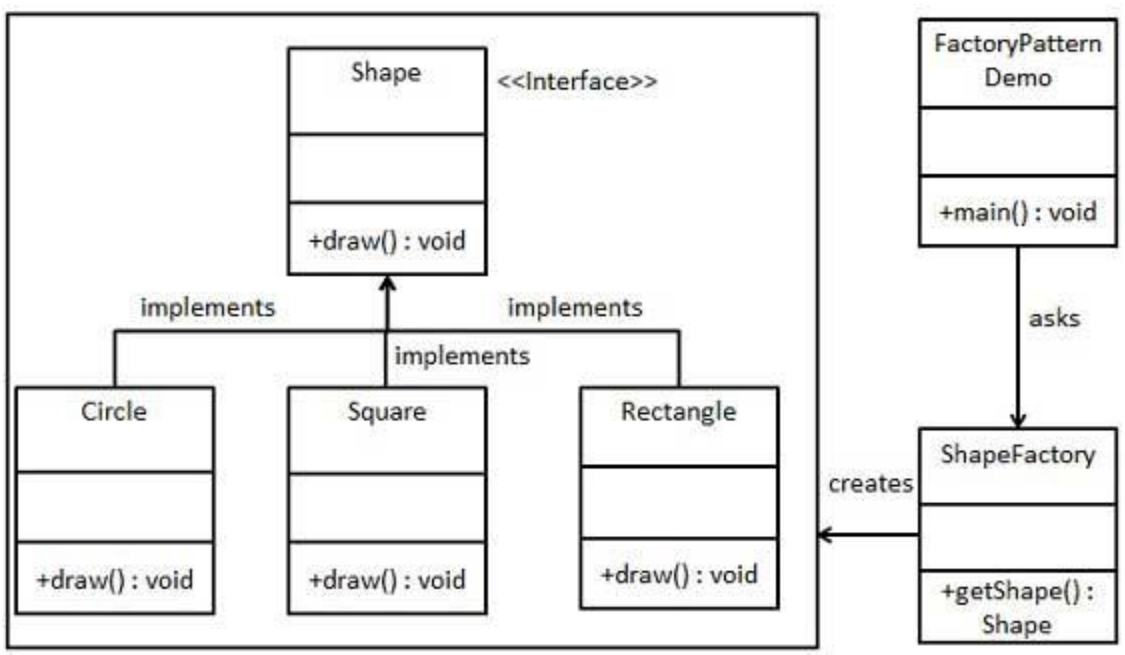
Factory Pattern

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.



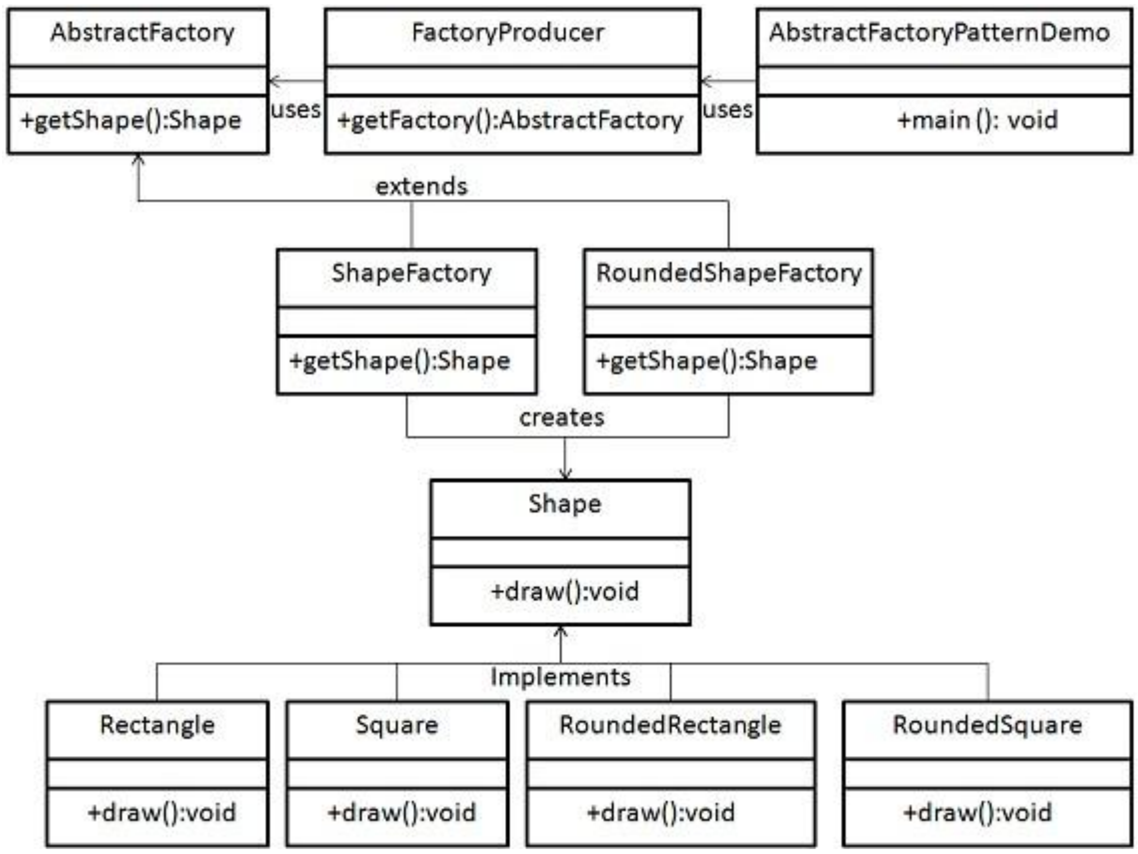
```
public class FactoryPatternDemo {


  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();
   }
}

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;
   }
}

interface Shape {
   void draw();
}

class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}

class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}

class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Abstract Factory pattern

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.



```
public class AbstractFactoryPatternDemo {

  public static void main(String[] args) {

    //get shape factory

    AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

    //get an object of Shape Rectangle

    Shape shape1 = shapeFactory.getShape("RECTANGLE");
    //call draw method of Shape Rectangle
    shape1.draw();
    //get an object of Shape Square
    Shape shape2 = shapeFactory.getShape("SQUARE");
    //call draw method of Shape Square
    shape2.draw();
    //get shape factory
    AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
    //get an object of Shape Rectangle
    Shape shape3 = shapeFactory1.getShape("RECTANGLE");
    //call draw method of Shape Rectangle
    shape3.draw();
    //get an object of Shape Square
    Shape shape4 = shapeFactory1.getShape("SQUARE");
    //call draw method of Shape Square
    shape4.draw();
  }
}

class FactoryProducer {
  public static AbstractFactory getFactory(boolean rounded){
    if(rounded){
      return new RoundedShapeFactory();
    }else{
      return new ShapeFactory();
    }
  }
}
```

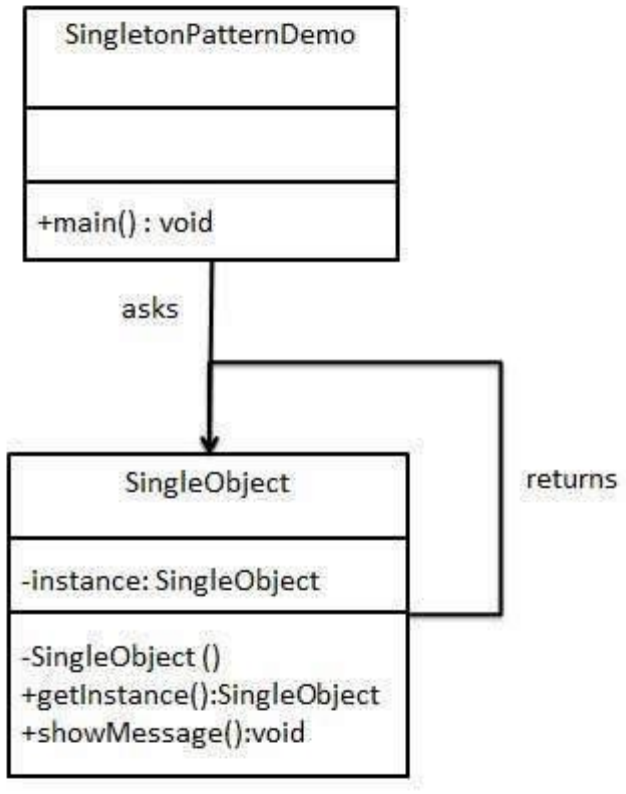
```
interface Shape {
  void draw();
}

class RoundedRectangle implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside RoundedRectangle::draw() method.");
  }
}

class RoundedSquare implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside RoundedSquare::draw() method.");
  }
}

class Rectangle implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
  }
}

class Square implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}

abstract class AbstractFactory {
  abstract Shape getShape(String shapeType) ;
}

class ShapeFactory extends AbstractFactory {
  @Override
  public Shape getShape(String shapeType){
    if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();
    }else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }
    return null;
  }
}

class RoundedShapeFactory extends AbstractFactory {
  @Override
  public Shape getShape(String shapeType){
    if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new RoundedRectangle();
    }else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new RoundedSquare();
    }
    return null;
  }
```

```
}
```

Singleton pattern

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.

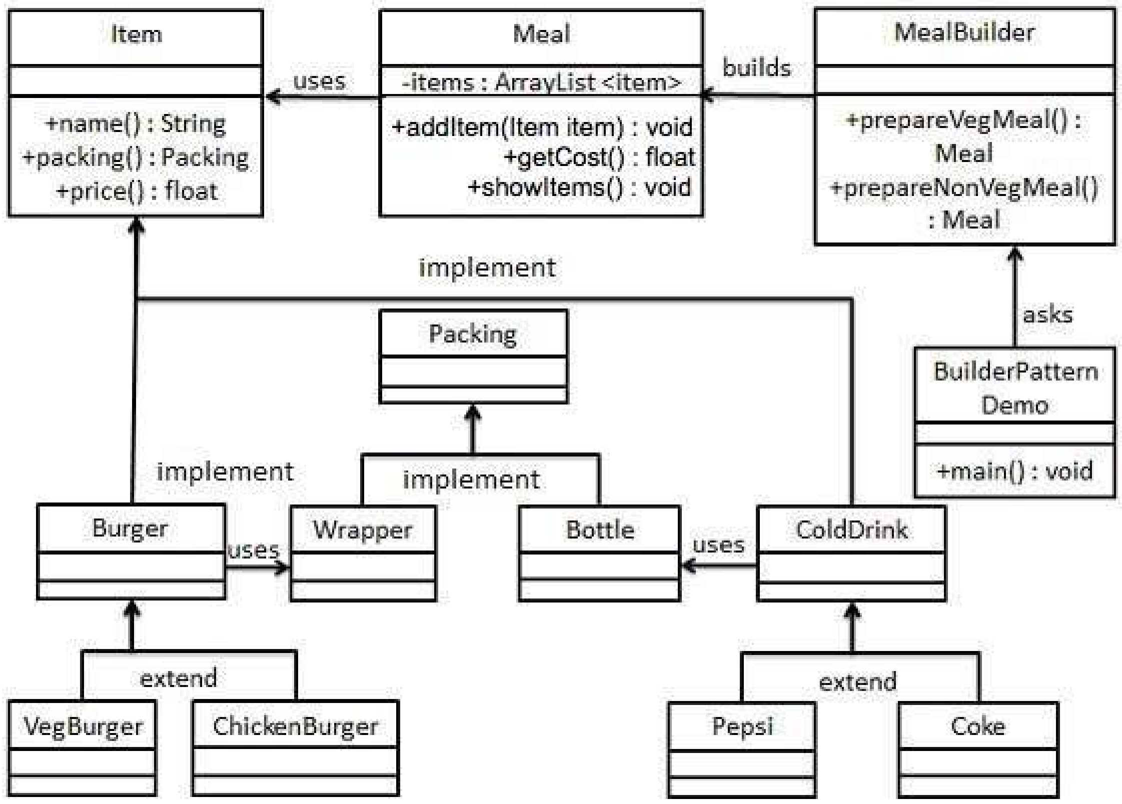


```
public class SingletonPatternDemo {
  public static void main(String[] args) {

    //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();
  }
}

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

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.



```
import java.util.ArrayList;

import java.util.List;


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());
  }
}

interface Item {
  public String name();
  public Packing packing();
  public float price();
}

interface Packing {
  public String pack();
}

class Wrapper implements Packing {
  @Override
  public String pack() {
    return "Wrapper";
  }
}

class Bottle implements Packing {
  @Override
  public String pack() {
    return "Bottle";
  }
}
```

```
abstract class Burger implements Item {

  @Override
  public Packing packing() {
    return new Wrapper();
  }

  @Override
  public abstract float price();
}

abstract class ColdDrink implements Item {

          @Override
          public Packing packing() {
     return new Bottle();
          }

          @Override
          public abstract float price();
}

class VegBurger extends Burger {

  @Override
  public float price() {
    return 25.0f;
  }
  @Override
  public String name() {
    return "Veg Burger";
  }
}

class ChickenBurger extends Burger {

  @Override
  public float price() {
    return 50.5f;
  }

  @Override
  public String name() {
    return "Chicken Burger";
  }
}

class Coke extends ColdDrink {

  @Override
  public float price() {
    return 30.0f;
  }

  @Override
  public String name() {
    return "Coke";
  }
}

class Pepsi extends ColdDrink {

  @Override
  public float price() {
    return 35.0f;
  }

  @Override
  public String name() {
    return "Pepsi";
  }
}

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());
    }
  }
}

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;
  }
```

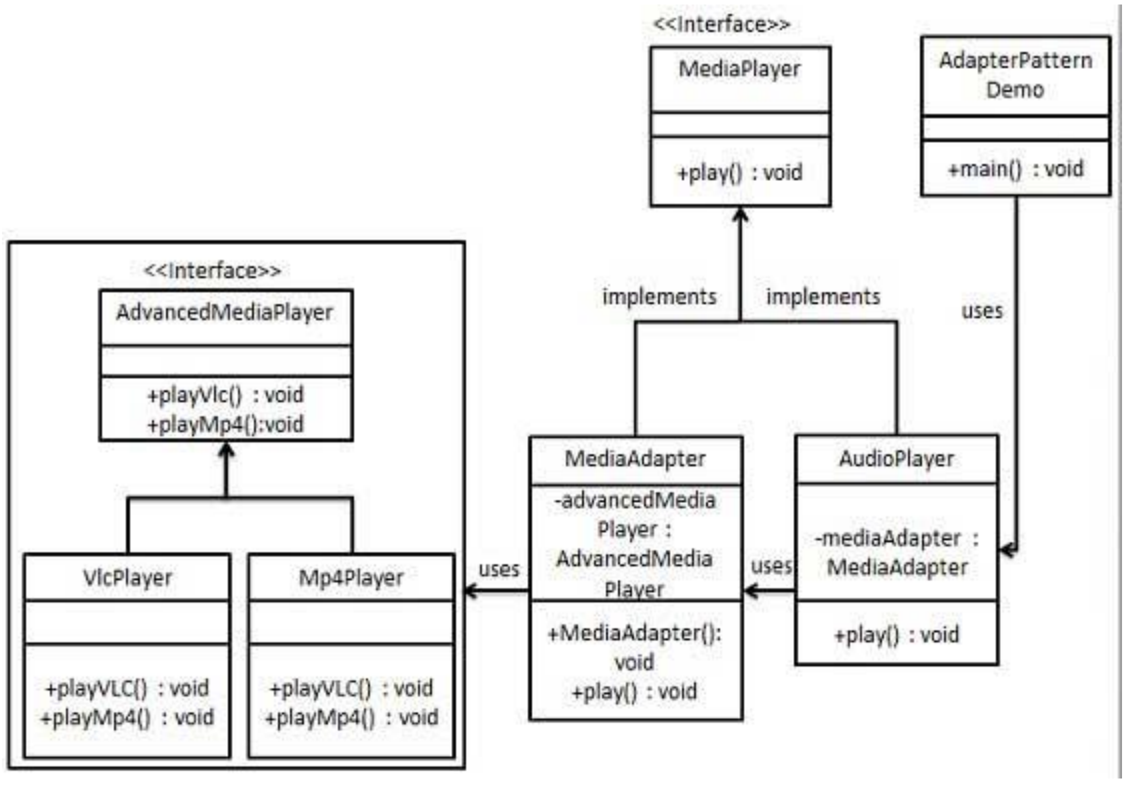
```
}
```

Adapter pattern

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

We are demonstrating use of Adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

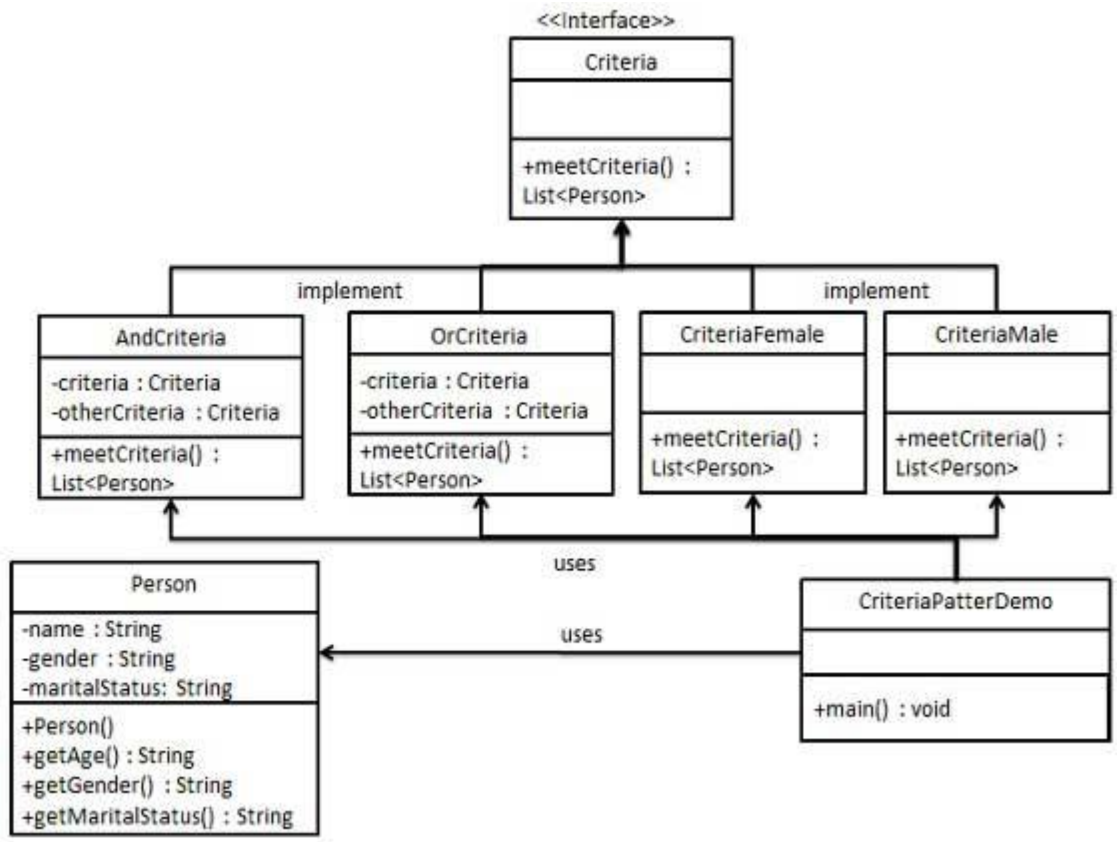


```
public class AdapterPatternDemo {
  public static void main(String[] args) {
    AudioPlayer audioPlayer = new AudioPlayer();

    audioPlayer.play("mp3", "beyond the horizon.mp3");
    audioPlayer.play("mp4", "alone.mp4");
    audioPlayer.play("vlc", "far far away.vlc");
    audioPlayer.play("avi", "mind me.avi");
  }
}

interface MediaPlayer {
  public void play(String audioType, String fileName);
}

interface AdvancedMediaPlayer {
  public void playVlc(String fileName);
  public void playMp4(String fileName);
}

class VlcPlayer implements AdvancedMediaPlayer{
  @Override
  public void playVlc(String fileName) {
    System.out.println("Playing vlc file. Name: "+ fileName);
  }

  @Override
  public void playMp4(String fileName) {
    //do nothing
  }
```

```
}

class Mp4Player implements AdvancedMediaPlayer{

  @Override
  public void playVlc(String fileName) {
    //do nothing
  }

  @Override
  public void playMp4(String fileName) {
    System.out.println("Playing mp4 file. Name: "+ fileName);
  }
}

class MediaAdapter implements MediaPlayer {

  AdvancedMediaPlayer advancedMusicPlayer;

  public MediaAdapter(String audioType){

    if(audioType.equalsIgnoreCase("vlc") ){
      advancedMusicPlayer = new VlcPlayer();

    }else if (audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer = new Mp4Player();
    }
  }
  @Override
  public void play(String audioType, String fileName) {

    if(audioType.equalsIgnoreCase("vlc")){
      advancedMusicPlayer.playVlc(fileName);
    }
    else if(audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer.playMp4(fileName);
    }
  }
}

class AudioPlayer implements MediaPlayer {
  MediaAdapter mediaAdapter;

  @Override
  public void play(String audioType, String fileName) {

    //inbuilt support to play mp3 music files
    if(audioType.equalsIgnoreCase("mp3")){
      System.out.println("Playing mp3 file. Name: " + fileName);
    }

    //mediaAdapter is providing support to play other file formats
    else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
      mediaAdapter = new MediaAdapter(audioType);
      mediaAdapter.play(audioType, fileName);
    }
    else{
      System.out.println("Invalid media. " + audioType + " format not supported");
    }
  }
}
```

Filter pattern

Filter pattern or Criteria pattern is a design pattern that enables developers to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations. This type of design pattern comes under structural pattern as this pattern combines multiple criteria to obtain single criteria.



Step 1

Create a class on which criteria is to be applied.

Person.java

```
package com.tutorialspoint;


public class Person {


   private String name;

   private String gender;

   private String maritalStatus;
```

```
  public Person(String name, String gender, String maritalStatus){
    this.name = name;
    this.gender = gender;
    this.maritalStatus = maritalStatus;
  }

  public String getName() {
    return name;
  }
  public String getGender() {
    return gender;
  }
  public String getMaritalStatus() {
    return maritalStatus;
  }
}
```

Step 2

Create an interface for Criteria.

Criteria.java

```
package com.tutorialspoint;

import java.util.List;

public interface Criteria {
  public List<Person> meetCriteria(List<Person> persons);
}
```

Step 3

Create concrete classes implementing the Criteria interface.

CriteriaMale.java

```
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.List;

public class CriteriaMale implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> malePersons = new ArrayList<Person>();

      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("MALE")){
            malePersons.add(person);
         }
      }
      return malePersons;
   }
}
```

CriteriaFemale.java

```
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.List;

public class CriteriaFemale implements Criteria {
  @Override
  public List<Person> meetCriteria(List<Person> persons) {
    List<Person> femalePersons = new ArrayList<Person>();

    for (Person person : persons) {
      if(person.getGender().equalsIgnoreCase("FEMALE")){
        femalePersons.add(person);
      }
    }
    return femalePersons;
  }
}
```

CriteriaSingle.java

```
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.List;

public class CriteriaSingle implements Criteria {

  @Override
  public List<Person> meetCriteria(List<Person> persons) {
    List<Person> singlePersons = new ArrayList<Person>();

    for (Person person : persons) {
      if(person.getMaritalStatus().equalsIgnoreCase("SINGLE")){
        singlePersons.add(person);
      }
    }
    return singlePersons;
  }
}
```

AndCriteria.java

```
package com.tutorialspoint;

import java.util.List;

public class AndCriteria implements Criteria {

  private Criteria criteria;
  private Criteria otherCriteria;

  public AndCriteria(Criteria criteria, Criteria otherCriteria) {
    this.criteria = criteria;
    this.otherCriteria = otherCriteria;
  }

  @Override
  public List<Person> meetCriteria(List<Person> persons) {

    List<Person> firstCriteriaPersons = criteria.meetCriteria(persons);
    return otherCriteria.meetCriteria(firstCriteriaPersons);
  }
}
```

OrCriteria.java

```
package com.tutorialspoint;
```

```
import java.util.List;

public class OrCriteria implements Criteria {

  private Criteria criteria;
  private Criteria otherCriteria;

  public OrCriteria(Criteria criteria, Criteria otherCriteria) {
    this.criteria = criteria;
    this.otherCriteria = otherCriteria;
  }

  @Override
  public List<Person> meetCriteria(List<Person> persons) {
    List<Person> firstCriteriaItems = criteria.meetCriteria(persons);
    List<Person> otherCriteriaItems = otherCriteria.meetCriteria(persons);

    for (Person person : otherCriteriaItems) {
      if(!firstCriteriaItems.contains(person)){
        firstCriteriaItems.add(person);
      }
    }
    return firstCriteriaItems;
  }
}
```

Example - Usage of Filter Pattern

Use different Criteria and their combination to filter out persons.

CriteriaPatternDemo.java

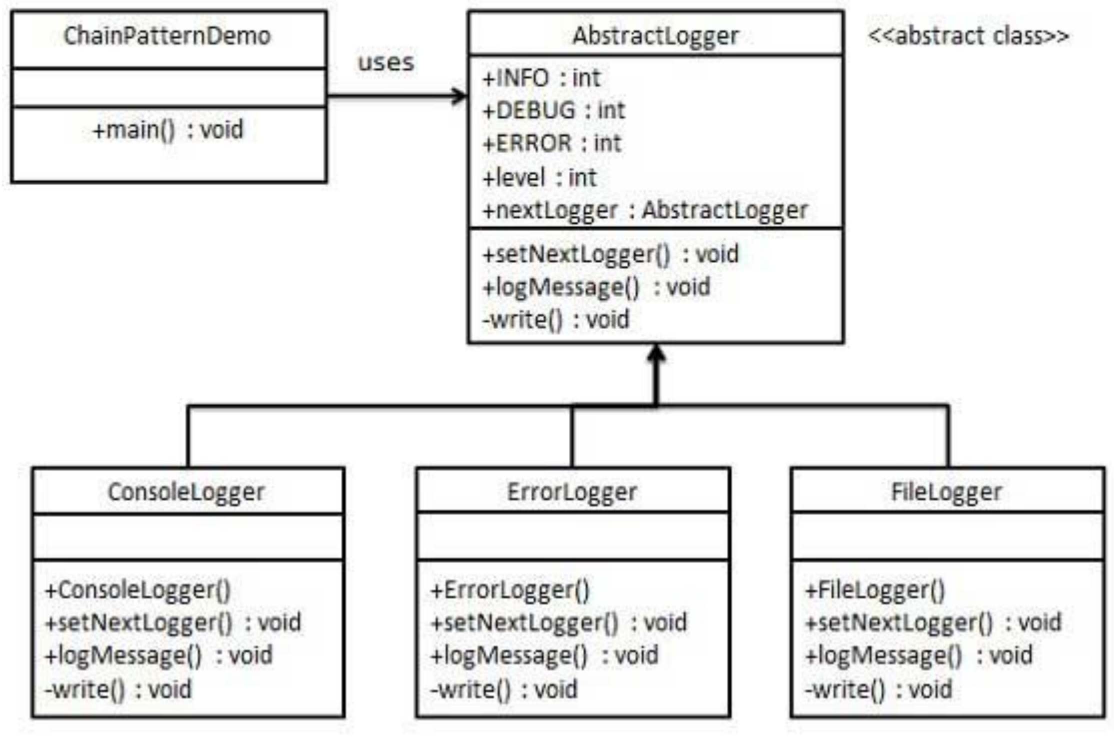
```java
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.List;

public class CriteriaPatternDemo {
  public static void main(String[] args) {
    List<Person> persons = new ArrayList<Person>();

    persons.add(new Person("Robert","Male", "Single"));
    persons.add(new Person("John", "Male", "Married"));
    persons.add(new Person("Laura", "Female", "Married"));
    persons.add(new Person("Diana", "Female", "Single"));
    persons.add(new Person("Mike", "Male", "Single"));
    persons.add(new Person("Bobby", "Male", "Single"));

    Criteria male = new CriteriaMale();
    Criteria female = new CriteriaFemale();
    Criteria single = new CriteriaSingle();
    Criteria singleMale = new AndCriteria(single, male);
    Criteria singleOrFemale = new OrCriteria(single, female);

    System.out.println("Males: ");
    printPersons(male.meetCriteria(persons));

    System.out.println("\nFemales: ");
    printPersons(female.meetCriteria(persons));
```

```
      System.out.println("\nSingle Males: ");
      printPersons(singleMale.meetCriteria(persons));

      System.out.println("\nSingle Or Females: ");
      printPersons(singleOrFemale.meetCriteria(persons));
   }

   public static void printPersons(List<Person> persons){

      for (Person person : persons) {
         System.out.println("Person : [ Name : " + person.getName() + ", Gender : " + person.getGender() +
", Marital Status : " + person.getMaritalStatus() + " ]");
      }
   }
}
```

Chain of Responsibility pattern

As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.

In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.



Step 1

Create an abstract logger class.

AbstractLogger.java

```
package com.tutorialspoint;
```

```
public abstract class AbstractLogger {
  public static int INFO = 1;
  public static int DEBUG = 2;
  public static int ERROR = 3;

  protected int level;

  //next element in chain or responsibility
  protected AbstractLogger nextLogger;

  public void setNextLogger(AbstractLogger nextLogger){
    this.nextLogger = nextLogger;
  }

  public void logMessage(int level, String message){
    if(this.level <= level){
      write(message);
    }
    if(nextLogger !=null){
      nextLogger.logMessage(level, message);
    }
  }

  abstract protected void write(String message);

}
```

Step 2

Create concrete classes extending the logger.

ConsoleLogger.java

```
package com.tutorialspoint;

public class ConsoleLogger extends AbstractLogger {

  public ConsoleLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("Standard Console::Logger: " + message);
  }
}
```

ErrorLogger.java

```
package com.tutorialspoint;

public class ErrorLogger extends AbstractLogger {

  public ErrorLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("Error Console::Logger: " + message);
  }
}
```

FileLogger.java

```
package com.tutorialspoint;


public class FileLogger extends AbstractLogger {


  public FileLogger(int level){
    this.level = level;
  }


  @Override
  protected void write(String message) {
    System.out.println("File::Logger: " + message);
  }
}
```

Example - Usage of Chain of Responsibility Pattern

Create different types of loggers. Assign them error levels and set next logger in each logger. Next logger in each logger represents the part of the chain.

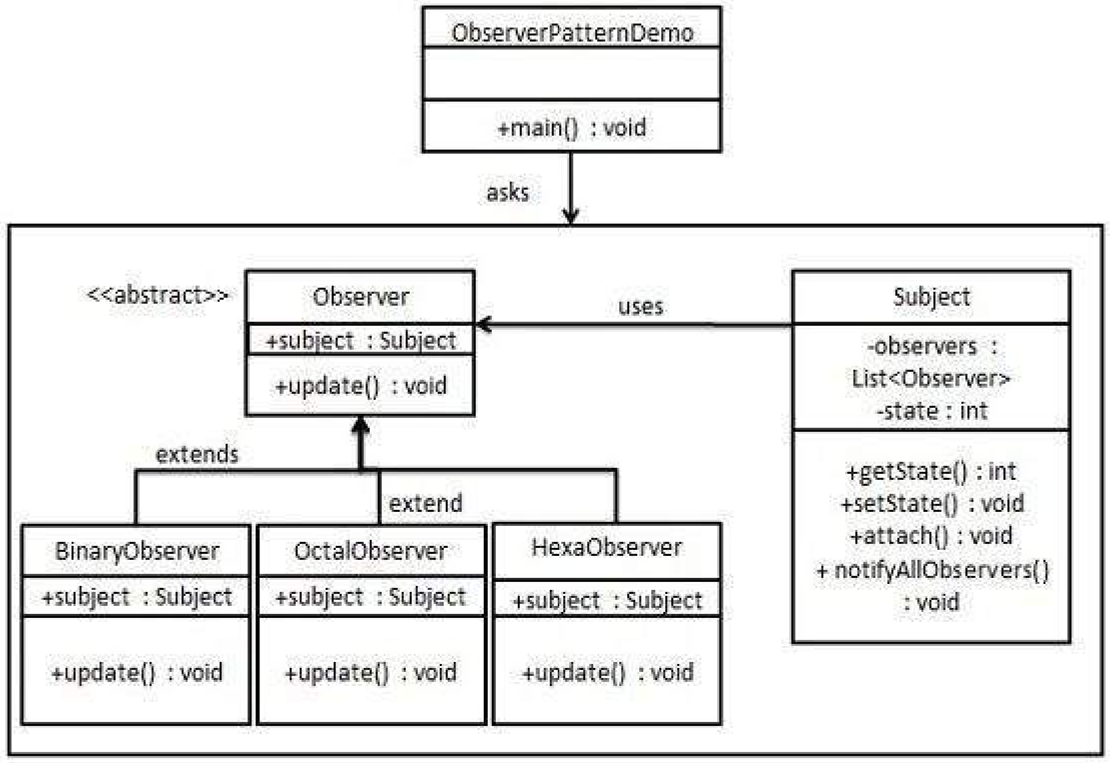
ChainPatternDemo.java

```
package com.tutorialspoint;


public class ChainPatternDemo {


  private static AbstractLogger getChainOfLoggers(){


    AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
    AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
    AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
    errorLogger.setNextLogger(fileLogger);
    fileLogger.setNextLogger(consoleLogger);

    return errorLogger;
  }

  public static void main(String[] args) {
    AbstractLogger loggerChain = getChainOfLoggers();

    loggerChain.logMessage(AbstractLogger.INFO,
      "This is an information.");

    loggerChain.logMessage(AbstractLogger.DEBUG,
      "This is an debug level information.");

    loggerChain.logMessage(AbstractLogger.ERROR,
      "This is an error information.");
  }
}

abstract class AbstractLogger {
  public static int INFO = 1;
  public static int DEBUG = 2;
  public static int ERROR = 3;

  protected int level;

  //next element in chain or responsibility
  protected AbstractLogger nextLogger;
  public void setNextLogger(AbstractLogger nextLogger){
    this.nextLogger = nextLogger;
  }

  public void logMessage(int level, String message){
    if(this.level <= level){
      write(message);
    }
    if(nextLogger !=null){
      nextLogger.logMessage(level, message);
    }
  }

  abstract protected void write(String message);

}

class ConsoleLogger extends AbstractLogger {

  public ConsoleLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("Standard Console::Logger: " + message);
  }
}
```

```
class ErrorLogger extends AbstractLogger {

  public ErrorLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("Error Console::Logger: " + message);
  }
}

class FileLogger extends AbstractLogger {

  public FileLogger(int level){
    this.level = level;
  }

  @Override
  protected void write(String message) {
    System.out.println("File::Logger: " + message);
  }
}
```

Observer design pattern

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.



```
import java.util.ArrayList;
import java.util.List;

public class ObserverPatternDemo {
  public static void main(String[] args) {
    Subject subject = new Subject();

    new HexaObserver(subject);
    new OctalObserver(subject);
    new BinaryObserver(subject);

    System.out.println("First state change: 15");
    subject.setState(15);
    System.out.println("Second state change: 10");
    subject.setState(10);
  }
}

class Subject {

  private List<Observer> observers = new ArrayList<Observer>();
  private int state;

  public int getState() {
    return state;
  }

  public void setState(int state) {
    this.state = state;
    notifyAllObservers();
  }

  public void attach(Observer observer){
    observers.add(observer);
  }

  public void notifyAllObservers(){
    for (Observer observer : observers) {
      observer.update();
    }
  }
}

abstract class Observer {
  protected Subject subject;
  public abstract void update();
}

class BinaryObserver extends Observer{

  public BinaryObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }

  @Override
  public void update() {
    System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
  }
```

```
}

class OctalObserver extends Observer{

   public OctalObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
     System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
   }
}

class HexaObserver extends Observer{

   public HexaObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );
   }
}
```