*Shape.java*

public interface Shape {

void draw();

}

Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

*Square.java*

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

Step 3

Create a Factory to generate object of concrete class based on given information.

*ShapeFactory.java*

public class ShapeFactory {

//use getShape method to get object of type shape

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

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

return new Circle();

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

return new Rectangle();

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

return new Square();

}

return null;

}

}

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

*FactoryPatternDemo.java*

public class FactoryPatternDemo {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

//get an object of Circle and call its draw method.

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

//call draw method of Circle

shape1.draw();

//get an object of Rectangle and call its draw method.

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

//call draw method of Rectangle

shape2.draw();

//get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of square

shape3.draw();

}

}

**Builder**

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

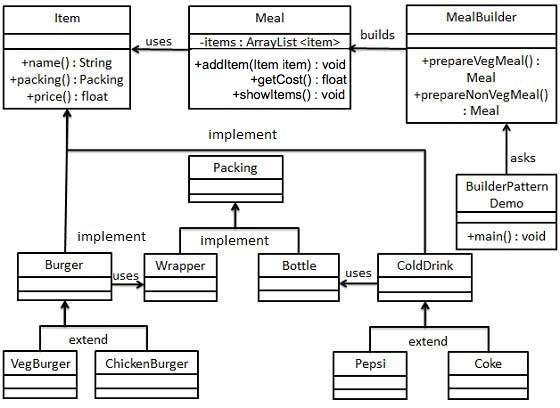
A Builder class builds the final object step by step. This builder is independent of other objects.

Implementation

We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.

We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle.

We then create a *Meal* class having *ArrayList* of *Item* and a *MealBuilder* to build different types of *Meal* objects by combining *Item*. *BuilderPatternDemo*, our demo class will use *MealBuilder* to build a *Meal*.



Step 1

Create an interface Item representing food item and packing.

*Item.java*

public interface Item {

public String name();

public Packing packing();

public float price();

}

*Packing.java*

public interface Packing {

public String pack();

}

Step 2

Create concrete classes implementing the Packing interface.

*Wrapper.java*

public class Wrapper implements Packing {

@Override

public String pack() {

return "Wrapper";

}

}

*Bottle.java*

public class Bottle implements Packing {

@Override

public String pack() {

return "Bottle";

}

}

Step 3

Create abstract classes implementing the item interface providing default functionalities.

*Burger.java*

public abstract class Burger implements Item {

@Override

public Packing packing() {

return new Wrapper();

}

@Override

public abstract float price();

}

*ColdDrink.java*

public abstract class ColdDrink implements Item {

@Override

public Packing packing() {

return new Bottle();

}

@Override

public abstract float price();

}

Step 4

Create concrete classes extending Burger and ColdDrink classes

*VegBurger.java*

public class VegBurger extends Burger {

@Override

public float price() {

return 25.0f;

}

@Override

public String name() {

return "Veg Burger";

}

}

*ChickenBurger.java*

public class ChickenBurger extends Burger {

@Override

public float price() {

return 50.5f;

}

@Override

public String name() {

return "Chicken Burger";

}

}

*Coke.java*

public class Coke extends ColdDrink {

@Override

public float price() {

return 30.0f;

}

@Override

public String name() {

return "Coke";

}

}

*Pepsi.java*

public class Pepsi extends ColdDrink {

@Override

public float price() {

return 35.0f;

}

@Override

public String name() {

return "Pepsi";

}

}

Step 5

Create a Meal class having Item objects defined above.

*Meal.java*

import java.util.ArrayList;

import java.util.List;

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());

}

}

}

Step 6

Create a MealBuilder class, the actual builder class responsible to create Meal objects.

*MealBuilder.java*

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;

}

}

Step 7

BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

*BuilderPatternDemo.java*

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());

}

}