ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ОЛЕСЯ ГОНЧАРА

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ МАТЕМАТИКИ ТА МАТЕМАТИЧНОЇ КІБЕРНЕТИКИ

ЗВІТ ПРО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ **№ 2**

З дисципліни «Сучасні середовища програмування»

Перший (бакалаврський) рівень вищої освіти

Спеціальність 113 Прикладна математика

Освітня програма Комп’ютерне моделювання

та технології програмування

Виконавець:

Студентка групи ПА–22–1

Пономаренко Марина

Варіант № 5

Дніпро

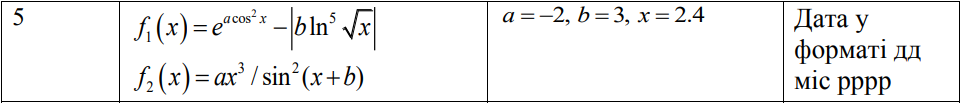
2023

1. ***Постановка завдання.***

− Застосувати патерн Компонувальник (Composite) для задачі обчислення

похідної довільної складеної функції. Використати дані табл.1.

− **Варіант №5**



1. ***Опис логічної структури Java-програми.***

Структура: 11 класів + Fucntion(interface) + клас Task для виконання програми

1. **Composite**

Є абстрактним базовим класом, який використовується для зберігання та роботи з колекцією функцій.

Має 2 дочірні класи – Sum, Multiplication, для виконання арифметичних операцій.

1. **Sum**

Успадковує клас Composite. Клас представляє функцію додавання, має свої методи для роботи з функціями. Об’єкт класу складається з суми інших функцій.

1. **Multiplication**

Успадковує клас Composite. Клас представляє функцію, яка є добутком кількох інших функцій, має свої методи для роботи з функціями.

1. **EFunction**

Клас редставляє експоненційну функцію, де основою є «e», а показником інша функція. Повертає значення експ. функції.

1. **Ln**

Клас представляє функцію натурального логарифма. Він обчислює значення натурального логарифма для результату внутрішньої функції.

1. **Exponential**

Повертає функцію у степіні.

1. **Const**

Клас представляє константну функцію, яка не має аргументів, зберігає значення константи і повертає це значення при обчисленні функції будь-якого «x».

1. **X**

Клас відповідає за представлення функції, яка повертає вхідне значення «x».

1. **Cos**

Клас відповідає за представлення функції косинуса та обчислення її значення для заданого внутрішнього виразу.

1. **Sin**

Клас відповідає за представлення функції синуса та обчислення її значення для заданого внутрішнього виразу.

1. **Abs**

Клас відповідає за представлення функції модулю та обчислення її значення для заданого внутрішнього виразу

**Function(interface):**

Містить 3 методи, що застосовуються у всіх класах, які реалізують цей інтерфейс:

double calculate(double x); //обчислює значення функції

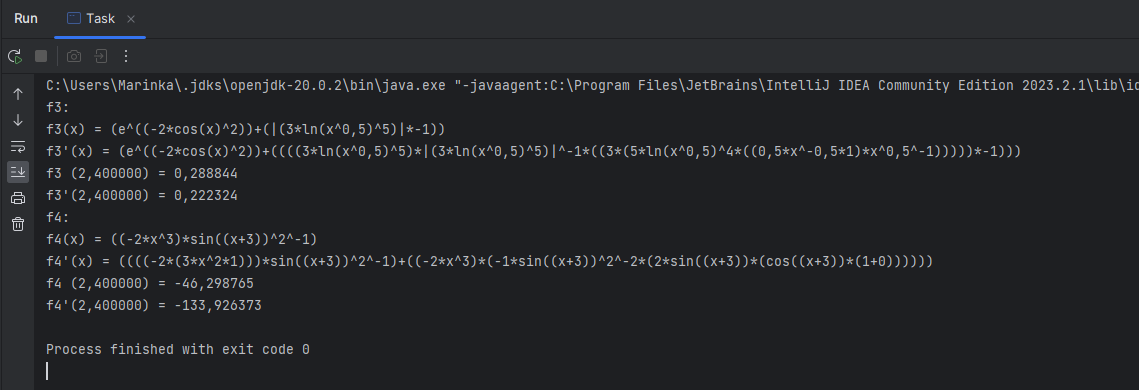
Function derivative(); //повертає похідну

String toPrettyString(NumberFormat nf); //повертає форматований рядок

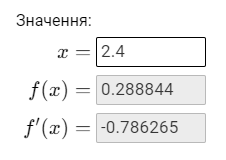
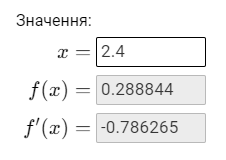
**Class Task:**

Тут виконується збірка виразів та обчислення результатів. Кожен вираз виконується рекурсивно(виконуються від зовнішніх до внутрішніх функцій).

1. ***Результати роботи програми та деталі її застосування.***



Перевірка виразів №1 та №2:



1. ***Висновки за результатами роботи.***

Під час виконання лабораторної роботи я ознайомилася з шаблоном проектування паттерну Composite для обчислення похідної довільної складеної функції.

1. ***Додатки (код програми).***

**Composite:**

import java.util.ArrayList; //для створення динамічних масивів  
import java.util.Arrays;  
  
public abstract class Composite implements Function {  
  
 private final ArrayList<Function> terms; //зберігає список функцій,які складають композитну ф-ю  
  
 public ArrayList<Function> terms() { //повертає список ф-й  
 return terms;  
 }  
  
 public Composite() {  
 terms = new ArrayList<>();  
 }  
  
 public Composite(Function... terms) { //приймає невизначену кіль-ть ф-й і ініціалізує списком, створеним з масиву  
 this.terms = new ArrayList<>(Arrays.*asList*(terms));  
 }  
  
 public Composite(ArrayList<Function> terms) {  
 this.terms = terms;  
 }  
}

**Sum:**

import java.text.NumberFormat;  
import java.util.ArrayList;  
import java.util.StringJoiner; //використовується для об'єднання рядків з роздільником  
  
public class Sum extends Composite {  
  
 public Sum() {  
 super();  
 }  
 public Sum(Function... terms) { //приймає змінну кіл-ь ф-й  
 super(terms);  
 }  
 public Sum(ArrayList<Function> terms) { //приймає список функцій  
 super(terms);  
 }  
 @Override  
 public double calculate(double x) {  
 double result = 0.0;  
 for (Function function : terms()) { //foreach  
 result += function.calculate(x);  
 }  
 return result;  
 }  
 @Override  
 public Function derivative() { //обчислює похідну для кожної ф-ії у списку  
 final ArrayList<Function> derivativeTerms = new ArrayList<>(terms().size());  
 for (Function function : terms()) {  
 derivativeTerms.add(function.derivative());  
 }  
 return new Sum(derivativeTerms);  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 final StringJoiner joiner = new StringJoiner("+");  
 for (Function function : terms()) {  
 joiner.add(function.toPrettyString(nf));  
 }  
 return String.*format*("(%s)",  
 joiner.toString()).replace("+-", "-");  
 }  
 public static Sum of(Function... terms) { //фабричний метод  
 return new Sum(terms);  
 }  
}

**Multiplication:**

import java.text.NumberFormat;  
import java.util.ArrayList;  
import java.util.StringJoiner; //використовується для об'єднання рядків з роздільником  
  
public class Multiplication extends Composite{  
 public Multiplication() {  
 super();  
 }  
 public Multiplication(Function... terms) { //приймає змінну кіл-ь ф-й  
 super(terms);  
 }  
 public Multiplication(ArrayList<Function> terms) {  
 super(terms);  
 }  
 @Override  
 public double calculate(double x) {  
 double result = 1;  
 for (Function function : terms()) {  
 result \*= function.calculate(x);  
 }  
 return result; //повертає добуток значень  
 }  
 @Override  
 public Function derivative() { //обчислює похідну  
 int counter = 0;  
 final ArrayList<Function> derivativeTerms = new ArrayList<>(terms().size()); //створення списку, куди додаватиметься частинний добуток після обчисл похідної  
 for (int i = 0; i < terms().size(); i++) {  
 ArrayList<Function> temp = new ArrayList<>(terms().size());  
 int j = 0; //індекс елемента списку  
 boolean isZero = false;  
 for (Function function : terms()) {  
 if(counter == j) temp.add(function.derivative()); //в temp додається похідна,якщо індекс дорівнює counter  
 else temp.add(function);  
 if (temp.get(j++) == Const.*ZERO*) isZero = true; //перевірка чи дорівнює поточний елемент нульовій константі  
 }  
 if(!isZero) derivativeTerms.add(Multiplication.*of*(temp)); //якщо не містить 0 константи, то додається у список derivativeTerms  
 counter++;  
  
 }  
  
 return new Sum(derivativeTerms); //gповертає суму частинни добутків  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 final StringJoiner joiner = new StringJoiner("\*"); //об'єднує рядки з роздільником "\*"  
 for (Function function : terms()) {  
 joiner.add(function.toPrettyString(nf)); //для кожної ф-ії викликається метод і перетворює у рядок  
 }  
 return String.*format*("(%s)",joiner.toString()); //повертається загальний рядок  
 }  
 public static Multiplication of(Function... terms) { //ств новий об'єкт класу з функціями як аргументи  
 return new Multiplication(terms);  
 }  
  
 public static Multiplication of(ArrayList<Function> terms) { //ств новий об'єкт класу з функціями в списку  
 return new Multiplication(terms);  
 }  
}

**EFunction:**

import java.text.NumberFormat;  
import static java.lang.Math.\*;  
public class EFunction implements Function {  
 private final Function exponent;  
  
 public EFunction(Function exponent) {  
 this.exponent = exponent;  
 }  
  
 @Override  
 public double calculate(double x) {  
 double exponentValue = exponent.calculate(x);  
 return *exp*(exponentValue);  
 }  
  
 @Override  
 public Function derivative() {  
 // Похідна експоненційної функції exp(x) є сама собою  
 return new EFunction(exponent);  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return "e^(" + exponent.toPrettyString(nf) + ")";  
 }  
  
 public static EFunction of(Function exponent) {  
 return new EFunction(exponent);  
 }  
}

**Ln:**

import java.text.NumberFormat;  
import static java.lang.Math.\*;  
public class Ln implements Function{  
  
 private final Function InnerF;  
  
 public Ln(Function InnerF) {  
 this.InnerF = InnerF;  
 }  
  
 @Override  
 public double calculate(double x) {  
 double InnerResult = InnerF.calculate(x);  
 if (InnerResult<=0){  
 return Double.*NaN*; // логарифм від від'ємного числа або нуля недійсний  
 }  
 return *log*(InnerResult); //обчислення натурального логарифму  
 }  
  
 @Override  
 public Function derivative() {  
 return new Multiplication(InnerF.derivative(), new Exponential(InnerF, -1));  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return "ln(" + InnerF.toPrettyString(nf) + ")";  
 }  
  
 public static Ln of(Function InnerF) { //фабричний метод  
 return new Ln(InnerF);  
 }  
}

**Exponential:**

import java.text.NumberFormat;//клас для форматування чисел у вигляді рядків  
  
  
import static java.lang.Math.\*;  
  
  
public class Exponential implements Function {  
  
 private final double power; //зберігає показник степеня  
 private final Function InnerF; //зберігає внутрішню функцію, яка буде пвднесена до степеня power  
  
 public Exponential(Function InnerF, double power) {  
 this.power = power;  
 this.InnerF = InnerF;  
 }  
 @Override  
 public double calculate(double x) {  
 return *pow*(InnerF.calculate(x),power);  
 }  
 @Override  
 public Function derivative() {  
 if (power - 1 == 1) return Multiplication.*of*(Const.*of*(power), InnerF, InnerF.derivative());  
 if (power - 1 == 0) return InnerF.derivative();  
 return Multiplication.*of*(Const.*of*(power), Exponential.*of*(InnerF, power-1), InnerF.derivative());  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return String.*format*("%s^%s", InnerF.toPrettyString(nf), nf.format(power));  
 }  
 public static Exponential of(Function InnerF, double power) {  
 return new Exponential(InnerF, power);  
 }  
}

**Const:**

import java.text.NumberFormat;  
  
public class Const implements Function {  
 public static final Const *ZERO* = new Const(0);  
  
 private final double value;  
  
 public Const(double value) {  
  
 this.value = value;  
 }  
 @Override  
 public double calculate(double x) {  
 return value;  
 }  
 @Override  
 public Function derivative() {  
 return *ZERO*;  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return nf.format(value);  
 }  
  
 public static Const of(double value) { //фабричний метод, який повертає новий екземпляр класу зі значенням  
 return new Const(value);  
 }  
}

**X:**

import java.text.NumberFormat; //клас для форматування чисел у вигляді рядків  
  
public class X implements Function {  
  
 public X() {  
  
 }  
 @Override  
 public double calculate(double x) {  
 return x;  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return "x";  
 }  
 @Override  
 public Function derivative() {  
 return new Const(1);  
 }  
  
 public static X x() { //дозволяє створювати об'єкти без необхідності явно викликати конструктор  
 return new X();  
 }  
}

**Cos:**

import java.text.NumberFormat;  
import static java.lang.Math.\*;  
  
public class Cos implements Function {  
  
 private final Function InnerF; //зберігає внутрішню ф-ю, яка буде виокристовуватися як аргумент для ф-ії косинуса  
  
 public Cos(Function InnerF) {  
 this.InnerF = InnerF;  
 }  
 @Override  
 public double calculate(double x) {  
 return *cos*(InnerF.calculate(x)); //обчисл InnerF для x і повертає косинус цього значення  
 }  
 @Override  
 public Function derivative() {  
 return Multiplication.*of*(Const.*of*(-1), Sin.*of*(InnerF), InnerF.derivative());  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return String.*format*("cos(%s)", InnerF.toPrettyString(nf));  
 }  
  
 public static Cos of(Function InnerF) {  
 return new Cos(InnerF);  
 }  
}

**Sin:**

import java.text.NumberFormat; //клас для форматування чисел у вигляді рядків  
import static java.lang.Math.\*;  
  
public class Sin implements Function {  
  
 private final Function InnerF;  
  
 public Sin(Function InnerF) {  
 this.InnerF = InnerF;  
 }  
 @Override  
 public double calculate(double x) {  
 return *sin*(InnerF.calculate(x));  
 }  
 @Override  
 public Function derivative() {  
 return Multiplication.*of*(Cos.*of*(InnerF), InnerF.derivative());  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return String.*format*("sin(%s)", InnerF.toPrettyString(nf));  
 }  
  
 public static Sin of(Function InnerF) {  
 return new Sin(InnerF);  
 }  
}

**Abs:**

import java.text.NumberFormat;  
import static java.lang.Math.\*;  
  
public class Abs implements Function {  
  
 private final Function InnerF; //внутрішня функція, для якої обчислюється модуль  
  
 public Abs(Function InnerF) {  
 this.InnerF = InnerF;  
 }  
 @Override  
 public double calculate(double x) {  
 return *abs*(InnerF.calculate(x)); //повертає модуль значення, обчисленого InnerF для x  
 }  
 @Override  
 public Function derivative() { //обчислення похідної  
  
 return Multiplication.*of*(InnerF, Exponential.*of*(Abs.*of*(InnerF), -1), InnerF.derivative());  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return String.*format*("|%s|", InnerF.toPrettyString(nf));  
 }  
 public static Abs of(Function InnerF) { //фабричний метод  
 return new Abs(InnerF);  
 }  
}