# Лабораторная работа № 6

Выполнила: Оленина Арина Игоревна
группа 6204-010302D

Оглавление

# Задание 1

Я добавила в класс Functions метод, возвращающий значение интеграла функции, вычисленное с помощью численного метода.
В качестве параметров метод получает ссылку типа Function на объект функции, значения левой и правой границы области интегрирования, а также шаг дискретизации.
Если интервал интегрирования выходит за границы области определения функции, метод выбрасывает исключение.

Вычисление значения интеграла выполняется по методу трапеций. Для этого вся область интегрирования разбивается на участки, длина которых (кроме одного) равна шагу дискретизации. На каждом таком участке площадь под кривой, описывающейся заданной функцией, приближается площадью трапеции, две вершины которой расположены на оси абсцисс на границах участка, а ещё две – на кривой в точках границ участка. В область интегрирования необязательно укладывается целое количество шагов дискретизации.

```
public static double integrate(Function function, double
leftBorder, double rightBorder, double step) {
    // Проверка корректности параметров
    if (step <= 0) {
        throw new IllegalArgumentException("Шаг интегрирования
должен быть положительным");
    }
    if (leftBorder >= rightBorder) {
        throw new IllegalArgumentException("Левая граница должна
быть меньше правой");
    }

    // Проверка области определения
    if (leftBorder < function.getLeftDomainBorder() ||
rightBorder > function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Интервал
интегрирования выходит за границы области определения функции");
    }

    double integral = 0.0;
    double currentX = leftBorder;

    // Первая точка
    double prevY = function.getFunctionValue(currentX);

    // Проход по всем отрезкам
    while (currentX < rightBorder) {
        double nextX = Math.min(currentX + step, rightBorder);
        double nextY = function.getFunctionValue(nextX);

        // Площадь трапеции: (a + b) * h / 2
```

```
            integral += (prevY + nextY) * (nextX - currentX) / 2.0;

            currentX = nextX;
            prevY = nextY;
        }

        return integral;
}
```

В методе main() проверила работу метода интегрирования. Для этого вычислила интеграл для экспоненты на отрезке от 0 до 1. Определила также, какой шаг дискретизации нужен, чтобы рассчитанное значение отличалось от теоретического в 7 знаке после запятой.

```
public static void main(String[] args) {
    // Тестирование интегрирования экспоненты на отрезке [0, 1]
    Function expFunction = new Exp();
    double theoreticalValue = Math.E - 1; // интеграл (e^x) dx
от 0 до 1 = e - 1
    double integralValue = Functions.integrate(expFunction, 0,
1, 0.0001);

    System.out.println("Теоретическое значение: " +
theoreticalValue);
    System.out.println("Значение, полученное при помощи функции:
" + integralValue);
    System.out.println("Шаг = " + 0.0001 + "\n");

}
```

Вывод:

```
Теоретическое значение: 1.718281828459045
Значение, полученное при помощи функции: 1.7182818298909435
Шаг = 1.0E-4
```

При таком шаге (0.0001) значение начинает отличаться от теоретического в 9 знаке после запятой

А при шаге 0,001 значение уже отличается в 7 точке после запятой:

```
Теоретическое значение: 1.718281828459045
Значение, полученное при помощи функции: 1.7182819716491948
Шаг = 0.001
```

## Задание 2

Далее в приложении один поток инструкций генерирует задачи для интегрирования, а второй поток – решает их. Для этого я создала объект задания, через который эти потоки будут взаимодействовать. В объекте хранятся параметры задания.

Создала пакет threads, в котором разместила классы, связанные с потоками. В пакете threads описала класс Task, объект которого хранит ссылку на объект интегрируемой функции, границы области интегрирования, шаг

дискретизации, а также целочисленное поле, хранящее количество выполняемых заданий. Т.е. объект описывает одно задание.

```
package threads;
import functions.*;
public class Task {
    private Function function;       // Интегрируемая функция
    private double leftBorder;      // Левая граница
интегрирования
    private double rightBorder;     // Правая граница
интегрирования
    private double step;            // Шаг дискретизации
    private int tasksCount;         // Количество выполняемых
заданий

    // Конструктор
    public Task() {
        this.tasksCount = 0;
    }

    // Геттеры и сеттеры
    public Function getFunction() {
        return function;
    }

    public void setFunction(Function function) {
        this.function = function;
    }

    public double getLeftBorder() {
        return leftBorder;
    }

    public void setLeftBorder(double leftBorder) {
        this.leftBorder = leftBorder;
    }

    public double getRightBorder() {
        return rightBorder;
    }

    public void setRightBorder(double rightBorder) {
        this.rightBorder = rightBorder;
    }

    public double getStep() {
        return step;
    }

    public void setStep(double step) {
        this.step = step;
    }

    public int getTasksCount() {
```

```
        return tasksCount;
    }

    public void setTasksCount(int tasksCount) {
        this.tasksCount = tasksCount;
    }
}
```

В главном классе программы описала метод nonThread(), реализующий последовательную (без применения потоков инструкций) версию программы. В методе создала объект класса Task и установила в нём количество выполняемых заданий (100). После этого в цикле (до количества заданий) выполнила следующие действия:

1. создала и поместила в объект задания объект логарифмической функции, основание которой является случайной величиной, распределённой равномерно на отрезке от 1 до 10;
2. указала в объекте задания левую границу области интегрирования (случайно распределена на отрезке от 0 до 100);
3. указала в объекте задания правую границу области интегрирования (случайно распределена на отрезке от 100 до 200);
4. указала в объекте задания шаг дискретизации (случайно распределён на отрезке от 0 до 1);
5. вывела в консоль сообщение вида "Source <левая граница> <правая граница> <шаг дискретизации>";
6. вычислила значение интеграла для параметров из объекта задания;
7. вывела в консоль сообщение вида "Result <левая граница> <правая граница> <шаг дискретизации> <результат интегрирования>".

```
public static void nonThread() {
    System.out.println("nonThread");
    // Объект задания
    Task task = new Task();
    task.setTasksCount(100); // Минимум 100 заданий

    Random random = new Random();

    for (int i = 0; i < task.getTasksCount(); i++) {
        try {
            // Логарифмическая функция со случайным основанием
от 1 до 10
            double base = 1 + random.nextDouble() * 9;
            Function logFunction = new Log(base);
            task.setFunction(logFunction);

            // Левая граница: [0, 100]
            double leftBorder = random.nextDouble() * 100;
            task.setLeftBorder(leftBorder);
```

```
            // Правая граница: [100, 200]
            double rightBorder = 100 + random.nextDouble() *
100;
            task.setRightBorder(rightBorder);

            // Шаг дискретизации: [0, 1]
            double step = 0.01 + random.nextDouble() * 0.99;
            task.setStep(step);

            // Вывод информации о задании
            System.out.println("Source LeftBorder: " +
task.getLeftBorder()
                    + " RightBorder: " + task.getRightBorder() +
" Step: " + task.getStep() + "\n");

            // Вычисление интеграла
            double result =
Functions.integrate(task.getFunction(),
                    task.getLeftBorder(), task.getRightBorder(),
task.getStep());

            // Вывод результата
            System.out.println("Result LeftBorder: " +
task.getLeftBorder()
                    + " RightBorder: " + task.getRightBorder() +
" Step: " + task.getStep() + " Result: " + result + "\n");

        } catch (Exception e) {
            System.out.println("Ошибка в задании " + i + ": " +
e.getMessage());
        }
    }
}
```

Проверила работу метода, вызвав его в методе main().
Часть вывода:

```
Source LeftBorder: 39.097640981476424 RightBorder: 140.78904813617527 Step:
0.40287087345397954

Result LeftBorder: 39.097640981476424 RightBorder: 140.78904813617527 Step:
0.40287087345397954 Result: 199.54784890850038

Source LeftBorder: 95.61884660832148 RightBorder: 183.4455921322648 Step:
0.989407675453242

Result LeftBorder: 95.61884660832148 RightBorder: 183.4455921322648 Step:
0.989407675453242 Result: 1724.179704839774
```

Source LeftBorder: 74.13952184028778 RightBorder: 179.59207541546812 Step: 0.623572598745622

Result LeftBorder: 74.13952184028778 RightBorder: 179.59207541546812 Step: 0.623572598745622 Result: 257.9794688112341

Source LeftBorder: 78.80098171975824 RightBorder: 112.03557216785069 Step: 0.6027783084039704

Result LeftBorder: 78.80098171975824 RightBorder: 112.03557216785069 Step: 0.6027783084039704 Result: 122.06834426345762

Source LeftBorder: 42.256581473423296 RightBorder: 147.20278082387313 Step: 0.6621581801346837

Result LeftBorder: 42.256581473423296 RightBorder: 147.20278082387313 Step: 0.6621581801346837 Result: 287.16268097473386

Source LeftBorder: 43.921616990686374 RightBorder: 189.86332130836087 Step: 0.03773424568845618

Result LeftBorder: 43.921616990686374 RightBorder: 189.86332130836087 Step: 0.03773424568845618 Result: 328.9408452060418

Source LeftBorder: 38.22011157933496 RightBorder: 179.5675943916512 Step: 0.22295327265755005

Result LeftBorder: 38.22011157933496 RightBorder: 179.5675943916512 Step: 0.22295327265755005 Result: 842.4911880981941

Source LeftBorder: 7.431151567427163 RightBorder: 144.8911347166911 Step: 0.26823063889318954

Result LeftBorder: 7.431151567427163 RightBorder: 144.8911347166911 Step: 0.26823063889318954 Result: 595.4140370097069

## Задание 3

В пакете threads создала два следующих класса. При их реализации воспользовалась фрагментами последовательной версии программы из предыдущего задания.
Класс SimpleGenerator реализует интерфейс Runnable, получает в конструкторе и сохраняет в своё поле ссылку на объект типа Task, а в методе run() в цикле формируются задачи и заносятся в полученный объект задания, а также выводятся сообщения в консоль.

```java
package threads;

import functions.*;
import functions.basic.*;
import java.util.Random;

public class SimpleGenerator implements Runnable {
    private Task task;

    public SimpleGenerator(Task task) {
        this.task = task;
    }

    public void run() {
        Random random = new Random();

        for (int i = 0; i < task.getTasksCount(); ++i) {
            // Логарифмическая функция со случайным основанием
от 1 до 10
            double base = 1 + random.nextDouble() * 9;
            Function function = new Log(base);

            // Левая граница: [0, 100]
            double leftBorder = random.nextDouble() * 100;

            // Правая граница: [100, 200]
            double rightBorder = 100 + random.nextDouble() *
100;

            // Шаг дискретизации: [0, 1]
            double step = random.nextDouble();

            // Установка данных в задание
            task.setFunction(function);
            task.setLeftBorder(leftBorder);
            task.setRightBorder(rightBorder);
            task.setStep(step);

            // Вывод сообщения
            System.out.println("Source " + leftBorder + " " +
rightBorder + " " + step);
        }
    }
}
```

Класс SimpleIntegrator реализует интерфейс Runnable, получает в конструкторе и сохраняет в своё поле ссылку на объект типа Task, а в методе run() в цикле решаются задачи, данные для которых берутся из полученного объекта задания, а также выводятся сообщения в консоль.

```java
package threads;

import functions.*;
```

```
import functions.basic.*;
import java.util.Random;

public class SimpleIntegrator implements Runnable {
    private Task task;

    public SimpleIntegrator(Task task) {
        this.task = task;
    }

    public void run() {
        for (int i = 0; i < task.getTasksCount(); ++i) {
            // Данные из задания
            double leftBorder = task.getLeftBorder();
            double rightBorder = task.getRightBorder();
            double step = task.getStep();
            if (step <= 0) {
                System.out.println("Пропущено задание с step=" +
step);
                continue;
            }

            // Вычисление интеграла
            double result =
Functions.integrate(task.getFunction(), leftBorder, rightBorder,
step);

            // Вывод результата
            System.out.println("Result " + leftBorder + " " +
rightBorder + " " + step + " " + result);
        }
    }
}
```

В главном классе программы создала метод simpleThreads(). В нём создала объект задания, указала количество выполняемых заданий (100), создала и запустила два потока вычислений, основанных на описанных классах SimpleGenerator и SimpleIntegrator. Проверила работу метода, вызвав его в методе main().

```
public static void simpleThreads() {
    System.out.println("simpleThreads");
    // Создание объекта задания
    Task task = new Task();
    // Количество выполняемых заданий (минимум 100)
    task.setTasksCount(100);
    // Создание потоков вычислений
    Thread generatorThread = new Thread(new
SimpleGenerator(task));
    Thread integratorThread = new Thread(new
SimpleIntegrator(task));
    // Запуск потоков
    generatorThread.start();
```

```
    integratorThread.start();
}
```
Затем попробовала запускать программу (несколько раз). Попробовала запускать программу, изменяя приоритеты потоков перед их запуском:
```
generatorThread.setPriority(Thread.MAX_PRIORITY);
integratorThread.setPriority(Thread.MIN_PRIORITY);
```
и наоборот:
```
integratorThread.setPriority(Thread.MAX_PRIORITY);
generatorThread.setPriority(Thread.MIN_PRIORITY);
```
Убедилась, что в некоторых случаях интегрирующий поток может прекращать своё выполнение из-за исключения NullPointerException. Определила причину возникновения этого исключения и сделала так, чтобы оно не возникало (добавила проверку функции на null в SimpleIntegrator) :
```
if (function == null) {
    System.out.println("Пропущено задание - функция не готова");
    continue;
}
```
Попробовала запускать программу (несколько раз). Убедилась, что в некоторых случаях интегрирующий поток выводит сообщение с набором данных, который не встречается в сообщениях генерирующего потока (например, левая граница области интегрирования оказывается взята из одного задания, а правая граница и шаг дискретизации – из другого). Определила причину возникновения таких ситуаций (отсутствие синхронизации) и устранила её, используя блоки синхронизации в методах run() генерирующего и интегрирующего потоков. Затем убедилась, что в коде нигде не возникает необоснованно длительной блокировки объектов.
```
public class SimpleIntegrator implements Runnable {
    private Task task;

    public SimpleIntegrator(Task task) {
        this.task = task;
    }

    public void run() {
        for (int i = 0; i < task.getTasksCount(); ++i) {
            synchronized (task) {
                // Данные из задания
                double leftBorder = task.getLeftBorder();
                double rightBorder = task.getRightBorder();
                double step = task.getStep();
                Function function = task.getFunction();

                if (step <= 0) {
                    System.out.println("Пропущено задание с
step=" + step);
                    continue;
                }
                if (function == null) {
                    System.out.println("Пропущено задание -
```

```
функция не готова");
                    continue;
                }

                // Вычисление интеграла
                double result = Functions.integrate(function,
leftBorder, rightBorder, step);

                // Вывод результата
                System.out.println("Result " + leftBorder + " "
+ rightBorder + " " + step + " " + result);
            }
        }
    }
}

public class SimpleGenerator implements Runnable {
    private Task task;

    public SimpleGenerator(Task task) {
        this.task = task;
    }

    public void run() {
        Random random = new Random();

        for (int i = 0; i < task.getTasksCount(); ++i) {
            synchronized (task) {
                // Логарифмическая функция со случайным
основанием от 1 до 10
                double base = 1 + random.nextDouble() * 9;
                Function function = new Log(base);

                // Левая граница: [0, 100]
                double leftBorder = random.nextDouble() * 100;

                // Правая граница: [100, 200]
                double rightBorder = 100 + random.nextDouble() *
100;

                // Шаг дискретизации: [0.01, 1]
                double step = 0.01 + random.nextDouble() * 0.99;

                // Установка данных в задание
                task.setFunction(function);
                task.setLeftBorder(leftBorder);
                task.setRightBorder(rightBorder);
                task.setStep(step);

                // Вывод сообщения
                System.out.println("Source " + leftBorder + " "
+ rightBorder + " " + step);
            }
```

```
        }
    }
}
```

## Задание 4

Затем я определила причину того, что не все сгенерированные задания оказываются выполнены интегрирующим потоком (генератор работает быстрее и часто захватывает монитор для чтения, а интегратор в это время не может работать и постоянно ждет доступа, в результате интегратор обрабатывает меньше заданий).

Для устранения этой проблемы потребуется дополнительный объект, представляющий собой одноместный семафор, различающий операции чтения и записи в защищаемый объект.

В пакете threads создала два следующих класса. При их реализации воспользовалась фрагментами классов из предыдущего задания.
Класс Generator расширяет класс Thread, получает в конструкторе и сохраняет в свои поля ссылки на объект типа Task и на объект семафора, а в методе run() выполняются те же действия, что и в предыдущей версии генерирующего класса.

```
package threads;

import functions.*;
import functions.basic.*;
import java.util.concurrent.Semaphore;

public class Generator extends Thread {
    private Task task;
    private Semaphore writeSemaphore;
    private Semaphore readSemaphore;

    public Generator(Task task, Semaphore writeSemaphore,
Semaphore readSemaphore) {
        this.task = task;
        this.writeSemaphore = writeSemaphore;
        this.readSemaphore = readSemaphore;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < task.getTasksCount(); ++i) {
                // Проверка прерывания
                if (Thread.interrupted()) {
                    System.out.println("Generator был прерван");
                    return;
                }
```

```
                // Генерация данных
                double base = 1 + Math.random() * 9;
                Function function = new Log(base);
                double leftBorder = Math.random() * 100;
                double rightBorder = 100 + Math.random() * 100;
                double step = 0.01 + Math.random() * 0.99;

                // Захват семафора записи
                writeSemaphore.acquire();
                try {
                    // Запись данных в задание
                    task.setFunction(function);
                    task.setLeftBorder(leftBorder);
                    task.setRightBorder(rightBorder);
                    task.setStep(step);

                    System.out.println("Source " + leftBorder +
" " + rightBorder + " " + step);
                } finally {
                    // Разрешение чтения
                    readSemaphore.release();
                }
            }
        } catch (InterruptedException e) {
            System.out.println("Generator был прерван при
ожидании семафора");
        }
    }
}
```

Класс Integrator расширяет класс Thread, получает в конструкторе и сохраняет в свои поля ссылки на объект типа Task и на объект семафора, а в методе run() выполняются те же действия, что и в предыдущей версии интегрирующего класса.

```
package threads;

import functions.Function;
import functions.Functions;
import functions.InappropriateFunctionPointException;
import java.util.concurrent.Semaphore;

public class Integrator extends Thread {
    private Task task;
    private Semaphore writeSemaphore;
    private Semaphore readSemaphore;

    public Integrator(Task task, Semaphore writeSemaphore,
Semaphore readSemaphore) {
        this.task = task;
        this.writeSemaphore = writeSemaphore;
        this.readSemaphore = readSemaphore;
    }
```

```
    @Override
    public void run() {
        try {
            for (int i = 0; i < task.getTasksCount(); ++i) {
                // Проверка прерывания
                if (Thread.interrupted()) {
                    System.out.println("Integrator был
прерван");
                    return;
                }

                // Ожидание данных для чтения
                readSemaphore.acquire();
                try {
                    // Чтение данных из задания
                    double leftBorder = task.getLeftBorder();
                    double rightBorder = task.getRightBorder();
                    double step = task.getStep();
                    Function function = task.getFunction();

                    // Проверка корректности данных
                    if (function == null || step <= 0) {
                        System.out.println("Пропущено задание -
некорректные данные");
                        continue;
                    }

                    // Вычисление интеграла
                    double result =
Functions.integrate(function, leftBorder, rightBorder, step);
                    System.out.println("Result " + leftBorder +
" " + rightBorder + " " + step + " " + result);
                } finally {
                    // Разрешение записи
                    writeSemaphore.release();
                }
            }
        } catch (InterruptedException e) {
            System.out.println("Integrator был прерван при
ожидании семафора");
        }
    }
}
```

Отличие этих классов от предыдущих версий заключается в том, что вместо средств синхронизации в методах run() используются возможности семафора. В главном классе программы создала метод complicatedThreads(). В нём создала объект задания, указала количество выполняемых заданий (минимум 100), создала и запустила два потока вычислений
классов Generator и Integrator. Проверила работу метода, вызвав его в методе main().

```
public static void complicatedThreads() {
    System.out.println("complicatedThreads");
```

```
    // Объект задания
    Task task = new Task();
    task.setTasksCount(100);

    // Два семафора:
    // writeSemaphore начинается с 1 разрешения (можно писать)
    // readSemaphore начинается с 0 разрешений (нельзя читать)
    Semaphore writeSemaphore = new Semaphore(1);
    Semaphore readSemaphore = new Semaphore(0);

    // Создание потоков
    Generator generator = new Generator(task, writeSemaphore,
readSemaphore);
    Integrator integrator = new Integrator(task, writeSemaphore,
readSemaphore);

    // Запуск потоков
    generator.start();
    integrator.start();

    try {
        // Ожидание завершения потоков
        generator.join();
        integrator.join();

    } catch (InterruptedException e) {
        System.out.println("Основной поток был прерван");
    }
}
```

Также попробовала запускать программу (несколько раз). И попробовала запускать программу, изменяя приоритеты потоков перед их запуском. Затем сделала так, чтобы после создания потоков основной поток программы выжидал 50 миллисекунд, после чего прерывал работу потов путём вызова метода interrupt(). Изменила код классов потоков так, чтобы прерывание происходило корректно.

```
try {
    // 50 миллисекунд
    Thread.sleep(50);

    // Прерывание потоки
    generator.interrupt();
    integrator.interrupt();

    // Ожидание завершения потоков
    generator.join();
    integrator.join();

} catch (InterruptedException e) {
    System.out.println("Основной поток был прерван");
}
```