* Creational patterns :
* Singleton pattern :

Sometimes it is important to have only one instance of a class .below is the simple Implementation of Singleton pattern .

Class create

{

Public static Create c =new Create();

Private Create()

{

//initialize object .

}

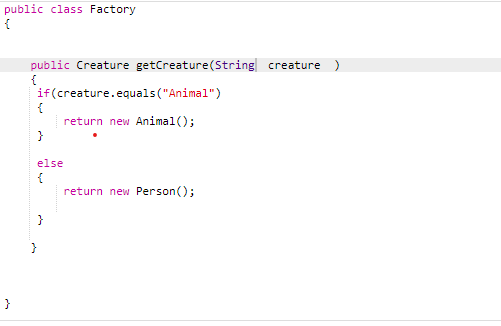
}

* **Factory pattern :**

The main goal of this pattern is to hide the object creation logic and directly give the object required by the consumer .

The implementation is really simple the consumer wants an object , instead of creating it directly using new operator , it asks factory object to provide an object it needs .

For example ,



* Abstract factory :

Step 1

Create an interface for Shapes and Colors.

*Shape.java*

public interface Shape {

void draw();

}

Step 2

Create concrete classes implementing the same interface.

*RoundedRectangle.java*

public class RoundedRectangle implements Shape {

@Override

public void draw() {

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

}

}

*RoundedSquare.java*

public class RoundedSquare implements Shape {

@Override

public void draw() {

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

}

}

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

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

}

}

Step 3

Create an Abstract class to get factories for Normal and Rounded Shape Objects.

*AbstractFactory.java*

public abstract class AbstractFactory {

abstract Shape getShape(String shapeType) ;

}

Step 4

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

*ShapeFactory.java*

public 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;

}

}

*RoundedShapeFactory.java*

public 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;

}

}

Step 5

Create a Factory generator/producer class to get factories by passing an information such as Shape

*FactoryProducer.java*

public class FactoryProducer {

public static AbstractFactory getFactory(boolean rounded){

if(rounded){

return new RoundedShapeFactory();

}else{

return new ShapeFactory();

}

}

}

Step 6

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

*AbstractFactoryPatternDemo.java*

public class AbstractFactoryPatternDemo {

public static void main(String[] args) {

AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

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

shape1.draw();

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

shape2.draw();

//get rounded shape factory

AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);

Shape shape3 = shapeFactory1.getShape("RECTANGLE");

shape3.draw();

Shape shape4 = shapeFactory1.getShape("SQUARE");

shape4.draw();

}

}

Step 7

Verify the output.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside RoundedRectangle::draw() method.

Inside RoundedSquare::draw() method.

* Builder Design Pattern :

These patterns serve their purpose and can be really useful . However, there are several use cases where we have to create a very complex object, which requires different steps and actions for each one. In such cases, the Builder design pattern can be really useful.

Lets consider a scenario where we want to send a mail which require too many complex tasks to complete like list of recipient. Message body ,protocol , etc.

package com.gkatzioura.design.creational.builder;

import java.util.HashSet;

import java.util.Set;

public class Email {

private final String title;

private final String recipients;

private final String message;

private Email(String title, String recipients, String message) {

this.title = title;

this.recipients = recipients;

this.message = message;

}

public String getTitle() {

return title;

}

public String getRecipients() {

return recipients;

}

public String getMessage() {

return message;

}

public void send() {

}

public static class EmailBuilder {

private Set recipients = new HashSet();

private String title;

private String greeting;

private String mainText;

private String closing;

public EmailBuilder addRecipient(String recipient) {

this.recipients.add(recipient);

return this;

}

public EmailBuilder removeRecipient(String recipient) {

this.recipients.remove(recipient);

return this;

}

public EmailBuilder setTitle(String title) {

this.title = title;

return this;

}

public EmailBuilder setGreeting(String greeting) {

this.greeting = greeting;

return this;

}

public EmailBuilder setMainText(String mainText) {

this.mainText = mainText;

return this;

}

public EmailBuilder setClosing(String closing) {

this.closing = closing;

return this;

}

public Email build() {

String message = greeting+"\n"+mainText+"\n"+closing;

String recipientSection = commaSeparatedRecipients();

return new Email(title,recipientSection,message);

}

private String commaSeparatedRecipients() {

StringBuilder sb = new StringBuilder();

for(String recipient:recipients) {

sb.append(",").append(recipient);

}

return sb.toString().replaceFirst(",","");

}

}

}

The end result of using the Builder pattern for creating an email will look like this:

Email email = new Email.EmailBuilder()

.addRecipient("john@Doe.com")

.setMainText("Check the builder pattern")

.setGreeting("Hi John!")

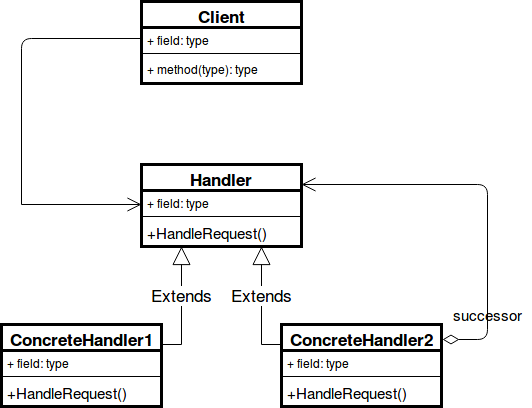
.setClosing("Regards")

.setTitle("Builder pattern resources")

.build();

* Behavioral patterns:
* Chain of responsibility :

Chain of responsibility is one of the way to achieve loose coupling .  request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.



**How to send a request in the application using the Chain of Responsibility**

The Client in need of a request to be handled sends it to the chain of handlers which are classes that extend the Handler class.  
Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client.  
If ConcreteHandler1 can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler2, the next one in the chain.

* Command

Like we have macros in microprocessor which is a set of command encapsulated as single command the same way we have the command design pattern lets see the below example of an home automation system.

|  |
| --- |
| interface Command  {      public void execute();  }    class Light  {      public void on()      {          System.out.println("Light is on");      }      public void off()      {          System.out.println("Light is off");      }  }  class LightOnCommand implements Command  {      Light light;        // The constructor is passed the light it      // is going to control.      public LightOnCommand(Light light)      {         this.light = light;      }      public void execute()      {         light.on();      }  }  class LightOffCommand implements Command  {      Light light;      public LightOffCommand(Light light)      {          this.light = light;      }      public void execute()      {           light.off();      }  }    // Stereo and its command classes  class Stereo  {      public void on()      {          System.out.println("Stereo is on");      }      public void off()      {          System.out.println("Stereo is off");      }      public void setCD()      {          System.out.println("Stereo is set " +                             "for CD input");      }      public void setDVD()      {          System.out.println("Stereo is set"+                           " for DVD input");      }      public void setRadio()      {          System.out.println("Stereo is set" +                             " for Radio");      }      public void setVolume(int volume)      {         // code to set the volume         System.out.println("Stereo volume set"                            + " to " + volume);      }  }  class StereoOffCommand implements Command  {      Stereo stereo;      public StereoOffCommand(Stereo stereo)      {          this.stereo = stereo;      }      public void execute()      {         stereo.off();      }  }  class StereoOnWithCDCommand implements Command  {       Stereo stereo;       public StereoOnWithCDCommand(Stereo stereo)       {           this.stereo = stereo;       }       public void execute()       {           stereo.on();           stereo.setCD();           stereo.setVolume(11);       }  }    class SimpleRemoteControl  {      Command slot;      public SimpleRemoteControl()      {      }        public void setCommand(Command command)      {            slot = command;      }        public void buttonWasPressed()      {          slot.execute();      }  }    class RemoteControlTest  {      public static void main(String[] args)      {          SimpleRemoteControl remote =                    new SimpleRemoteControl();          Light light = new Light();          Stereo stereo = new Stereo();            // we can change command dynamically          remote.setCommand(new                      LightOnCommand(light));          remote.buttonWasPressed();          remote.setCommand(new                  StereoOnWithCDCommand(stereo));          remote.buttonWasPressed();          remote.setCommand(new                     StereoOffCommand(stereo));          remote.buttonWasPressed();       }    } |

* Strategy

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

For example lets see the below implemented code ,

## Step 1

Create an interface.

*Strategy.java*

public interface Strategy {

public int doOperation(int num1, int num2);

}

## Step 2

Create concrete classes implementing the same interface.

*OperationAdd.java*

public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

*OperationSubstract.java*

public class OperationSubstract implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

*OperationMultiply.java*

public class OperationMultiply implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 \* num2;

}

}

## Step 3

Create *Context* Class.

*Context.java*

public class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public int executeStrategy(int num1, int num2){

return strategy.doOperation(num1, num2);

}

}

## Step 4

Use the *Context* to see change in behaviour when it changes its *Strategy*.

*StrategyPatternDemo.java*

public class StrategyPatternDemo {

public static void main(String[] args) {

Context context = new Context(new OperationAdd());

System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());

System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationMultiply());

System.out.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

* Structural pattern :
* Adapter Pattern:

**Definition:**

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Suppose you have a Bird class with fly() , and makeSound()methods. And also a ToyDuck class with squeak() method. Let’s assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can’t use them directly. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.

Below is Java implementation of it.

interface Bird

{

    public void fly();

    public void makeSound();

}

class Sparrow implements Bird

{

    public void fly()

    {

        System.out.println("Flying");

    }

    public void makeSound()

    {

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

    }

}

interface ToyDuck

{

    public void squeak();

}

class PlasticToyDuck implements ToyDuck

{

    public void squeak()

    {

        System.out.println("Squeak");

    }

}

class BirdAdapter implements ToyDuck

{

    Bird bird;

    public BirdAdapter(Bird bird)

    {

        this.bird = bird;

    }

    public void squeak()

    {

        bird.makeSound();

    }

}

class Main

{

    public static void main(String args[])

    {

        Sparrow sparrow = new Sparrow();

        ToyDuck toyDuck = new PlasticToyDuck();

        ToyDuck birdAdapter = new BirdAdapter(sparrow);

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

        sparrow.fly();

        sparrow.makeSound();

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

        toyDuck.squeak();

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

        birdAdapter.squeak();

    }

}

* Decorator pattern :

Decorator pattern is used to modify the functionality of object at runtime .

Let us create a Phone interface and BasicPhone class (having basic features) to implement it.

public interface Phone {

public void printModel();

}

public class BasicPhone implements Phone {

@Override

public void printModel() {

System.out.println("Basic Phone");;

}

}

Now, we will create a PhoneDecorator class that implements the Phoneinterface, having one Phone object as its property.

public class PhoneDecorator implements Phone {

public Phone phone;

public PhoneDecorator(Phone phone) {

this.phone = phone;

}

@Override

public void printModel() {

this.phone.printModel();

}

}

Now, let us create two more classes extending PhoneDecorator — say AndroidPhone and IPhone.

public class AndroidPhone extends PhoneDecorator {

public AndroidPhone(Phone phone) {

super(phone);

}

@Override

public void printModel() {

super.printModel();

System.out.println("Adding Features of Android");

}

}

Similarily,

public class IPhone extends PhoneDecorator {

public IPhone(Phone phone) {

super(phone);

}

@Override

public void printModel() {

super.printModel();

System.out.println("Adding Features of iPhone");

}

}

Now, let’s test it!

public class Test {

public static void main(String[] args) {

System.out.println("Test 1\n");

Phone phone = new AndroidPhone(new BasicPhone());

phone.printModel();

System.out.println("\nTest 2\n");

Phone phone1 = new IPhone(phone);

phone1.printModel();

}

}