МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

КАФЕДРА «ПРОГРАМНОЇ ІНЖЕНЕРІЇ ТА ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ ІМЕНІ А. В. ДАБАГЯНА»

ЗВІТ

З лабораторної роботи №5

«РЕФЛЕКСІЯ. СТВОРЕННЯ ЗАСТОСУНКІВ БАЗ ДАНИХ»

ВИКОНАВ:

студент групи КН-422ч

Максим БЕЛОШИЦЬКИЙ

ПЕРЕВІРИВ:

Старший викладач каф. ПІІТУ

Лев ІВАНОВ

Харків – 2023

**ЗМІСТ**

[МЕТА 3](#_Toc168788588)

[Завдання 1.1 4](#_Toc168788589)

[Код завдання 1.1: 4](#_Toc168788590)

[FunctionPloter.java 4](#_Toc168788591)

[JavaSourceFromString.java 12](#_Toc168788592)

[Тестування за допомогою JUnit: 13](#_Toc168788593)

[FunctionPloterTest.java 13](#_Toc168788594)

[Результати виконання програми 14](#_Toc168788595)

[Завдання 1.2 16](#_Toc168788596)

[Код завдання 1.2: 16](#_Toc168788597)

[PrimeFactorsApp.java 16](#_Toc168788598)

[Тестування за допомогою JUnit: 20](#_Toc168788599)

[PrimeFactorsAppTest.java 20](#_Toc168788600)

[Результати виконання програми 21](#_Toc168788601)

[Завдання 1.3 24](#_Toc168788602)

[Код завдання 1.3: 24](#_Toc168788603)

[BlockingQueueExample.java 24](#_Toc168788604)

[Тестування за допомогою JUnit: 26](#_Toc168788605)

[BlockingQueueExampleTest.java 26](#_Toc168788606)

[Результати виконання програм 28](#_Toc168788607)

[Завдання 1.4 30](#_Toc168788608)

[Код завдання 1.4: 30](#_Toc168788609)

[MathExpressionInterpreture.java 30](#_Toc168788610)

[Тестування за допомогою JUnit: 32](#_Toc168788611)

[MathExpressionInterpretureTest.java 32](#_Toc168788612)

[Результати виконання програми 33](#_Toc168788613)

[Завдання 1.5 36](#_Toc168788614)

[Код завдання 1.5: 36](#_Toc168788615)

[PiCalculator.java 36](#_Toc168788616)

[Тестування за допомогою JUnit: 38](#_Toc168788617)

[PiCalculatorTest.java 38](#_Toc168788618)

[Результати виконання програми 39](#_Toc168788619)

[Завдання для самоконтролю 42](#_Toc168788620)

[Код завдання 2.1: 42](#_Toc168788621)

[Main.java 42](#_Toc168788622)

[Counter.java 42](#_Toc168788623)

[Worker.java 43](#_Toc168788624)

[Результати виконання програми 43](#_Toc168788625)

[Код завдання 2.2: 44](#_Toc168788626)

[FunctionPlotter.java 44](#_Toc168788627)

[plot.html 44](#_Toc168788628)

[Результати виконання програми 45](#_Toc168788629)

[ВИСНОВОК 46](#_Toc168788630)

# **МЕТА**

Метою цієї лабораторної роботи є надання студентам практичних навичок у створенні програм з графічним інтерфейсом користувача, застосуванні багатопотокового програмування та використанні динамічної компіляції коду. Студенти мають змогу ознайомитися з JavaFX для розробки сучасних графічних інтерфейсів, вчаться обробляти користувацькі дані в реальному часі за допомогою динамічної компіляції, а також здобувають досвід роботи з потоками для виконання паралельних обчислень.

Ця робота також включає опанування технік інтерпретації математичних виразів та розрахунку математичних констант, що допомагає поглибити розуміння алгоритмів та математичних обчислень. В цілому, лабораторна робота спрямована на закріплення теоретичних знань та розвиток професійних навичок програмування, що є необхідними для розробки ефективних і функціональних програм.

# Завдання 1.1

Створити програму графічного інтерфейсу користувача, яка призначена для побудови графіку довільних функцій. Користувач повинен увести дійсні значення *a* і *b*, функції *f(x)* і *g(x)* у вигляді рядків, які відповідають синтаксису Java. У програмі здійснюється обчислення функції *h(x)* відповідно до індивідуального завдання:



Після введення необхідних функцій, діапазону відображення графіку і натиснення відповідної кнопки здійснюється побудова графіку. Слід також передбачити функцію очищення рядків уведення й графіку.

Для програмної реалізації обробки введених виразів слід застосувати динамічну компіляцію коду. Для створення застосунку графічного інтерфейсу користувача слід використати засоби JavaFX. Рекомендований підхід - використання компоненту LineChart.

## Код завдання 1.1:

### FunctionPloter.java

package *edu.lab05\_ind00*;

import *javafx.application.Application*;

import *javafx.application.Platform*;

import *javafx.embed.swing.SwingFXUtils*;

import *javafx.geometry.Insets*;

import *javafx.geometry.Pos*;

import *javafx.scene.Scene*;

import *javafx.scene.chart.LineChart*;

import *javafx.scene.chart.NumberAxis*;

import *javafx.scene.chart.XYChart*;

import *javafx.scene.control.Alert*;

import *javafx.scene.control.Button*;

import *javafx.scene.control.Label*;

import *javafx.scene.control.ProgressIndicator*;

import *javafx.scene.control.TextField*;

import *javafx.scene.image.WritableImage*;

import *javafx.scene.layout.HBox*;

import *javafx.scene.layout.VBox*;

import *javafx.stage.FileChooser*;

import *javafx.stage.Stage*;

import *javafx.concurrent.Task*;

import *javax.imageio.ImageIO*;

import *javax.tools.JavaCompiler*;

import *javax.tools.ToolProvider*;

import *javax.tools.DiagnosticCollector*;

import *javax.tools.JavaFileObject*;

import *java.io.File*;

import *java.io.IOException*;

import *java.lang.reflect.Method*;

import *java.net.URLClassLoader*;

import *java.util.List*;

import *java.util.ArrayList*;

import *java.util.Arrays*;

import *java.util.Collections*;

import *java.net.URL*;

*/\*\**

*\* This class represents a function plotter application.*

*\* It allows users to input function parameters and plot the graph of the function.*

*\*/*

*public* *class* FunctionPlotter *extends* *Application* {

*// Fields*

*private* *TextField* aField;

*private* *TextField* bField;

*private* *TextField* fField;

*private* *TextField* gField;

*private* *TextField* minXField;

*private* *TextField* maxXField;

*private* *LineChart*<*Number*, *Number*> lineChart;

*private* *ProgressIndicator* progressIndicator;

*/\*\**

*\* The entry point of the application.*

*\* @param args The command line arguments.*

*\*/*

*public* *static* *void* main(*String*[] *args*) {

        launch(args);

    }

*/\*\**

*\* Initializes and displays the user interface of the application.*

*\* @param primaryStage The primary stage for the application.*

*\*/*

    @*Override*

*public* *void* start(@*SuppressWarnings*("exports") *Stage* *primaryStage*) {

*// Set the title of the application window*

        primaryStage.setTitle("Function Plotter");

*// Create labels and text fields for input*

*Label* aLabel = new Label("a:");

        aField = new TextField();

*Label* bLabel = new Label("b:");

        bField = new TextField();

*Label* fLabel = new Label("f(x):");

        fField = new TextField();

*Label* gLabel = new Label("g(x):");

        gField = new TextField();

*Label* minXLabel = new Label("min x:");

        minXField = new TextField();

*Label* maxXLabel = new Label("max x:");

        maxXField = new TextField();

*// Create buttons for plotting, clearing, and saving the graph*

*Button* plotButton = new Button("Plot");

        plotButton.setOnAction(e -> plotGraph());

*Button* clearButton = new Button("Clear");

        clearButton.setOnAction(e -> clearFields());

*Button* saveButton = new Button("Save");

        saveButton.setOnAction(e -> saveGraph(primaryStage));

*// Create a progress indicator*

        progressIndicator = new ProgressIndicator();

        progressIndicator.setVisible(false);

*// Create a line chart for displaying the graph*

*NumberAxis* xAxis = new NumberAxis();

*NumberAxis* yAxis = new NumberAxis();

        lineChart = new *LineChart*<>(xAxis, yAxis);

*// Create vertical and horizontal boxes to organize the content*

*VBox* inputVBox = new VBox(10); *// Vertical box for input fields*

        inputVBox.getChildren().addAll(aLabel, aField, bLabel, bField, fLabel, fField, gLabel, gField, minXLabel, minXField, maxXLabel, maxXField);

*HBox* buttonHBox = new HBox(10); *// Horizontal box for buttons*

        buttonHBox.getChildren().addAll(plotButton, clearButton, saveButton);

*VBox* vbox = new VBox(20); *// Vertical box for all content*

        vbox.setAlignment(Pos.CENTER); *// Center align all content*

        vbox.setPadding(new Insets(20)); *// Add padding*

        vbox.getChildren().addAll(inputVBox, buttonHBox, progressIndicator, lineChart);

*// Create the scene and set it on the primary stage*

*Scene* scene = new Scene(vbox, 800, 600);

        primaryStage.setScene(scene);

        primaryStage.show();

    }

*/\*\**

*\* Plots the graph of the function based on the user input.*

*\*/*

*private* *void* plotGraph() {

        try {

*// Parse input values*

*double* a = Double.parseDouble(aField.getText());

*double* b = Double.parseDouble(bField.getText());

*String* fFunction = formatFunction(fField.getText());

*String* gFunction = formatFunction(gField.getText());

*double* minX = Double.parseDouble(minXField.getText());

*double* maxX = Double.parseDouble(maxXField.getText());

*// Validate input values*

            if (minX >= maxX) {

                throw new IllegalArgumentException("min x must be less than max x.");

            }

*// Create a background task for calculating the function values*

*Task*<*Void*> task = new *Task*<*Void*>() {

                @*Override*

*protected* *Void* call() *throws* *Exception* {

*// Generate x values*

*List*<*Double*> xValues = generateXValues(minX, maxX, 0.1);

*// Calculate h values based on the input functions and parameters*

*List*<*Double*> hValues = calculateHValues(xValues, a, b, fFunction, gFunction);

*// Create a series for the line chart and add data points*

*XYChart*.*Series*<*Number*, *Number*> series = new *XYChart*.*Series*<>();

                    series.setName("Function Plot");

                    for (*int* i = 0; i < xValues.size(); i++) {

*final* *int* index = i;

                        Platform.runLater(() -> series.getData().add(new *XYChart*.*Data*<>(xValues.get(index), hValues.get(index))));

                    }

*// Update the line chart and hide the progress indicator*

                    Platform.runLater(() -> {

                        lineChart.getData().clear();

                        lineChart.setCreateSymbols(false); *// Disable symbols*

                        lineChart.getData().add(series);

                        progressIndicator.setVisible(false);

                    });

                    return null;

                }

                @*Override*

*protected* *void* failed() {

*Throwable* exception = getException();

                    Platform.runLater(() -> {

                        showAlert("Execution Error", "An error occurred while calculating the function.");

                        exception.printStackTrace();

                        progressIndicator.setVisible(false);

                    });

                }

            };

*// Show the progress indicator and start the background task*

            progressIndicator.setVisible(true);

            new Thread(task).start();

        } catch (*NumberFormatException* *e*) {

            showAlert("Input Error", "Please enter valid numbers for a, b, min x, and max x.");

        } catch (*IllegalArgumentException* *e*) {

            showAlert("Function Error", e.getMessage());

        } catch (*Exception* *e*) {

            showAlert("Execution Error", "An error occurred while calculating the function.");

            e.printStackTrace();

        }

    }

*/\*\**

*\* Clears all input fields and the line chart.*

*\*/*

*private* *void* clearFields() {

        aField.clear();

        bField.clear();

        fField.clear();

        gField.clear();

        minXField.clear();

        maxXField.clear();

        lineChart.getData().clear();

        progressIndicator.setVisible(false);

    }

*/\*\**

*\* Saves the graph as a PNG image file.*

*\* @param stage The stage used to display the file chooser dialog.*

*\*/*

*private* *void* saveGraph(*Stage* *stage*) {

*FileChooser* fileChooser = new FileChooser();

        fileChooser.setTitle("Save Graph");

        fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("PNG files", "\*.png"));

*File* file = fileChooser.showSaveDialog(stage);

        if (file != null) {

*WritableImage* image = lineChart.snapshot(null, null);

            try {

                ImageIO.write(SwingFXUtils.fromFXImage(image, null), "png", file);

            } catch (*IOException* *e*) {

                showAlert("Save Error", "An error occurred while saving the graph.");

                e.printStackTrace();

            }

        }

    }

*/\*\**

*\* Generates a list of x values within the specified range and step size.*

*\* @param start The start value of x.*

*\* @param end The end value of x.*

*\* @param step The step size between x values.*

*\* @return A list of x values.*

*\*/*

*public* *List*<*Double*> generateXValues(*double* *start*, *double* *end*, *double* *step*) {

*List*<*Double*> xValues = new *ArrayList*<>();

        for (*double* x = start; x <= end; x += step) {

            xValues.add(x);

        }

        return xValues;

    }

*/\*\**

*\* Calculates the h values based on the input x values, functions, and parameters.*

*\* @param xValues The list of x values.*

*\* @param a The parameter a.*

*\* @param b The parameter b.*

*\* @param fFunction The function f(x).*

*\* @param gFunction The function g(x).*

*\* @return A list of h values.*

*\* @throws Exception If an error occurs during function calculation.*

*\*/*

*public* *List*<*Double*> calculateHValues(*List*<*Double*> *xValues*, *double* *a*, *double* *b*, *String* *fFunction*, *String* *gFunction*) *throws* *Exception* {

*List*<*Double*> hValues = new *ArrayList*<>();

*// Compile the function classes dynamically*

*Class*<*?*> fClass = compileFunction("edu.lab05\_ind00", "F", fFunction);

*Class*<*?*> gClass = compileFunction("edu.lab05\_ind00", "G", gFunction);

*// Get the apply method from the compiled function classes*

*Method* fMethod = fClass.getMethod("apply", double.class);

*Method* gMethod = gClass.getMethod("apply", double.class);

*// Calculate h values for each x value*

        for (*double* x : xValues) {

*double* fValue = (*double*) fMethod.invoke(null, x);

*double* gValue = (*double*) gMethod.invoke(null, x);

            hValues.add(a \* fValue - b \* gValue);

        }

        return hValues;

    }

*/\*\**

*\* Compiles a function class dynamically from the provided function body.*

*\* @param packageName The package name for the function class.*

*\* @param className The class name for the function class.*

*\* @param functionBody The body of the function.*

*\* @return The compiled function class.*

*\* @throws Exception If an error occurs during compilation.*

*\*/*

*public* *Class*<*?*> compileFunction(*String* *packageName*, *String* *className*, *String* *functionBody*) *throws* *Exception* {

*// Construct the source code for the function class*

*String* sourceCode = "package " + packageName + ";\n\npublic class " + className + " { public static double apply(double x) { return " + functionBody + "; } }";

        System.out.println("Source code: " + sourceCode); *// Debugging line*

*// Get the system Java compiler*

*JavaCompiler* compiler = ToolProvider.getSystemJavaCompiler();

        if (compiler == null) {

            throw new IllegalStateException("Java Compiler is not available. Ensure you are using a JDK and not a JRE.");

        }

*// Specify the output directory for compiled classes*

*String* outputDirectory = "target/classes";

*List*<*String*> options = new *ArrayList*<>();

        options.addAll(Arrays.asList("-d", outputDirectory));

*// Create a diagnostic collector for compilation errors*

*DiagnosticCollector*<*JavaFileObject*> diagnostics = new *DiagnosticCollector*<>();

*// Create a custom JavaFileObject from the source code*

*JavaSourceFromString* javaSource = new JavaSourceFromString(packageName + "." + className, sourceCode);

*// Compile the function class*

*boolean* success = compiler.getTask(null, null, diagnostics, options, null, Collections.singletonList(javaSource)).call();

        if (!success) {

*// Collect and display compilation error messages*

*StringBuilder* errorMsg = new StringBuilder();

            diagnostics.getDiagnostics().forEach(diagnostic -> errorMsg.append(diagnostic.getMessage(null)).append("\n"));

            throw new IllegalArgumentException("Compilation failed: \n" + errorMsg.toString());

        }

*// Load the compiled class from the specified output directory*

        @*SuppressWarnings*("resource")

*URLClassLoader* classLoader = new URLClassLoader(new *URL*[]{new File(outputDirectory).toURI().toURL()});

        return classLoader.loadClass(packageName + "." + className);

    }

*/\*\**

*\* Formats the function string by adding necessary prefixes and closures.*

*\* @param function The function string to format.*

*\* @return The formatted function string.*

*\*/*

*public* *String* formatFunction(*String* *function*) {

        function = function.replaceAll("\\s", ""); *// Remove all whitespaces*

        function = function.replaceAll("(?<!Math\\.)(sin|cos|tan|log|sqrt|abs|pow|exp|asin|acos|atan)", "Math.$1"); *// Add Math. prefix to functions*

        function = function.replaceAll("(\\d+|x|e|PI)\\^(\\d+|x|e|PI)", "Math.pow($1,$2)"); *// Convert `x^2` to `Math.pow(x,2)`*

*// Ensure proper closure of parentheses for power functions*

*int* openParenCount = function.length() - function.replace("(", "").length();

*int* closeParenCount = function.length() - function.replace(")", "").length();

        if (openParenCount > closeParenCount) {

            for (*int* i = 0; i < openParenCount - closeParenCount; i++) {

                function += ")";

            }

        }

        return function;

    }

*private* *void* showAlert(*String* *title*, *String* *message*) {

*Alert* alert = new Alert(Alert.AlertType.ERROR);

        alert.setTitle(title);

        alert.setHeaderText(null);

        alert.setContentText(message);

        alert.showAndWait();

    }

}

### JavaSourceFromString.java

package *edu.lab05\_ind00*;

import *javax.tools.SimpleJavaFileObject*;

import *java.net.URI*;

*/\*\**

*\* A custom implementation of the SimpleJavaFileObject class that represents Java source code as a string.*

*\*/*

*public* *class* JavaSourceFromString *extends* *SimpleJavaFileObject* {

*private* *final* *String* code;

*/\*\**

*\* Constructs a new JavaSourceFromString object with the specified name and code.*

*\**

*\* @param name The name of the Java source file.*

*\* @param code The source code represented as a string.*

*\*/*

*protected* JavaSourceFromString(*String* *name*, *String* *code*) {

*super*(URI.create("string:///" + name.replace('.', '/') + Kind.SOURCE.extension), Kind.SOURCE);

*this*.code = code;

    }

*/\*\**

*\* Retrieves the source code content of this JavaSourceFromString object.*

*\**

*\* @param ignoreEncodingErrors Indicates whether encoding errors should be ignored.*

*\* @return The source code content as a CharSequence.*

*\*/*

    @*Override*

*public* *CharSequence* getCharContent(*boolean* *ignoreEncodingErrors*) {

        return code;

    }

}

## Тестування за допомогою JUnit:

### FunctionPloterTest.java

package *edu.lab05\_ind00*;

import *static* *org.junit.jupiter.api.Assertions.\**;

import *org.junit.jupiter.api.Test*;

import *java.util.List*;

*public* *class* FunctionPlotterTest {

*private* *final* *FunctionPlotter* functionPlotter = new FunctionPlotter();

    @*Test*

*public* *void* testFormatFunction() {

        assertEquals("Math.sin(x)", functionPlotter.formatFunction("sin(x)"));

        assertEquals("Math.pow(x,2)", functionPlotter.formatFunction("x^2"));

        assertEquals("Math.sin(x)\*Math.cos(x)", functionPlotter.formatFunction("sin(x) \* cos(x)"));

    }

    @*Test*

*public* *void* testGenerateXValues() {

*List*<*Double*> xValues = functionPlotter.generateXValues(0, 1, 0.5);

        assertEquals(3, xValues.size());

        assertEquals(0.0, xValues.get(0));

        assertEquals(0.5, xValues.get(1));

        assertEquals(1.0, xValues.get(2));

    }

}

## Результати виконання програми

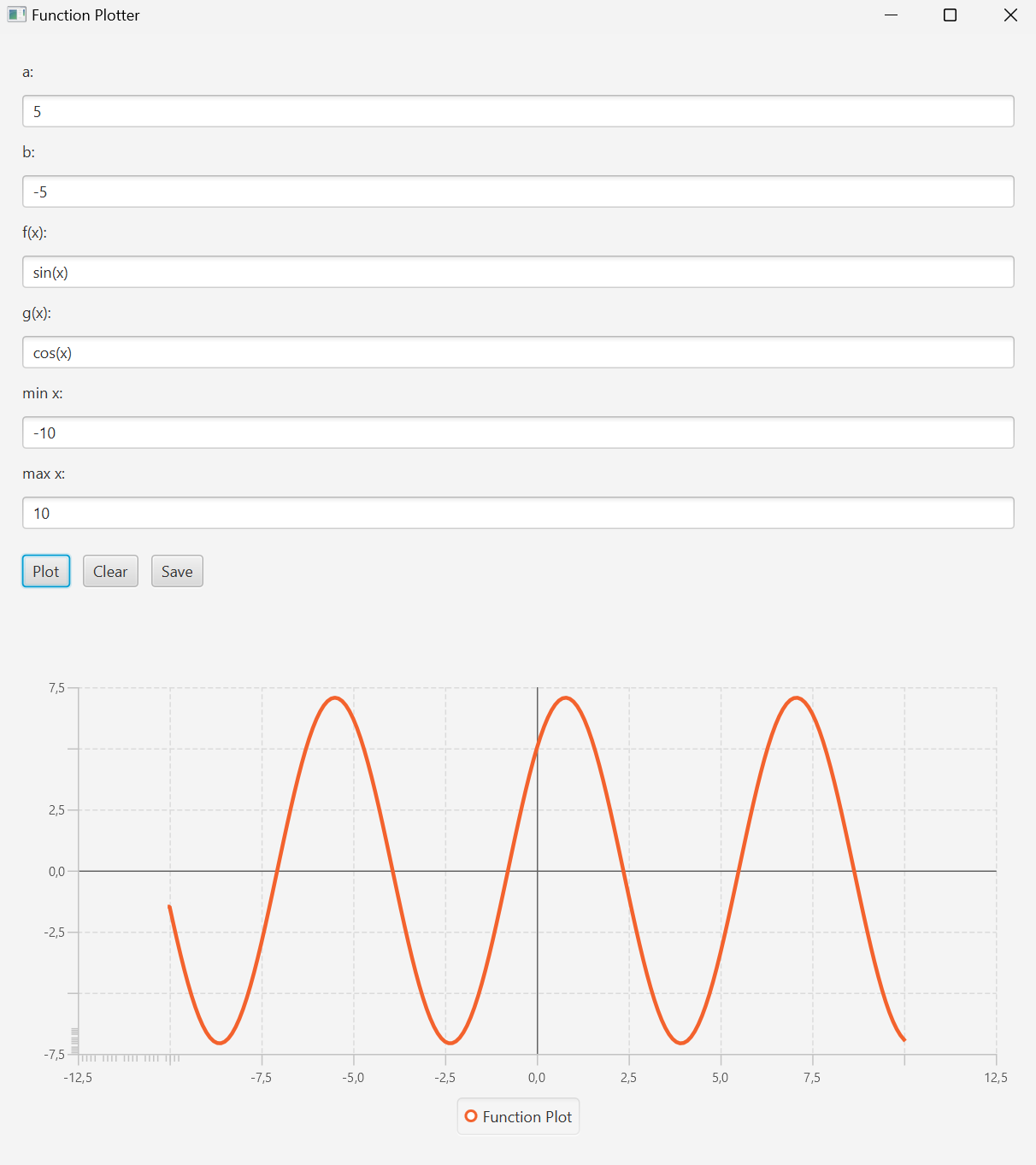


Рисунок 1.1 – «Створення графіку заданої функції»

У цьому завданні було розроблено Java-додаток з графічним інтерфейсом користувача (GUI), який дозволяє користувачам будувати графіки довільних функцій. Використовуючи JavaFX, додаток надає можливість вводити параметри функцій, обчислювати їх значення та відображати графіки. Ось ключові компоненти та етапи реалізації:

1. Інтерфейс користувача:
   * Створено основне вікно додатка з полями введення для значень a, b, f(x), g(x), а також діапазону min x та max x.
   * Додано кнопки для побудови графіка, очищення полів введення та збереження графіка у вигляді зображення.
   * Використано LineChart для відображення графіка функцій.
   * Додано індикатор прогресу, який відображається під час обчислення функцій.
2. Обробка введених даних:
   * При натисканні кнопки "Plot" (Побудувати графік) додаток читає введені користувачем значення та функції.
   * Перевіряється коректність введених даних (перетворення рядків у числа, перевірка діапазону).
3. Динамічна компіляція коду:
   * Використовується JavaCompiler для динамічної компіляції Java-коду функцій f(x) та g(x), введених користувачем.
   * Створено клас JavaSourceFromString, який дозволяє представляти Java-джерело коду у вигляді рядка.
   * Компільований код завантажується у додаток за допомогою URLClassLoader.
4. Обчислення значень функцій:
   * На основі введених користувачем функцій f(x) та g(x), а також параметрів a та b, обчислюються значення функції h(x) для кожного значення x у заданому діапазоні.
   * Результати обчислень відображаються на графіку.
5. Додаткові функції:
   * Очищення всіх полів введення та графіка.
   * Збереження графіка у файл у форматі PNG.

Основна мета цього завдання полягала у наданні практичних навичок роботи з JavaFX для створення графічних інтерфейсів, динамічної компіляції та виконання Java-коду, а також реалізації математичних обчислень та відображення результатів у вигляді графіків.

# Завдання 1.2

За допомогою засобів JavaFX розробити застосунок графічного інтерфейсу користувача, в якому користувач вводить діапазон чисел (від і до), а у вікні відображаються числа і їх прості множники. Реалізувати можливість призупинення, відновлення потоку, а також повного припинення і повторного обчислення з новими даними.

## Код завдання 1.2:

### PrimeFactorsApp.java

package *edu.lab05\_t00*;

import *javafx.application.Application*;

import *javafx.application.Platform*;

import *javafx.geometry.Insets*;

import *javafx.scene.Scene*;

import *javafx.scene.control.\**;

import *javafx.scene.layout.HBox*;

import *javafx.scene.layout.VBox*;

import *javafx.stage.Stage*;

import *java.util.ArrayList*;

import *java.util.List*;

import *java.util.concurrent.ExecutorService*;

import *java.util.concurrent.Executors*;

import *java.util.concurrent.TimeUnit*;

*public* *class* PrimeFactorsApp *extends* *Application* {

*private* *TextField* startField;

*private* *TextField* endField;

*private* *TextArea* resultArea;

*private* *Button* startButton;

*private* *Button* pauseButton;

*private* *Button* resumeButton;

*private* *Button* stopButton;

*private* *ExecutorService* executorService;

*private* *volatile* *boolean* running = true;

*private* *volatile* *boolean* paused = false;

*public* *static* *void* main(*String*[] *args*) {

        launch(args);

    }

    @*Override*

*public* *void* start(@*SuppressWarnings*("exports") *Stage* *primaryStage*) {

        primaryStage.setTitle("Prime Factors Finder");

        startField = new TextField();

        startField.setPromptText("Start of range");

        endField = new TextField();

        endField.setPromptText("End of range");

        resultArea = new TextArea();

        resultArea.setEditable(false);

        startButton = new Button("Start");

        pauseButton = new Button("Pause");

        resumeButton = new Button("Resume");

        stopButton = new Button("Stop");

        pauseButton.setDisable(true);

        resumeButton.setDisable(true);

        stopButton.setDisable(true);

        startButton.setOnAction(e -> startCalculation());

        pauseButton.setOnAction(e -> pauseCalculation());

        resumeButton.setOnAction(e -> resumeCalculation());

        stopButton.setOnAction(e -> stopCalculation());

*// Create HBox for the input fields with margin*

*HBox* inputBox = new HBox(10, startField, endField);

        VBox.setMargin(inputBox, new Insets(10));

*// Create HBox for the buttons with margin*

*HBox* buttonBox = new HBox(10, startButton, pauseButton, resumeButton, stopButton);

        VBox.setMargin(buttonBox, new Insets(10));

*// Create VBox and add all elements*

*VBox* vbox = new VBox(10, inputBox, buttonBox, resultArea);

        vbox.setPadding(new Insets(20)); *// Add padding to the VBox*

*Scene* scene = new Scene(vbox, 400, 300);

        primaryStage.setScene(scene);

        primaryStage.show();

    }

*private* *void* startCalculation() {

*String* startText = startField.getText();

*String* endText = endField.getText();

        if (!isValidInput(startText) || !isValidInput(endText)) {

            showAlert("Invalid input", "Please enter valid integers for the range.");

            return;

        }

*int* start = Integer.parseInt(startText);

*int* end = Integer.parseInt(endText);

        if (start > end) {

            showAlert("Invalid range", "Start of range should be less than or equal to end of range.");

            return;

        }

        resultArea.clear();

        running = true;

        paused = false;

        pauseButton.setDisable(false);

        stopButton.setDisable(false);

        executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        for (*int* i = start; i <= end; i++) {

*final* *int* number = i;

            executorService.submit(() -> {

                if (!running) return;

*synchronized* (*this*) {

                    while (paused) {

                        try {

                            wait();

                        } catch (*InterruptedException* *e*) {

                            if (!running) return;

                        }

                    }

                }

*List*<*Integer*> factors = findPrimeFactors(number);

*String* result = number + ": " + factors.toString() + "\n";

                Platform.runLater(() -> resultArea.appendText(result));

                try {

                    TimeUnit.MILLISECONDS.sleep(100);  *// Add delay to allow pause button interaction*

                } catch (*InterruptedException* *e*) {

                    Thread.currentThread().interrupt();

                }

            });

        }

        executorService.shutdown();

    }

*private* *boolean* isValidInput(*String* *input*) {

        try {

            Integer.parseInt(input);

            return true;

        } catch (*NumberFormatException* *e*) {

            return false;

        }

    }

*private* *void* showAlert(*String* *title*, *String* *message*) {

*Alert* alert = new Alert(Alert.AlertType.ERROR);

        alert.setTitle(title);

        alert.setHeaderText(null);

        alert.setContentText(message);

        alert.showAndWait();

    }

*private* *void* pauseCalculation() {

        paused = true;

        pauseButton.setDisable(true);

        resumeButton.setDisable(false);

    }

*private* *synchronized* *void* resumeCalculation() {

        paused = false;

        resumeButton.setDisable(true);

        pauseButton.setDisable(false);

        notifyAll();

    }

*private* *void* stopCalculation() {

        running = false;

        if (executorService != null) {

            executorService.shutdownNow();

        }

        pauseButton.setDisable(true);

        resumeButton.setDisable(true);

        stopButton.setDisable(true);

    }

*public* *List*<*Integer*> findPrimeFactors(*int* *number*) {

*List*<*Integer*> factors = new *ArrayList*<>();

        while (number % 2 == 0) {

            factors.add(2);

            number /= 2;

        }

        for (*int* i = 3; i <= Math.sqrt(number); i += 2) {

            while (number % i == 0) {

                factors.add(i);

                number /= i;

            }

        }

        if (number > 2) {

            factors.add(number);

        }

        return factors;

    }

}

## Тестування за допомогою JUnit:

### PrimeFactorsAppTest.java

package *edu.lab05\_t00*;

import *org.junit.jupiter.api.Test*;

import *static* *org.junit.jupiter.api.Assertions.assertEquals*;

import *java.util.List*;

*public* *class* PrimeFactorsAppTest {

    @*Test*

*public* *void* testFindPrimeFactors() {

*PrimeFactorsApp* primeFactorsApp = new PrimeFactorsApp();

*// Test for a prime number*

*int* primeNumber = 13;

*List*<*Integer*> primeFactors = primeFactorsApp.findPrimeFactors(primeNumber);

        assertEquals(1, primeFactors.size());

        assertEquals(primeNumber, primeFactors.get(0));

*// Test for a non-prime number*

*int* nonPrimeNumber = 36;

*List*<*Integer*> nonPrimeFactors = primeFactorsApp.findPrimeFactors(nonPrimeNumber);

        assertEquals(4, nonPrimeFactors.size());

        assertEquals(2, nonPrimeFactors.get(0));

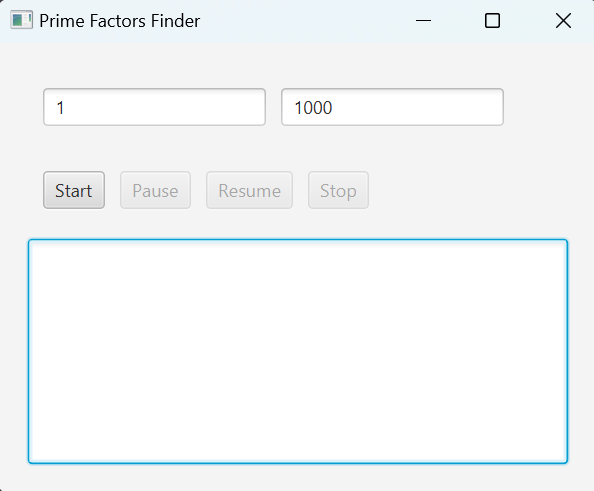
        assertEquals(2, nonPrimeFactors.get(1));

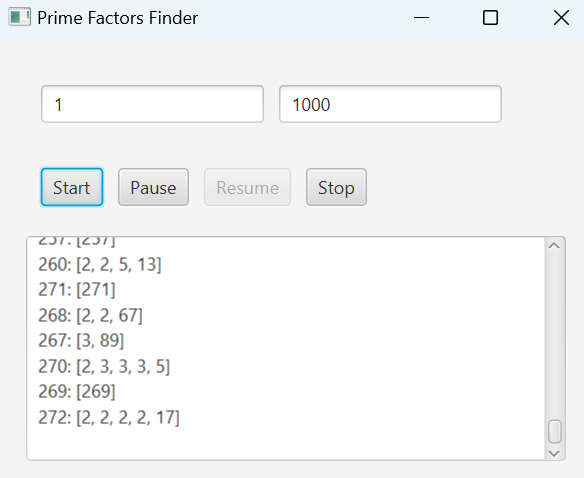
        assertEquals(3, nonPrimeFactors.get(2));

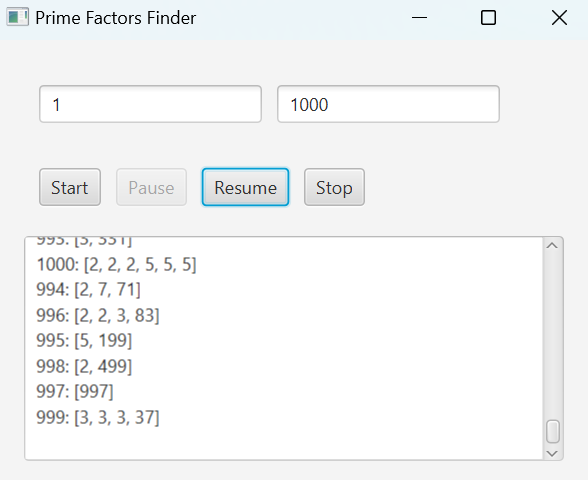
    }

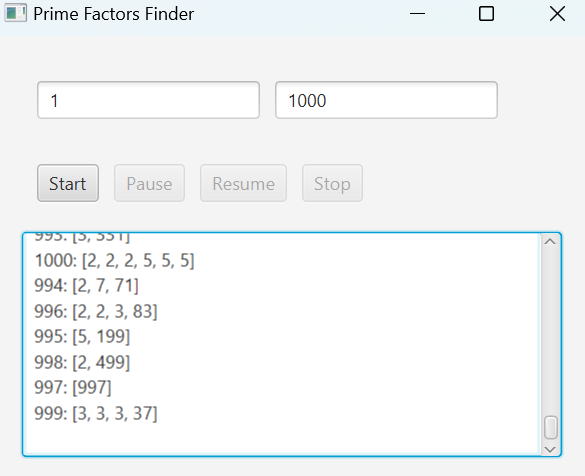
}

## Результати виконання програми









Рисуноки 1.2-1.6 – «Робота потоків, пауза та зупинка»

Використані технології та інструменти:

* JavaFX для створення графічного інтерфейсу користувача.
* Java Concurrency API для управління потоками та забезпечення асинхронної роботи.
* Java Collections Framework для зберігання та обробки даних.

Опис роботи програми

Основні компоненти інтерфейсу

* Текстові поля для введення діапазону чисел:
  + startField для введення початку діапазону.
  + endField для введення кінця діапазону.
* Текстова область (TextArea) для відображення результатів обчислень.
* Кнопки для керування процесом обчислень:
  + startButton для початку обчислень.
  + pauseButton для паузи обчислень.
  + resumeButton для відновлення обчислень після паузи.
  + stopButton для зупинки обчислень.

Основний процес роботи:

1. Інтерфейс користувача
   * Користувач вводить діапазон чисел у текстові поля startField та endField.
   * Натискає кнопку Start для початку обчислень.
   * Після цього активуються кнопки Pause та Stop, а кнопка Resume залишається неактивною до натискання кнопки Pause.
2. Обробка введених даних
   * Перевірка коректності введених значень (цілі числа, правильний діапазон).
   * Відображення повідомлень про помилки у випадку некоректного введення.
3. Процес обчислень
   * Для кожного числа з вказаного діапазону запускається окремий потік для обчислення простих множників.
   * Результати обчислень відображаються у текстовій області resultArea.
4. Контроль стану обчислень
   * Використання логічних змінних running та paused для контролю стану обчислень (виконуються, призупинені або зупинені).
   * Кнопка Pause дозволяє призупинити процес обчислень.
   * Кнопка Resume дозволяє відновити обчислення після паузи.
   * Кнопка Stop дозволяє повністю зупинити обчислення.
5. Використання багатопоточності
   * Використання ExecutorService для управління потоками та забезпечення асинхронної роботи.
   * Кожне число з діапазону обробляється в окремому потоці, що дозволяє користувачеві взаємодіяти з інтерфейсом під час виконання обчислень.

Застосунок для знаходження простих множників чисел демонструє можливості JavaFX для створення графічного інтерфейсу та використання багатопоточності для виконання обчислень у фоновому режимі. Це дозволяє користувачам ефективно взаємодіяти з програмою навіть під час тривалих обчислювальних процесів.

# Завдання 1.3

Створити консольну програму, в якій один потік виконання додає цілі числа до черги BlockingQueue,а інший обчислює їх середнє арифметичне.

## Код завдання 1.3:

### BlockingQueueExample.java

package *edu.lab05\_t01*;

import *java.util.concurrent.ArrayBlockingQueue*;

import *java.util.concurrent.BlockingQueue*;

*public* *class* BlockingQueueExample {

*private* *static* *final* *int* QUEUE\_CAPACITY = 10;

*private* *static* *final* *int* NUMBERS\_TO\_ADD = 20;

*public* *static* *void* main(*String*[] *args*) {

*BlockingQueue*<*Integer*> queue = new *ArrayBlockingQueue*<>(QUEUE\_CAPACITY);

*Thread* producerThread = new Thread(new NumberProducer(queue, NUMBERS\_TO\_ADD));

*Thread* consumerThread = new Thread(new AverageCalculator(queue));

        producerThread.start();

        consumerThread.start();

        try {

            producerThread.join();

            consumerThread.interrupt();

            consumerThread.join();

        } catch (*InterruptedException* *e*) {

            e.printStackTrace();

        }

    }

*static* *class* NumberProducer *implements* *Runnable* {

*private* *final* *BlockingQueue*<*Integer*> queue;

*private* *final* *int* numbersToAdd;

*public* NumberProducer(*BlockingQueue*<*Integer*> *queue*, *int* *numbersToAdd*) {

*this*.queue = queue;

*this*.numbersToAdd = numbersToAdd;

        }

        @*Override*

*public* *void* run() {

            for (*int* i = 1; i <= numbersToAdd; i++) {

                try {

                    queue.put(i);

                    System.out.println("Produced: " + i);

                    Thread.sleep(100);

                } catch (*InterruptedException* *e*) {

                    Thread.currentThread().interrupt();

                }

            }

        }

    }

*static* *class* AverageCalculator *implements* *Runnable* {

*private* *final* *BlockingQueue*<*Integer*> queue;

*public* AverageCalculator(*BlockingQueue*<*Integer*> *queue*) {

*this*.queue = queue;

        }

        @*Override*

*public* *void* run() {

            try {

*int* sum = 0;

*int* count = 0;

                while (!Thread.currentThread().isInterrupted()) {

*Integer* number = queue.take();

                    sum += number;

                    count++;

*double* average = (*double*) sum / count;

                    System.out.println("Consumed: " + number + ", Current Average: " + average);

                    System.out.println();

                }

            } catch (*InterruptedException* *e*) {

*// Thread interrupted, exit gracefully*

                System.out.println("Average calculation interrupted.");

            }

        }

    }

}

## Тестування за допомогою JUnit:

### BlockingQueueExampleTest.java

package *edu.lab05\_t01*;

import *org.junit.jupiter.api.Test*;

import *static* *org.junit.jupiter.api.Assertions.assertEquals*;

import *java.util.concurrent.ArrayBlockingQueue*;

import *java.util.concurrent.BlockingQueue*;

*public* *class* BlockingQueueExampleTest {

    @*Test*

*public* *void* testNumberProducer() *throws* *InterruptedException* {

*BlockingQueue*<*Integer*> queue = new *ArrayBlockingQueue*<>(10);

*int* numbersToAdd = 5;

*BlockingQueueExample*.*NumberProducer* producer = new BlockingQueueExample.NumberProducer(queue, numbersToAdd);

*Thread* producerThread = new Thread(producer);

        producerThread.start();

        producerThread.join();

*// Ensure all numbers are added to the queue*

        for (*int* i = 1; i <= numbersToAdd; i++) {

            assertEquals(i, queue.take().intValue());

        }

    }

}

## Результати виконання програм

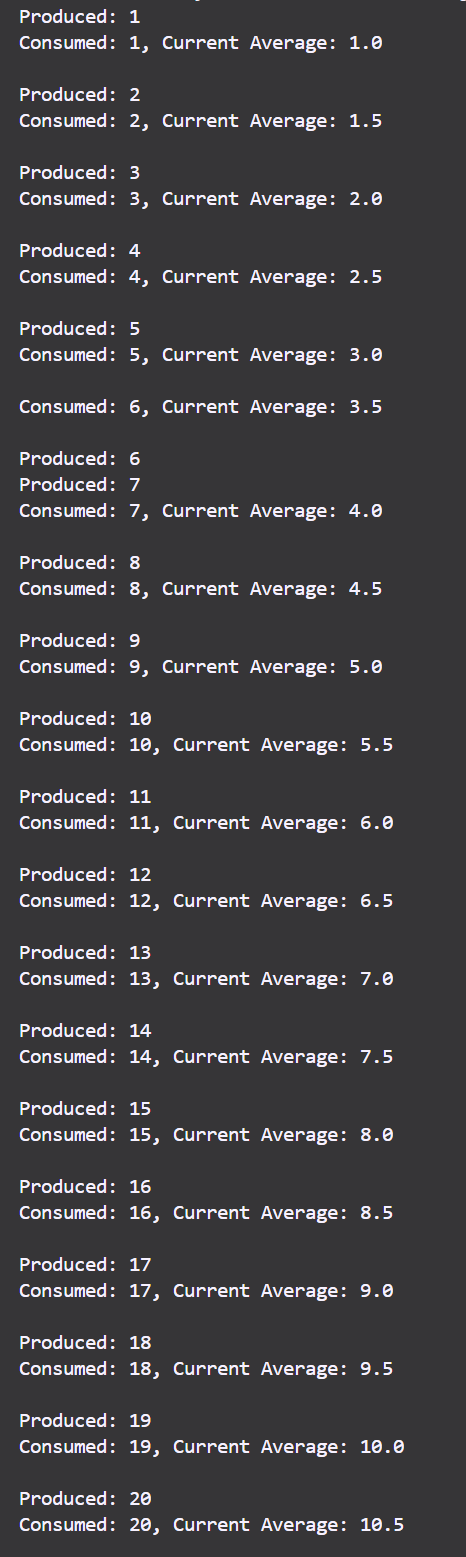


Рисунок 1.7 – «Обмін інформації за допомогою BlockingQueue»

Використані технології та інструменти:

* Java Concurrency API для реалізації багатопоточності.
* BlockingQueue для реалізації безпечного обміну даними між потоками.
* JUnit для тестування застосунку.

Опис роботи програми

Основні компоненти:

1. BlockingQueueExample - основний клас програми, що запускає потоки виробника та споживача.
2. NumberProducer - клас, що реалізує виробника чисел, який додає числа в чергу.
3. AverageCalculator - клас, що реалізує споживача, який обчислює середнє значення з чисел у черзі.

Опис класів:

1. BlockingQueueExample
   * Ініціалізує чергу з фіксованою ємністю (10 елементів).
   * Створює та запускає потоки виробника та споживача.
   * Чекає на завершення роботи виробника, після чого перериває роботу споживача.
2. NumberProducer
   * Реалізує інтерфейс Runnable.
   * Додає задану кількість чисел у чергу.
   * Використовує метод queue.put(i) для безпечного додавання елементів у чергу, що блокує потік у разі переповнення черги.
   * Виводить повідомлення про додавання кожного числа.
   * Додає затримку між додаванням чисел для демонстрації роботи.
3. AverageCalculator
   * Реалізує інтерфейс Runnable.
   * Виймає числа з черги та обчислює поточне середнє значення.
   * Використовує метод queue.take() для безпечного взяття елементів з черги, що блокує потік у разі порожньої черги.
   * Виводить повідомлення про поточне середнє значення після кожного взятого числа.
   * Реалізує обробку переривання потоку для коректного завершення роботи.

Застосунок демонструє ефективне використання BlockingQueue для синхронізації між потоками виробника та споживача. Програма наочно показує можливості Java Concurrency API для реалізації багатопотокових обчислень з безпечною передачею даних між потоками.

# Завдання 1.4

Створити консольний застосунок, який дозволяє вводити математичні вирази, обчислювати й виводити результат. Вираз може складатися з констант, математичних операцій і дужок. Для реалізації використовувати засоби інтерпретації скриптів JavaScript.

Примітка. Синтаксис математичних виразів JavaScript аналогічний Java. Результат можна виводити за допомогою функції print() без створення додаткових змінних.

## Код завдання 1.4:

### MathExpressionInterpreture.java

package *edu.lab05\_t02*;

import *org.graalvm.polyglot.Context*;

import *org.graalvm.polyglot.Value*;

import *java.util.HashMap*;

import *java.util.Map*;

import *java.util.Scanner*;

*public* *class* MathExpressionInterpreter {

*public* *static* *void* main(*String*[] *args*) {

*// Створюємо контекст GraalVM з ввімкненою підтримкою модуля polyglot*

        try (*Context* context = Context.newBuilder("js")

                .allowAllAccess(true)

                .build()) {

*// Мапа для зберігання змінних*

*Map*<*String*, *Object*> variables = new *HashMap*<>();

*Scanner* scanner = new Scanner(System.in);

            System.out.println("Введіть математичний вираз (або 'exit' для завершення):");

            while (true) {

                System.out.print("> ");

*String* input = scanner.nextLine();

                if (input.equalsIgnoreCase("exit")) {

                    break;

                }

                try {

*// Додаємо всі змінні до контексту перед обчисленням*

                    for (*Map*.*Entry*<*String*, *Object*> entry : variables.entrySet()) {

                        context.getBindings("js").putMember(entry.getKey(), entry.getValue());

                    }

*// Виконуємо введений вираз*

*Value* result = context.eval("js", input);

*// Зберігаємо результат у змінну, якщо задано ім'я змінної*

                    if (input.contains("=")) {

*String*[] parts = input.split("=");

                        if (parts.length == 2) {

*String* varName = parts[0].trim();

                            variables.put(varName, result);

                            System.out.println(varName + " = " + result);

                        }

                    } else {

                        System.out.println("Результат: " + result);

                    }

                } catch (*Exception* *e*) {

                    System.out.println("Помилка в виразі: " + e.getMessage());

                }

            }

            scanner.close();

        }

    }

}

## Тестування за допомогою JUnit:

### MathExpressionInterpretureTest.java

package *edu.lab05\_t02*;

import *org.junit.jupiter.api.AfterEach*;

import *org.junit.jupiter.api.BeforeEach*;

import *org.junit.jupiter.api.Test*;

import *java.io.ByteArrayInputStream*;

import *java.io.ByteArrayOutputStream*;

import *java.io.InputStream*;

import *java.io.PrintStream*;

import *static* *org.junit.jupiter.api.Assertions.assertEquals*;

*public* *class* MathExpressionInterpreterTest {

*private* *final* *InputStream* originalInput = System.in;

*private* *final* *PrintStream* originalOutput = System.out;

*private* *ByteArrayOutputStream* outputStream;

*private* *ByteArrayInputStream* inputStream;

    @*BeforeEach*

*public* *void* setUpStreams() {

        outputStream = new ByteArrayOutputStream();

        System.setOut(new PrintStream(outputStream));

    }

    @*AfterEach*

*public* *void* restoreStreams() {

        System.setIn(originalInput);

        System.setOut(originalOutput);

    }

    @*Test*

*public* *void* testMathExpressionInterpreter() {

*// Test a simple addition expression*

*String* input = "2 + 3";

*String* expectedOutput = "Результат: 5\n";

        executeTest(input, expectedOutput);

*// Test a more complex expression*

        input = "Math.pow(2, 3)";

        expectedOutput = "Результат: 8\n";

        executeTest(input, expectedOutput);

*// Test variable assignment and usage*

        input = "x = 5";

        expectedOutput = "x = 5\n";

        executeTest(input, expectedOutput);

        input = "x + 3";

        expectedOutput = "Результат: 8\n";

        executeTest(input, expectedOutput);

*// Test division by zero*

        input = "1 / 0";

        expectedOutput = "Помилка в виразі: Division by zero\n";

        executeTest(input, expectedOutput);

    }

*private* *void* executeTest(*String* *input*, *String* *expectedOutput*) {

        inputStream = new ByteArrayInputStream(input.getBytes());

        System.setIn(inputStream);

        MathExpressionInterpreter.main(new *String*[]{});

        assertEquals(expectedOutput, outputStream.toString());

    }

}

## Результати виконання програми

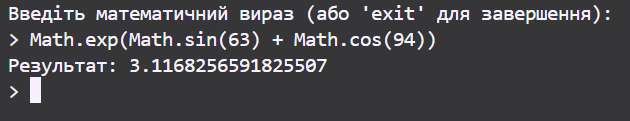


Рисунок 1.8 – «Приклад обчислень»

Використані технології та інструменти

* GraalVM Polyglot API для виконання скриптів на JavaScript.
* Java для реалізації основної логіки застосунку.

Опис роботи програми

Основні компоненти:

1. MathExpressionInterpreter - основний клас програми, що реалізує логіку інтерпретатора математичних виразів.

Опис класу:

1. MathExpressionInterpreter
   * Створює контекст GraalVM з ввімкненою підтримкою JavaScript.
   * Використовує мапу variables для зберігання змінних та їх значень.
   * Використовує Scanner для зчитування вводу користувача в інтерактивному режимі.
   * Підтримує безперервний ввід математичних виразів користувачем до введення команди exit.

Основні кроки роботи програми:

1. Ініціалізація контексту GraalVM
   * Контекст створюється з ввімкненою підтримкою JavaScript (js) та доступом до всіх ресурсів (allowAllAccess(true)).
2. Інтерактивний режим
   * Програма виводить запрошення для вводу математичного виразу.
   * Користувач вводить вираз або команду exit для завершення роботи.
3. Обробка вводу
   * Всі змінні з мапи variables додаються до контексту перед обчисленням.
   * Введений вираз виконується за допомогою методу context.eval("js", input).
   * Якщо вираз містить присвоєння змінної (через =), змінна та її значення зберігаються в мапу variables для подальшого використання.
   * Якщо вираз не містить присвоєння змінної, результат обчислення виводиться на екран.
4. Обробка помилок
   * Якщо введений вираз містить помилки, програма виводить відповідне повідомлення про помилку.

Застосунок демонструє можливості GraalVM для виконання JavaScript коду з використанням Java, забезпечуючи інтерактивний інтерпретатор математичних виразів. Програма дозволяє користувачам вводити та обчислювати вирази, а також зберігати та використовувати змінні для побудови складніших обчислень.

# Завдання 1.5

Реалізувати програму обчислення π с точністю до заданого ε як суму послідовності:



Обчислення здійснювати в окремому потоці виконання. Під час виконання обчислення надавати користувачеві можливість уводити запит про кількість обчислених елементів суми.

## Код завдання 1.5:

### PiCalculator.java

package *edu.lab05\_t03*;

import *java.util.Scanner*;

import *java.util.concurrent.atomic.AtomicBoolean*;

import *java.util.concurrent.atomic.AtomicInteger*;

import *java.util.concurrent.atomic.AtomicReference*;

*public* *class* PiCalculation {

*public* *static* *void* main(*String*[] *args*) {

*Scanner* scanner = new Scanner(System.in);

        System.out.print("Введіть значення ε для точності обчислення π: ");

*double* epsilon = scanner.nextDouble();

*PiCalculator* piCalculator = new PiCalculator(epsilon);

*Thread* calculationThread = new Thread(piCalculator);

        calculationThread.start();

        while (true) {

            System.out.print("Введіть 'status' для отримання поточного значення π, 'count' для кількості обчислених елементів або 'exit' для завершення: ");

*String* input = scanner.next();

            if (input.equalsIgnoreCase("exit")) {

                piCalculator.stopCalculation();

                break;

            } else if (input.equalsIgnoreCase("status")) {

                System.out.println("Поточне значення π: " + piCalculator.getCurrentPi());

            } else if (input.equalsIgnoreCase("count")) {

                System.out.println("Кількість обчислених елементів: " + piCalculator.getTermCount());

            }

        }

        try {

            calculationThread.join();

            System.out.println("Остаточне значення π: " + piCalculator.getCurrentPi());

            System.out.println("Кількість обчислених елементів: " + piCalculator.getTermCount());

        } catch (*InterruptedException* *e*) {

            System.out.println("Обчислювальний потік був перерваний.");

        }

        scanner.close();

    }

}

*class* PiCalculator *implements* *Runnable* {

*private* *final* *double* epsilon;

*private* *final* *AtomicReference*<*Double*> currentPi = new *AtomicReference*<>(0.0);

*private* *final* *AtomicBoolean* running = new AtomicBoolean(true);

*private* *final* *AtomicInteger* termCount = new AtomicInteger(0);

*public* PiCalculator(*double* *epsilon*) {

*this*.epsilon = epsilon;

    }

*public* *double* getCurrentPi() {

        return currentPi.get();

    }

*public* *int* getTermCount() {

        return termCount.get();

    }

*public* *void* stopCalculation() {

        running.set(false);

    }

    @*Override*

*public* *void* run() {

*double* pi = 0.0;

*double* term;

*int* i = 0;

        while (running.get()) {

            term = 4.0 \* Math.pow(-1, i) / (2 \* i + 1);

            pi += term;

            currentPi.set(pi);

            termCount.incrementAndGet();

            if (Math.abs(term) < epsilon) {

                running.set(false);

                break;

            }

            i++;

            try {

                Thread.sleep(100); *// Затримка для імітації довготривалих обчислень*

            } catch (*InterruptedException* *e*) {

                Thread.currentThread().interrupt();

                break;

            }

        }

    }

}

## Тестування за допомогою JUnit:

### PiCalculatorTest.java

package *edu.lab05\_t03*;

import *org.junit.jupiter.api.Test*;

import *static* *org.junit.jupiter.api.Assertions.assertEquals*;

*public* *class* PiCalculationTest {

    @*Test*

*public* *void* testPiCalculation() {

*double* epsilon = 0.0001; *// Set a small epsilon for precision*

*PiCalculator* piCalculator = new PiCalculator(epsilon);

*Thread* calculationThread = new Thread(piCalculator);

        calculationThread.start();

*// Wait for the calculation thread to complete*

        try {

            calculationThread.join();

        } catch (*InterruptedException* *e*) {

            e.printStackTrace();

        }

*// Retrieve the calculated value of pi and the number of terms computed*

*double* calculatedPi = piCalculator.getCurrentPi();

*int* termCount = piCalculator.getTermCount();

*// Verify the calculated value of pi using a known approximation*

*double* expectedPi = Math.PI;

        assertEquals(expectedPi, calculatedPi, epsilon, "Calculated pi does not match expected pi");

*// Verify that the number of terms computed is reasonable for the given epsilon*

*int* expectedTermCount = (*int*) (Math.PI / epsilon) + 1; *// Expected term count based on epsilon*

        assertEquals(expectedTermCount, termCount, "Number of terms computed is incorrect");

    }

}

## Результати виконання програми

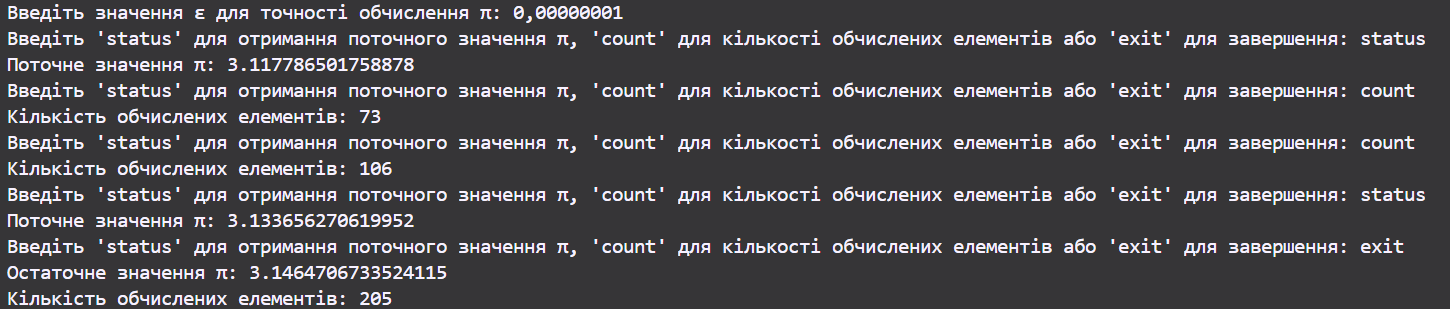


Рисунок 1.9 – «Обчислення числа π до заданого ε»

Опис роботи програми

Основні компоненти

* **PiCalculation** - головний клас, який запускає застосунок та керує введенням користувача.
* **PiCalculator** - клас, що реалізує інтерфейс Runnable та відповідає за обчислення числа π.

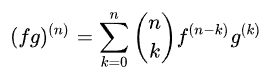
Опис класів:

**PiCalculation**

1. Відповідає за запуск основного потоку обчислень та обробку вводу користувача.
2. Створює екземпляр класу PiCalculator з заданою точністю ε.
3. Запускає обчислювальний потік.
4. Обробляє команди користувача:
   * + status - виводить поточне значення π.
     + count - виводить кількість обчислених елементів.
     + exit - зупиняє обчислення та завершує програму.

**PiCalculator**

1. Реалізує інтерфейс Runnable, що дозволяє запускати обчислення у новому потоці.
2. Використовує атомарні змінні AtomicReference<Double> currentPi, AtomicBoolean running та AtomicInteger termCount для забезпечення безпечної роботи в багатопоточному середовищі.
3. Реалізує метод run(), в якому обчислює значення π за формулою Лейбніца:



1. Виконує обчислення до досягнення заданої точності або до отримання команди на завершення.
2. Використовує затримку Thread.sleep(100) для імітації довготривалих обчислень та забезпечення можливості обробки команд користувача.

Основні кроки роботи програми:

1. **Ініціалізація та запуск**
2. Програма запитує у користувача точність ε для обчислення π.
3. Створює екземпляр PiCalculator з заданою точністю.
4. Запускає новий потік для виконання обчислень.

**Інтерактивний режим:**

1. Програма обробляє ввід користувача в циклі:

* status - виводить поточне значення π.
* count - виводить кількість обчислених елементів.
* exit - зупиняє обчислювальний потік та завершує програму.

**Зупинка та завершення:**

Після введення команди exit, обчислювальний потік зупиняється. Програма виводить остаточне значення π та кількість обчислених елементів.

Застосунок демонструє використання багатопоточності для обчислення числа π з заданою точністю. Програма дозволяє користувачам отримувати поточні результати обчислень та кількість обчислених елементів в реальному часі, що є корисним для довготривалих обчислень.

# Завдання для самоконтролю

Створити консольний застосунок, в якому здійснюється демонстрація можливостей синхронізації.

## Код завдання 2.1:

## Main.java

package *edu.lab05\_add00*;

*public* *class* Main {

*public* *static* *void* main(*String*[] *args*) {

*Counter* counter = new Counter();

*Thread* thread1 = new Thread(new Worker(counter));

*Thread* thread2 = new Thread(new Worker(counter));

*Thread* thread3 = new Thread(new Worker(counter));

        thread1.start();

        thread2.start();

        thread3.start();

        try {

            thread1.join();

            thread2.join();

            thread3.join();

        } catch (*InterruptedException* *e*) {

            e.printStackTrace();

        }

        System.out.println("Final count: " + counter.getCount());

    }

}

## Counter.java

package *edu.lab05\_add00*;

*// Counter.java*

*public* *class* Counter {

*private* *int* count = 0;

*public* *synchronized* *void* increment() {

        count++;

    }

*public* *int* getCount() {

        return count;

    }

}

## Worker.java

package *edu.lab05\_add00*;

*public* *class* Worker *implements* *Runnable* {

*private* *final* *Counter* counter;

*public* Worker(*Counter* *counter*) {

*this*.counter = counter;

    }

    @*Override*

*public* *void* run() {

        for (*int* i = 0; i < 1000; i++) {

            counter.increment();

        }

    }

}

## Результати виконання програми



Рисунок 2.1 – «Результат виконання»

Створити програму графічного інтерфейсу користувача, яка призначена для побудови графіку довільних функцій. Для реалізації використовувати засоби інтерпретації скриптів JavaScript.

## Код завдання 2.2:

## FunctionPlotter.java

package *edu.lab05\_add01*;

import *javafx.application.Application*;

import *javafx.scene.Scene*;

import *javafx.scene.web.WebEngine*;

import *javafx.scene.web.WebView*;

import *javafx.stage.Stage*;

*public* *class* FunctionPlotter *extends* *Application* {

    @*Override*

*public* *void* start(*Stage* *primaryStage*) {

*WebView* webView = new WebView();

*WebEngine* webEngine = webView.getEngine();

*// Завантаження HTML-сторінки, яка відображає графік*

        webEngine.load(getClass().getResource("plot.html").toExternalForm());

*Scene* scene = new Scene(webView, 800, 600);

        primaryStage.setScene(scene);

        primaryStage.setTitle("Function Plotter");

        primaryStage.show();

    }

*public* *static* *void* main(*String*[] *args*) {

        launch(args);

    }

}

## plot.html

<!DOCTYPE *html*>

<html *lang*="en">

<head>

    <meta *charset*="UTF-8">

    <title>Function Plot</title>

*<!-- Підключення plotly.js -->*

    <script *src*="https://cdn.plot.ly/plotly-latest.min.js"></script>

</head>

<body>

    <div *id*="plot"></div>

    <script>

*// JavaScript-код для побудови графіку*

*var* x = [], y = [];

        for (*var* i = 0; i < 10; i++) {

            x.push(i);

            y.push(Math.sin(i)); *// Функція, яку будемо відображати (sin(x) в даному випадку)*

        }

*var* data = [{

            x: x,

            y: y,

            type: 'scatter'

        }];

        Plotly.newPlot('plot', data);

    </script>

</body>

</html>

## Результати виконання програми

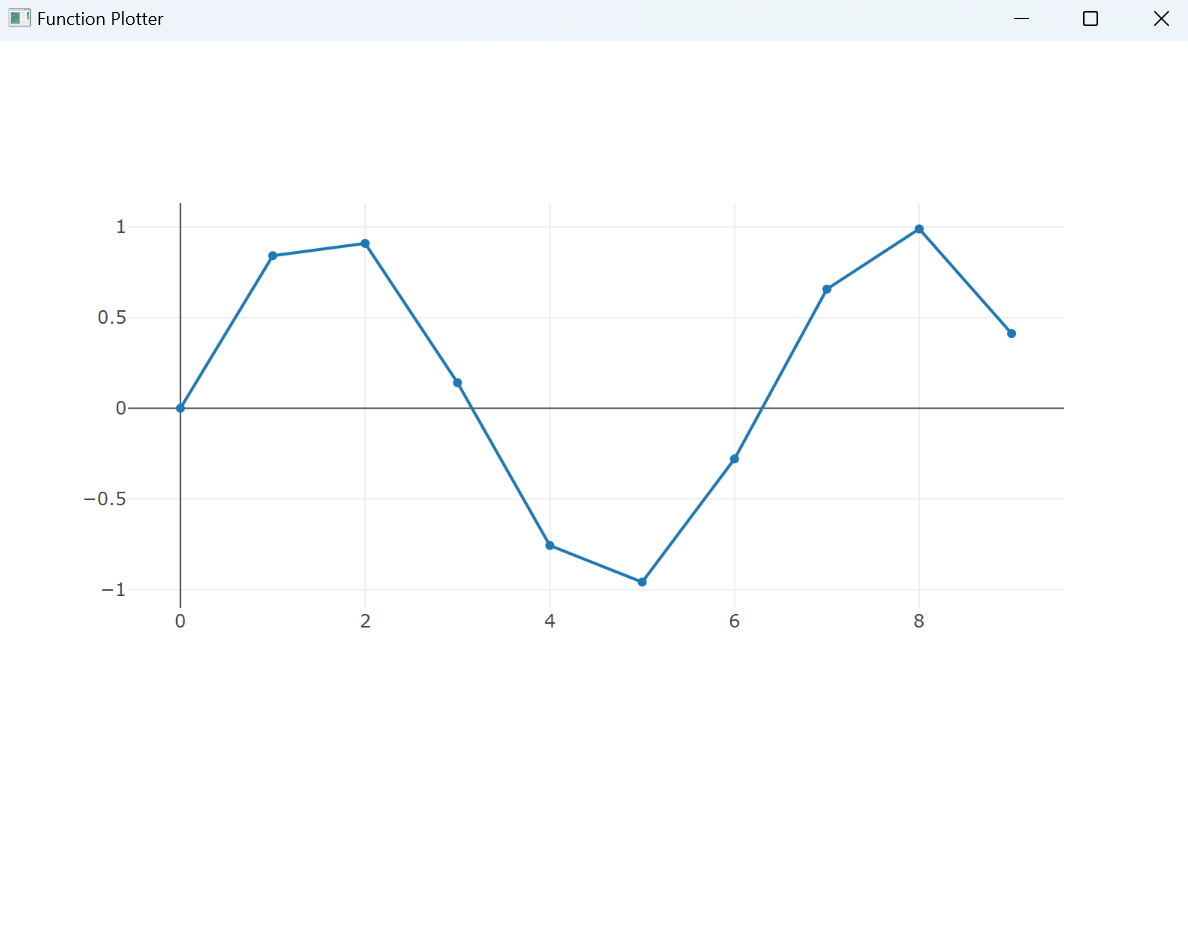


Рисунок 2.2 – «Результат виконання»

# **ВИСНОВОК**

Під час цієї лабораторної роботи було ознайомлення з декількома ключовими технологіями та концепціями програмування. Вивчено JavaFX як інструмент для розробки сучасних графічних інтерфейсів користувача, що дозволяє створювати інтуїтивно зрозумілі та естетичні програми.

Засвоєно принципи багатопотокового програмування в Java, які є необхідними для оптимізації роботи програм та реалізації паралельних обчислень, що особливо важливо для завдань, які вимагають обробки великої кількості даних чи виконання декількох завдань одночасно.

Крім того, було розглянуто використання динамічної компіляції коду для обробки математичних виразів у реальному часі, що відкриває широкі можливості для створення програм, які взаємодіють з користувачем та виконують складні обчислення безпосередньо під час роботи.

Знання та навички, здобуті в ході цієї роботи, є необхідними для розробки ефективних, функціональних та інтерактивних програм у сучасному світі програмування.