**Tutorial No. 3**

**Problem statement:**

Extend Calculator application for :

1. Geometric Calculator
2. Statistical Calculator
3. Matrix Calculator

Apply Open Close Principle and use Factory method design pattern.

**Design Assumptions:**

In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" that is, such an entity can allow its behaviour to be extended without modifying its [source code](https://en.wikipedia.org/wiki/Source_code).

[Bertrand Meyer](https://en.wikipedia.org/wiki/Bertrand_Meyer) is generally credited for having originated the term open/closed principle, which appeared in his 1988 book [Object Oriented Software Construction](https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction).

* A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
* A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

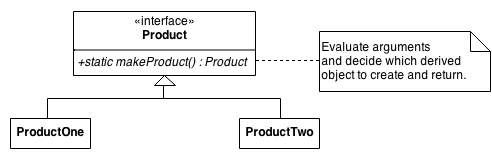
**Factory Design Pattern**

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client. Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

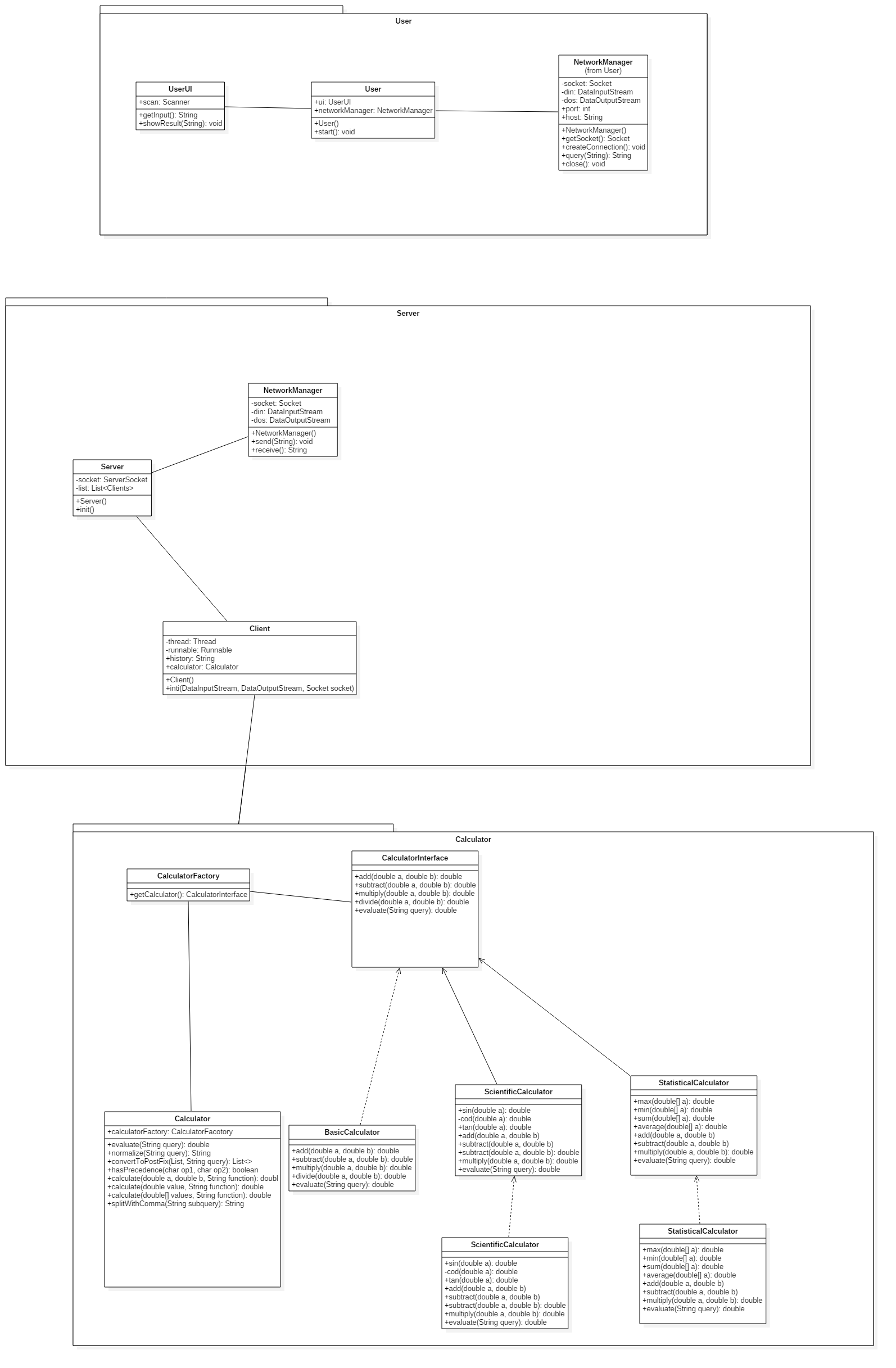
People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).

Factory Method is similar to Abstract Factory but without the emphasis on families.

Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

[](https://en.wikipedia.org/wiki/Open/closed_principle#cite_note-2)

**Design Diagrams:**



**Code:**

**server side**

**ClientHandler.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.server;

import com.mangnaik.yogesh.calculator.Calculator;

import com.mangnaik.yogesh.networkmanager.NetworkManagerServer;

import java.net.Socket;

/\*\*

\* Created by Yogesh on 2/2/2018.

\*/

class ClientHandler {

private Thread thread;

private Runnable runnable;

private String history;

private Calculator calculator;

ClientHandler(Socket socket) {

init(socket);

}

private void init(Socket socket){

NetworkManagerServer networkManager = new NetworkManagerServer(socket);

calculator = new Calculator();

history = "";

runnable = () -> {

while (true) {

System.out.println("ClientHandler is receiving");

String query = networkManager.listen();

System.out.println(query);

double answer = calculator.evaluate(query);

networkManager.send(answer+"");

}

};

thread = new Thread(runnable);

thread.start();

}

**Server.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.server;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.ArrayList;

import java.util.List;

import com.mangnaik.yogesh.calculator.Calculator;

/\*\*

\* Created by Yogesh on 2/2/2018.

\*/

public class Server {

private ServerSocket socket;

private List<ClientHandler> clients = new ArrayList<>();

private Server(){

Calculator calculator = new Calculator();

init();

}

private void init(){

try {

socket = new ServerSocket(8192);

} catch (IOException e) {

e.printStackTrace();

System.out.println("Failed to create Server!!");

}

while(true){

Socket s;

try{

s = socket.accept();

System.out.println("A new client has connected");

clients.add(new ClientHandler(s));

} catch (IOException e) {

e.printStackTrace();

break;

}

}

}

public static void main(String[] args){

new Server();

}

}

**User.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.user;

import com.mangnaik.yogesh.networkmanager.NetworkManagerClient;

/\*\*

\* Created by Yogesh on 2/2/2018.

\*/

public class User {

private UserUI ui;

private NetworkManagerClient networkManager;

public static void main(String args[]) {

new User();

}

private User() {

ui = new UserUI();

networkManager = new NetworkManagerClient("localhost", 8192);

init();

}

private void init(){

networkManager.createConnection();

String query = "";

while(!query.equals("exit")){

query = ui.getInput();

if(!query.equals("")){

String ans;

ans = networkManager.send(query);

ui.showResult(ans);

}

}

}

}

**UserUI.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.user;

import java.util.Scanner;

/\*\*

\* Created by Yogesh on 2/15/2018.

\*/

public class UserUI {

Scanner scan = new Scanner(System.in);

public String getInput(){

String query = scan.nextLine();

return query;

}

public void showResult(String answer){

System.out.println(answer);

}

}

**Calculator package**

**BasicCalculator.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

/\*\*

\* Created by Yogesh on 2/10/2018.

\*/

public class BasicCalculator implements BasicCalculatorInterface{

@Override

public double add(double a, double b) {

return a+b;

}

@Override

public double subtract(double a, double b) {

return b-a;

}

@Override

public double multiply(double a, double b) {

return a\*b;

}

@Override

public double divide(double a, double b) {

return a/b;

}

@Override

public double calculate(double[] values, String function) {

switch(function){

case "+":

return add(values[0], values[1]);

case "-":

return subtract(values[1], values[0]);

case "\*":

return multiply(values[0], values[1]);

case "/":

return divide(values[1], values[0]);

}

return 0;

}

}

**BasicCalculatorInterface.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

/\*\*

\* Created by Yogesh on 2/2/2018.

\*/

public interface BasicCalculatorInterface {

public double add(double a, double b);

public double subtract(double a, double b);

public double multiply(double a, double b);

public double divide(double a, double b);

public double calculate(double[] values, String function);

}

**Calculator.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

import com.sun.istack.internal.NotNull;

import java.util.ArrayList;

import java.util.List;

import java.util.Stack;

/\*\*

\* Created by Yogesh on 2/7/2018.

\*/

public class Calculator{

//call to evaluate the string query

public double evaluate(String query) {

query = query.replaceAll("pi", Math.PI+"");

List<String> list = new ArrayList<>();

query = normalize(query);

convertToPostFix(list, query);

return evaluate(list);

}

//evaluate the postfixed expression stored in list

private double evaluate(@NotNull List<String> list){

Stack<String> stack = new Stack<>();

for (String aList : list) {

if (aList.equals("+") || aList.equals("-") || aList.equals("\*") || aList.equals("/")) {

double a = Double.parseDouble(stack.pop());

double b = Double.parseDouble(stack.pop());

double[] values = new double[]{a, b};

stack.push("" + calculate(values, aList.charAt(0) + ""));

} else {

stack.push(aList);

}

}

return Double.valueOf(stack.pop());

}

//evaluate and replace all the trigonometric and statistical terms

private String normalize(String query){

String[] functions = new String[]{"sin", "cos", "tan", "atan", "asin", "acos", "log"};

for (String function : functions) {

while (query.contains(function)) {

int index = query.indexOf(function);

int start = index + function.length();

int end = start;

int count = 0;

for (int j = start; j < query.length(); j++) {

end++;

if (query.charAt(j) == '(') {

count++;

} else if (query.charAt(j) == ')') {

count--;

if (count == 0) {

break;

}

}

}

String subquery = query.substring(start, end);

query = query.substring(0, index) + calculate(new double[]{evaluate(subquery)}, function) + query.substring(end, query.length());

}

}

functions = new String[]{"min", "max", "avg", "sum"};

for (String function : functions) {

while (query.contains(function)) {

int index = query.indexOf(function);

int start = index + 3;

int end = start;

int count = 0;

for (int j = start; j < query.length(); j++) {

end++;

if (query.charAt(j) == '{') {

count++;

} else if (query.charAt(j) == '}') {

count--;

if (count == 0) {

break;

}

}

}

String subquery = query.substring(start + 1, end - 1);

double[] values = splitWithComma(subquery);

query = query.substring(0, index) + calculate(values, function) + query.substring(end, query.length());

}

}

return query;

}

//convert the string to postfix and store the result in a list

private void convertToPostFix(List<String> list, String query) {

Stack<Character> operators = new Stack<>();

char[] tokens = query.toCharArray();

for(int i=0; i<tokens.length; i++){

if (tokens[i]>='0'&&tokens[i]<='9'){

StringBuilder stringBuilder = new StringBuilder();

while (i < tokens.length && (tokens[i]=='.' || (tokens[i] >= '0' && tokens[i] <= '9'))){

stringBuilder.append(tokens[i++]);

}

i--;

list.add(stringBuilder.toString());

}

else if(tokens[i] == '(')

operators.push('(');

else if (tokens[i] == ')'){

while (operators.peek() != '(')

list.add(operators.pop()+"");

operators.pop();

}

else if (tokens[i] == '+' || tokens[i] == '-' || tokens[i] == '\*' || tokens[i] == '/'){

while (!operators.empty() && hasPrecedence(tokens[i], operators.peek())){

list.add(operators.pop()+"");

}

operators.push(tokens[i]);

}

}

while (!operators.empty())

list.add(operators.pop()+"");

}

//check for precedence

private boolean hasPrecedence(char op1, char op2) {

return op2 != '(' && op2 != ')' && ((op1 != '\*' && op1 != '/') || (op2 != '+' && op2 != '-'));

}

//call the evalutate function of calculators

private double calculate(double[] values, String function){

BasicCalculatorInterface calculator = CalculatorFactory.getCalculator(function);

return calculator.calculate(values, function);

}

//splitting with comma

private double[] splitWithComma(String subquery){

List<String> elementList = new ArrayList<>();

int bracketCount = 0;

int s = 0;

for(int k=0; k<subquery.length(); k++){

if(subquery.charAt(k)=='{'){

bracketCount++;

}

else if(subquery.charAt(k)=='}'){

bracketCount--;

}

if(bracketCount==0){

if(subquery.charAt(k)==','){

String st = subquery.substring(s,k);

elementList.add(st);

s=k+1;

}

}

}

if(!subquery.substring(s,subquery.length()).equals("")){

String st = subquery.substring(s, subquery.length());

elementList.add(st);

}

String[] elements = new String[elementList.size()];

for(int k=0; k<elementList.size(); k++){

elements[k] = elementList.get(k);

}

double[] values = new double[elements.length];

for(int j=0; j<values.length; j++){

values[j] = evaluate(elements[j]);

}

return values;

}

}

**CalculatorFactory.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Created by Yogesh on 2/7/2018.

\*/

public class CalculatorFactory {

private static List<String> scientificFunctions;

private static List<String> basicFunctions;

private static List<String> statisticalFunctions;

static{

scientificFunctions = new ArrayList<>();

basicFunctions = new ArrayList<>();

statisticalFunctions = new ArrayList<>();

scientificFunctions.add("sin");

scientificFunctions.add("cos");

scientificFunctions.add("tan");

basicFunctions.add("+");

basicFunctions.add("-");

basicFunctions.add("\*");

basicFunctions.add("/");

statisticalFunctions.add("max");

statisticalFunctions.add("min");

statisticalFunctions.add("avg");

statisticalFunctions.add("sum");

}

static BasicCalculatorInterface getCalculator(String function){

BasicCalculatorInterface basicCalculatorInterface = null;

if(scientificFunctions.contains(function)){

basicCalculatorInterface = new ScientificCalculator();

}

else if(basicFunctions.contains(function)){

basicCalculatorInterface = new BasicCalculator();

}

else if(statisticalFunctions.contains(function)){

basicCalculatorInterface = new StatisticalCalculator();

}

return basicCalculatorInterface;

}

}

**ScientificCalculator.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

/\*\*

\* Created by Yogesh on 2/3/2018.

\*/

public class ScientificCalculator implements ScientificCalculatorInterface{

@Override

public double sin(double angle) {

System.out.println("sin + " + angle);

System.out.println(Math.sin(angle));

return Math.sin(angle);

}

@Override

public double cos(double angle) {

return Math.cos(angle);

}

@Override

public double tan(double angle) {

return Math.tan(angle);

}

@Override

public double asin(double value) {

return Math.asin(value);

}

@Override

public double acos(double value) {

return Math.acos(value);

}

@Override

public double atan(double value) {

return Math.atan(value);

}

@Override

public double log(double number) {

return Math.log(number);

}

@Override

public double add(double a, double b) {

return a+b;

}

@Override

public double subtract(double a, double b) {

return a-b;

}

@Override

public double multiply(double a, double b) {

return a\*b;

}

@Override

public double divide(double a, double b) {

return a/b;

}

@Override

public double calculate(double[] values, String function) {

switch(function){

case "sin":

return sin(values[0]);

case "cos":

return cos(values[0]);

case "tan":

return tan(values[0]);

case "asin":

return asin(values[0]);

case "acos":

return acos(values[0]);

case "atan":

return atan(values[0]);

case "log":

return log(values[0]);

}

return 0;

}

}

**ScientificCalculatorInterface.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

public interface ScientificCalculatorInterface extends BasicCalculatorInterface {

//Trigonometric

public double sin(double angle);

public double cos(double angle);

public double tan(double angle);

}

**StatisticalCalculator.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

/\*\*

\* Created by Yogesh on 2/7/2018.

\*/

public class StatisticalCalculator implements StatisticalCalculatorInterface{

@Override

public double add(double a, double b) {

return a+b;

}

@Override

public double subtract(double a, double b) {

return a-b;

}

@Override

public double multiply(double a, double b) {

return a\*b;

}

@Override

public double divide(double a, double b) {

return a/b;

}

@Override

public double average(double[] values) {

double sum = 0;

for (double number : values) {

sum += number;

}

return sum/values.length;

}

@Override

public double max(double[] values) {

double max = -Integer.MAX\_VALUE;

for (double number : values) {

if (number > max) {

max = number;

}

}

return max;

}

@Override

public double min(double[] values) {

double min = Integer.MAX\_VALUE;

for (double number : values) {

if (number < min) {

min = number;

}

}

return min;

}

@Override

public double sum(double[] values) {

double sum = 0;

for (double number : values) {

sum += number;

}

return sum;

}

@Override

public double calculate(double[] values, String function) {

switch (function){

case "max":

return max(values);

case "min":

return min(values);

case "avg":

return average(values);

case "sum":

return sum(values);

}

return 0;

}

}

**StatisticalCalculatorInterface.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.calculator;

public interface StatisticalCalculatorInterface extends BasicCalculatorInterface {

public double average(double[] numbers);

public double max(double[] numbers);

public double min(double[] numbers);

public double sum(double[] numbers);

}

**NetworkManager package**

**NetworkManagerClient.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.networkmanager;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.Socket;

public class NetworkManagerClient {

private Socket socket;

private DataInputStream dis;

private DataOutputStream dos;

private int port;

private String host;

public NetworkManagerClient(String host, int port){

this.port = port;

this.host = host;

}

private Socket getSocket() throws IOException {

return new Socket(host, port);

}

public void createConnection(){

try {

socket = getSocket();

} catch (IOException e) {

e.printStackTrace();

return;

}

try {

dis = new DataInputStream(socket.getInputStream());

dos = new DataOutputStream(socket.getOutputStream());

} catch (IOException e) {

e.printStackTrace();

}

}

public String send(String query) {

String answer = "";

try {

dos.writeUTF(query);

answer = dis.readUTF();

} catch (IOException e) {

close();

e.printStackTrace();

}

return answer;

}

private void close() {

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**NetworkManagerServer.java**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.mangnaik.yogesh.networkmanager;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.Socket;

public class NetworkManagerServer {

private DataInputStream dis;

private DataOutputStream dos;

private Socket socket;

public NetworkManagerServer(Socket socket) {

this.socket = socket;

try {

this.dis = new DataInputStream(socket.getInputStream());

this.dos = new DataOutputStream(socket.getOutputStream());

} catch (IOException e) {

e.printStackTrace();

}

}

public String listen(){

String received;

try {

received = dis.readUTF();

return received;

} catch (IOException e) {

close();

e.printStackTrace();

}

return "";

}

public void send(String answer){

try{

dos.writeUTF(answer);

}

catch (Exception e){

e.printStackTrace();

}

}

private void close() {

try {

socket.close();

} catch (IOException e) {

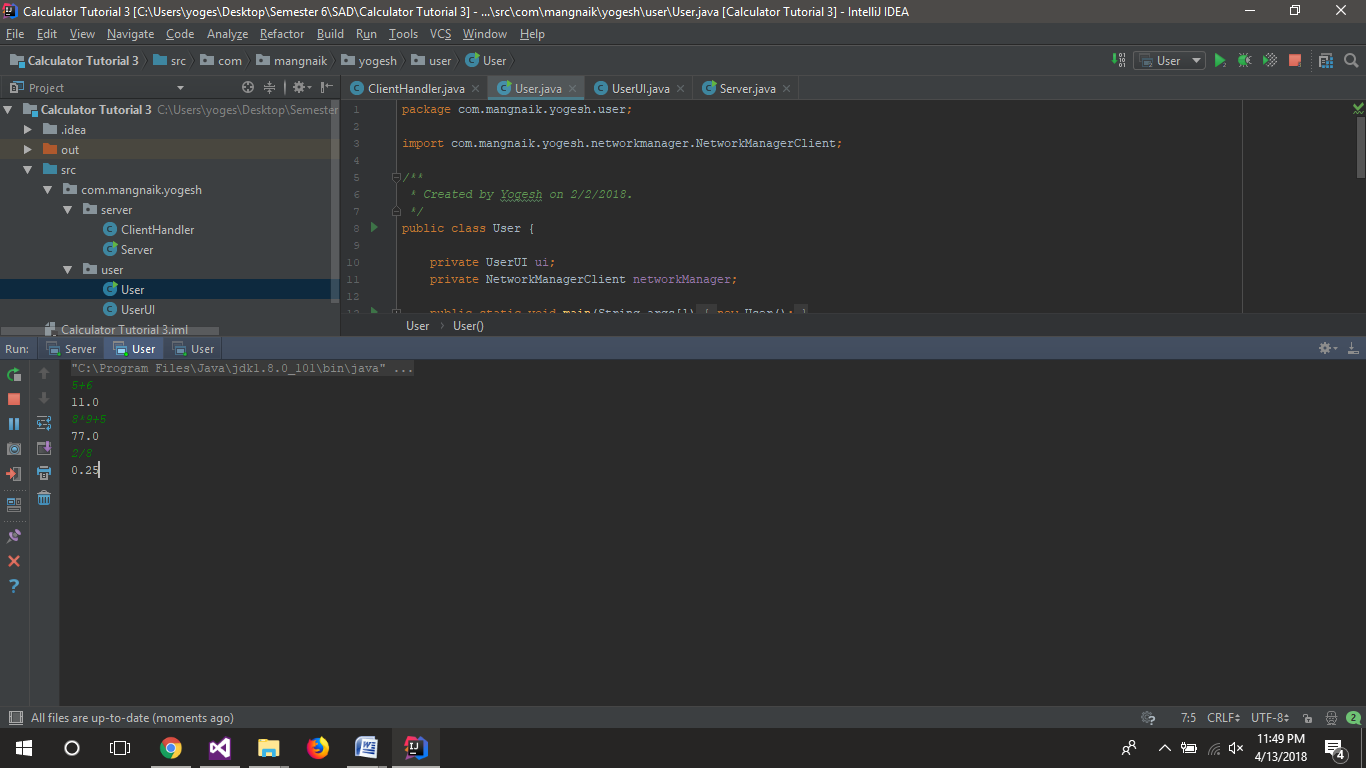
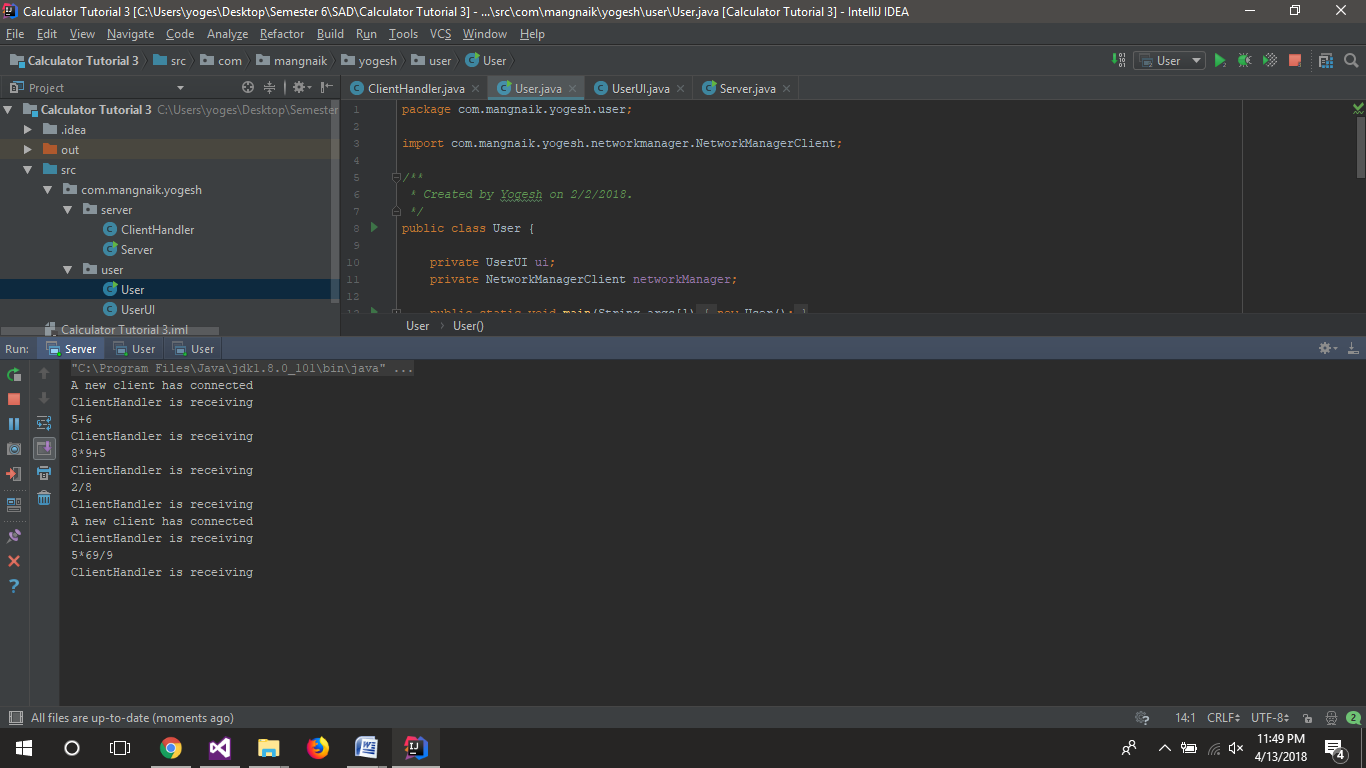
e.printStackTrace();

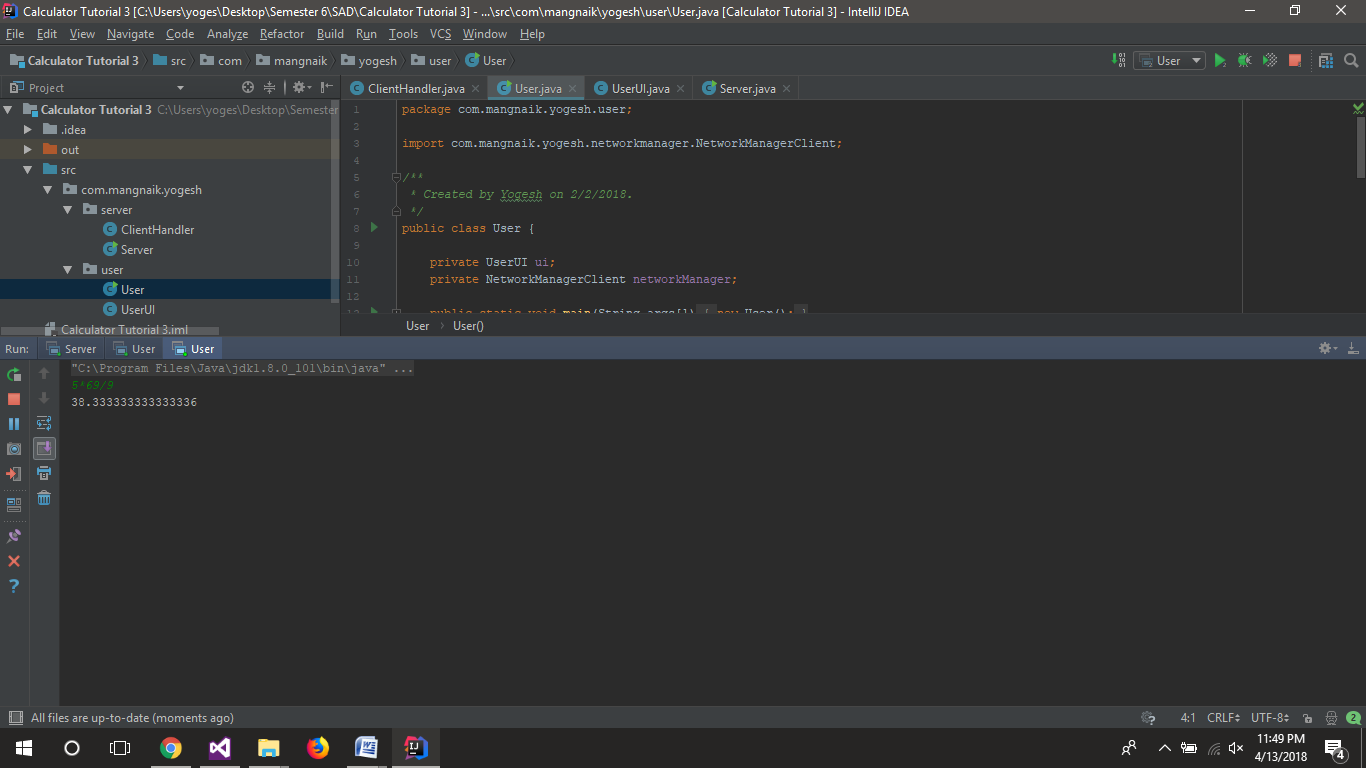
}

}

}

**Output:**





**Observation:**

While implementing open close principle we have to make a base class which will be an interface and declare all the methods which will be used by the classes that implement this interface. While designing this interface care has to be taken to not violate other principles such as including unnecessary functions in the interface.

While designing the factory method of producing objects we have to consider which type of input we should get to decide which object we should produce. Here we have taken string which tells the factory which calculator the user needs. We can also use constants defined in other class to tell which object is needed at the runtime. Factory pattern is very useful for extensibility and runtime binding of objects.