Лабораторная работа № 6

Титульный лист

Дисциплина: Объектно-ориентированное программирование
Выполнил: Мостовщиков Владимир Витальевич
Группа: 6204-010302D
Преподаватель: Борисов Дмитрий Сергеевич
Год: 2025

Содержание:

Задание 1. Реализация метода integral() в классе Functions

В первом задании был реализован статический метод integral в классе Functions. Метод принимает ссылку на объект типа Function и параметры интервала интегрирования: левую границу, правую границу и шаг дискретизации. Численное интегрирование выполняется по методу трапеций: вся область интегрирования делится на отрезки длиной step, на каждом участке площадь под графиком функции аппроксимируется площадью трапеции.
Перед началом расчетов метод проверяет корректность параметров: левая граница должна быть строго меньше правой, шаг дискретизации должен быть положительным. Дополнительно проверяется область определения функции. Если при вычислении значения функции на какой-то точке интервала получается не число (NaN) или бесконечность (Infinity), метод рассматривает это как выход за область определения и выбрасывает исключение IllegalArgumentException.

Для проверки корректности работы метода integral в методе main предварительно вычислялся интеграл экспоненты exp(x) на отрезке [0; 1] и сравнивался с теоретическим значением e - 1.

```
public static double integral(Function f, double left, double right, double step) {  no usages  new *
        // Проверка входных данных
    if (f == null)
        throw new IllegalArgumentException("Функция не должна быть null");

    if (Double.isNaN(left) || Double.isNaN(right) || Double.isNaN(step))
        throw new IllegalArgumentException("Границы и шаг интегрирования не должны быть NaN");

    if (!Double.isFinite(left) || !Double.isFinite(right) || !Double.isFinite(step))
        throw new IllegalArgumentException("Границы и шаг интегрирования должны быть конечными числами");

    if (step <= 0.0)
        throw new IllegalArgumentException("Шаг интегрирования должен быть > 0. Получено: " + step);

    if (left >= right)
        throw new IllegalArgumentException("Левая граница должна быть меньше правой. left = " + left + ", right = " + right);

    // Получаем границы области определения функции
    double domainLeft = f.getLeftDomainBorder();
    double domainRight = f.getRightDomainBorder();
```

```
    // Проверяем что запрашиваемый интервал полностью находится в области определения
    if (left < domainLeft || right > domainRight) {
        throw new IllegalArgumentException(
                "Интервал [" + left + "; " + right + "] выходит за область определения функции: [" +
                        domainLeft + "; " + domainRight + "]");}
    double result = 0;
    double x = left;      // Начинаем с левой границы интервала

    while (x < right) {  // Проходим по всему интервалу от left до right с шагом step
        double next = x + step;
        if (next > right)
            next = right;  // На последнем отрезке берем правую границу
        // Вычисляем значения функции в текущей и следующей точках
        double fx = f.getFunctionValue(x);
        double fNext = f.getFunctionValue(next);
        double h = next - x;  // Фактическая длина текущего отрезка
        // Добавляем площадь текущей трапеции к результату
        result += (fx + fNext) * h / 2.0;
        x = next;} // Переходим к следующему отрезку
    return result;  // Возвращаем вычисленное значение интеграла
}
```

Рисунок 1,2 - реализация метода integral в классе Functions

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


    Function exp = new Exp();
    double theoretical = Math.E - 1;
    double step = 0.0001;
    double numeric = Functions.integral(exp, left: 0, right: 1, step);
    System.out.println("Теоретическое значение: " + theoretical);
    System.out.println("Численное значение: " + numeric);
    System.out.println("Разность: " + Math.abs(theoretical - numeric));
    System.out.println("Шаг: " + step);


}
```

Рисунок 3 - фрагмент кода main интеграла exp(x) на отрезке [0; 1]

```
Теоретическое значение: 1.718281828459045
Численное значение: 1.7182818298909435
Разность: 1.4318983776462346E-9
Шаг: 1.0E-4
```

Рисунок 4 – вывод проверки в консоль

Задание 2. Последовательная реализация программы nonThread

Я реализовал последовательную версию программы в методе nonThread(), где все вычисления выполняются в одном основном потоке.

Сначала создается объект Task, который хранит параметры для вычислений: функцию, границы интегрирования, шаг и общее количество заданий. Затем в цикле последовательно генерируются и сразу же вычисляются задания:

1. Генерация параметров:
    - Основание логарифма в диапазоне (1; 10)
    - Левая граница в пределах (0; 100)
    - Правая граница в диапазоне (100; 200)
    - Шаг интегрирования от 0 до 1
2. Вычисления:
    - Параметры сохраняются в объект Task
    - Вычисляется интеграл методом трапеций
    - Результаты выводятся в консоль

Таким образом, каждое задание полностью обрабатывается перед переходом к следующему, что обеспечивает последовательное выполнение всех операций в одном потоке.

В отчете я оставлю только первые и последние 5 заданий, чтобы не было избыточного количества картинок.

```
=== nonThread(): последовательная версия ===
[NON] task=  1 left=19,03664 right=112,15627 step=0,50636
[NON-RES] task=  1 value=192,8302836664
[NON] task=  2 left=67,40017 right=153,96174 step=0,38728
[NON-RES] task=  2 value=185,8550092794
[NON] task=  3 left=92,77816 right=156,67641 step=0,31857
[NON-RES] task=  3 value=200,9811585213
[NON] task=  4 left=66,55419 right=190,86272 step=0,74261
[NON-RES] task=  4 value=646,0172196003
[NON] task=  5 left=8,85236 right=107,05770 step=0,27501
[NON-RES] task=  5 value=167,5120307627
```

```
[NON] task= 95 left=92,87384 right=101,37340 step=0,02836
[NON-RES] task= 95 value=26,2996273135
[NON] task= 96 left=84,99520 right=119,43139 step=0,12958
[NON-RES] task= 96 value=116,4811840600
[NON] task= 97 left=3,25777 right=172,11721 step=0,43178
[NON-RES] task= 97 value=376,7350622072
[NON] task= 98 left=54,13014 right=120,97912 step=0,07553
[NON-RES] task= 98 value=139,6758316127
[NON] task= 99 left=81,35612 right=115,09723 step=0,40018
[NON-RES] task= 99 value=83,3572328947
[NON] task=100 left=35,11638 right=125,52101 step=0,39048
[NON-RES] task=100 value=394,8677002563
```

Рисунок 5,6 - вывод работы nonThread (сделал вывод в 1 строку, так же изменено для последующих заданий)

Задание 3. Простая многопоточная версия simpleThreads

Я реализовал простую многопоточную версию программы, где генерация задач и вычисление интегралов разделены между двумя потоками.

SimpleGenerator - поток для генерации задач. В цикле он создает параметры логарифмической функции, записывает их в общий объект Task и выводит информацию с пометкой [S-GEN].

SimpleIntegrator - поток для вычислений. Он читает параметры из того же объекта Task, вычисляет интеграл и выводит результаты с пометкой [S-INT].

Запуск в simpleThreads():

1. Создаю общий Task на 100 заданий
2. Запускаю оба потока параллельно
3. Ожидаю их завершения

```java
package threads;

import functions.Function;
import functions.basic.Log;

import java.util.Random;

// Генератор задач для вычисления интеграла логарифмической функции
public class SimpleGenerator implements Runnable {  3 usages  Metilka *
    private final Task task;  7 usages
    private final Random random = new Random();  // Генератор случайных чисел  4 usages

    public SimpleGenerator(Task task) {  5 usages  Metilka
        if (task == null)
            throw new IllegalArgumentException("Task не должен быть null");
        this.task = task;
    }

    @Override  Metilka *
    public void run() {
        int tasksCount = task.getTasksCount(); // Копия количества задач
        for (int i = 0; i < tasksCount; ++i) { // Используем локальную переменную
            int taskNumber = i + 1;
```

```
double base = 1 + 9 * random.nextDouble();     // Генерируем основание логарифма в диапазоне (1, 10)
if (Math.abs(base - 1) < 1e-6)                 // Избегаем основания 1
    base += 1e-3;

Function logFunction = new Log(base);

// Левая граница в диапазоне (0, 100)
double left = 100 * random.nextDouble();

if (left <= 0)
    left = Double.MIN_VALUE; // Гарантируем положительное значение

// Правая граница в диапазоне (100, 200), обязательно больше левой
double right = 100 + 100 * random.nextDouble();

if (right <= left)
    right = left + 1; // Гарантируем что right > left

// Шаг интегрирования в диапазоне (0, 1]
double step = random.nextDouble();
```

```
            double step = random.nextDouble();

            if (step <= 0)
                step = 1; // Минимальный шаг = 1

            // Заполняем общую задачу сгенерированными параметрами
            synchronized (task) {
                task.setFunction(logFunction);
                task.setLeft(left);
                task.setRight(right);
                task.setStep(step);

                // Выводим информацию о сгенерированной задаче
                System.out.printf("[S-GEN] task=%3d base=%.5f left=%.5f right=%.5f step=%.5f%n",
                        taskNumber, base, left, right, step);
            }
        }
    }
}
```

Рисунок 7,8,9 – исправленная версия класса SimpleGenerator

```java
package threads;
import functions.Function;
import functions.Functions;

// Вычислитель интеграла
public class SimpleIntegrator implements Runnable {  3 usages   Metilka *
    private final Task task;  7 usages

    public SimpleIntegrator(Task task) {  5 usages   Metilka
        if (task == null)
            throw new IllegalArgumentException("Task не должен быть null");
        this.task = task;
    }
    @Override  Metilka *
    public void run() {
        int tasksCount = task.getTasksCount(); // копия количества задач
        for (int i = 0; i < tasksCount; ++i) { // Выполняем вычисления для указанного количества задач
            int taskNumber = i + 1;

            Function f;
            double left;
            double right;
            double step;
```

```java
            synchronized (task) {
                f = task.getFunction();
                left = task.getLeft();
                right = task.getRight();
                step = task.getStep();
            }
            // Простая защита от NullPointerException
            if (f == null) {
                System.out.printf("[S-INT] task=%3d function=null, интегрирование пропущено%n", taskNumber);
                continue;
            }
            // Вычисляем интеграл с полученными параметрами
            double value = Functions.integral(f, left, right, step);

            // Вывод результата в одну строку
            System.out.printf(
                    "[S-INT] task=%3d left=%.5f right=%.5f step=%.5f value=%.10f%n",
                    taskNumber, left, right, step, value
            );
        }
    }
```

Рисунок 10,11 – исправленная версия класса SimpleIntegrator

```
=== simpleThreads(): простая многопоточная версия ===
[S-GEN] task=  1 base=3,84019 left=22,95119 right=124,73539 step=0,28824
[S-GEN] task=  2 base=2,21266 left=28,51660 right=182,58286 step=0,29644
[S-GEN] task=  3 base=4,85591 left=34,97715 right=116,01423 step=0,39906
[S-GEN] task=  4 base=7,69268 left=56,65223 right=147,24121 step=0,61342
[S-GEN] task=  5 base=8,46043 left=57,84326 right=174,60903 step=0,63892
[S-GEN] task=  6 base=3,10671 left=82,60740 right=138,29721 step=0,47201
```

```
[S-GEN] task= 95 base=6,10886 left=66,30586 right=135,28984 step=0,37161
[S-GEN] task= 96 base=5,08292 left=65,26189 right=146,25961 step=0,57200
[S-GEN] task= 97 base=5,76032 left=85,26655 right=162,31888 step=0,74868
[S-GEN] task= 98 base=8,59692 left=8,38332 right=158,13890 step=0,14101
[S-GEN] task= 99 base=4,70644 left=61,33371 right=158,59586 step=0,29959
[S-GEN] task=100 base=3,40780 left=90,33461 right=198,94175 step=0,32731
[S-INT] task=  3 left=30,15850 right=125,41327 step=0,87939 value=210,6318968475
[S-INT] task=  4 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task=  5 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task=  6 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task=  7 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task=  8 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
```

```
[S-INT] task= 95 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task= 96 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task= 97 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task= 98 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task= 99 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
[S-INT] task=100 left=90,33461 right=198,94175 step=0,32731 value=438,4511711146
simpleThreads(): оба потока завершили работу
```

Рисунок 12,13,14 - пример консольного вывода работы simpleThreads первые и последние задания

Задание 4. Многопоточная версия с семафорами complicatedThreads

В ходе выполнения данного задания я реализовал многопоточную версию с семафорами, которая решает проблему гонки данных из предыдущей версии.

Поэтому я использую два семафора для четкого чередования операций, dataProcessed (начальное значение 1) - разрешает генератору начать запись

dataReady (начальное значение 0) - сигнализирует о готовности данных для интегратора

Как работают потоки:

Generator:

1. Ждет разрешения от dataProcessed
2. Генерирует параметры задачи
3. Записывает их в Task
4. Разрешает чтение через dataReady

Integrator:

1. Ждет сигнала dataReady
2. Читает параметры из Task
3. Вычисляет интеграл
4. Освобождает dataProcessed для следующей генерации

В методе complicatedThreads():
Запускаю оба потока, даю им поработать 50мс, затем корректно прерываю и ожидаю завершения.

Теперь каждое задание гарантированно обрабатывается ровно один раз, без потерь и смешивания параметров. Потоки работают согласованно, чередуя генерацию и вычисления.

```java
package threads;
import functions.Function;
import functions.basic.Log;
import java.util.Random;
import java.util.concurrent.Semaphore;
// Генератор задач с синхронизацией через семафоры
public class Generator extends Thread {  1 usage  new *
    private final Task task;              // Общая задача для передачи данных  6 usages
    private final Semaphore dataReady;    // Семафор "данные готовы" (отпускается генератором)  2 usage
    private final Semaphore dataProcessed; // Семафор "данные обработаны" (отпускается интегратором)
    private final Random random = new Random();  4 usages

    public Generator(Task task, Semaphore dataReady, Semaphore dataProcessed) {  4 usages  new *
        super( name: "Generator");
        if (task == null)
            throw new IllegalArgumentException("Task не должен быть null");
        if (dataReady == null || dataProcessed == null)
            throw new IllegalArgumentException("Семафоры не должны быть null");
        this.task = task;
        this.dataReady = dataReady;
        this.dataProcessed = dataProcessed;
    }
    @Override  new *
    public void run() {
public void run() {
    int tasksCount = task.getTasksCount();
    try {
        for (int i = 0; i < tasksCount; ++i) {
            // Проверка прерывания потока
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("Generator: обнаружен флаг прерывания, выходим из цикла");
                break;
            }
            int taskNumber = i + 1;
            // Генерация параметров логарифмической функции
            double base = 1.0 + 9.0 * random.nextDouble();
            if (Math.abs(base - 1.0) < 1e-6)
                base += 1e-3; // Избегаем основания 1
            Function logFunction = new Log(base);
            // Генерация левой границы
            double left = 100.0 * random.nextDouble();
            if (left <= 0.0)
                left = Double.MIN_VALUE;
            // Генерация правой границы
            double right = 100.0 + 100.0 * random.nextDouble();
            if (right <= left)
                right = left + 1.0;
```

```
            double step = random.nextDouble();
            if (step <= 0.0)
                step = 1.0;
            dataProcessed.acquire();  // Ожидание разрешения от интегратора
            try { // Запись сгенерированных параметров в общую задачу
                task.setFunction(logFunction);
                task.setLeft(left);
                task.setRight(right);
                task.setStep(step);
                System.out.printf( // Вывод информации
                        "[GEN] Задание %3d:%n" + " левая граница = %.6f%n" + " правая граница = %.6f%n" +
                                " шаг дискретизации = %.6f%n" + " основание log a    = %.6f%n%n",
                        taskNumber, left, right, step, base);
            } finally {
                dataReady.release();} // Уведомление интегратора о готовности данных
        }
    } catch (InterruptedException e) {
        System.out.println("Generator: прерван при ожидании семафора");
        Thread.currentThread().interrupt();}
    System.out.println("Generator: завершение работы потока");
}
```

Рисунок 15,16,17 – реализация класса Generator

```
package threads;
import functions.Function;
import functions.Functions;
import java.util.concurrent.Semaphore;
// Вычислитель интеграла с синхронизацией через семафоры
public class Integrator extends Thread {  1 usage  new *
    private final Task task;              // Общая задача для получения данных  6 usages
    private final Semaphore dataReady;    // Семафор "данные готовы" (отпускается генератором)  2 usag
    private final Semaphore dataProcessed; // Семафор "данные обработаны" (отпускается интегратором)

    public Integrator(Task task, Semaphore dataReady, Semaphore dataProcessed) {  4 usages  new *
        super( name: "Integrator");
        if (task == null)
            throw new IllegalArgumentException("Task не должен быть null");

        if (dataReady == null || dataProcessed == null)
            throw new IllegalArgumentException("Семафоры не должны быть null");
        this.task = task;
        this.dataReady = dataReady;
        this.dataProcessed = dataProcessed;
    }
    @Override  new *
    public void run() {
        int tasksCount = task.getTasksCount();
```

```java
public void run() {
    int tasksCount = task.getTasksCount();
    try {
        for (int i = 0; i < tasksCount; ++i) {
            // Проверка прерывания потока
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("Integrator: обнаружен флаг прерывания, выходим из цикла");
                break;
            }
            int taskNumber = i + 1;
            // Ожидаем пока генератор подготовит данные
            dataReady.acquire();
            double left;
            double right;
            double step;
            Function function;
            try {
                // Читаем параметры из общей задачи
                function = task.getFunction();
                left = task.getLeft();
                right = task.getRight();
                step = task.getStep();
            } finally { // Уведомляем генератор о том что данные обработаны
```

```java
            try {
                // Читаем параметры из общей задачи
                function = task.getFunction();
                left = task.getLeft();
                right = task.getRight();
                step = task.getStep();
            } finally { // Уведомляем генератор о том что данные обработаны
                dataProcessed.release();
            }
            // Вычисляем интеграл
            double result = Functions.integral(function, left, right, step);
            // Выводим результат вычислений
            System.out.printf("[INT] Задание %3d:%n" + " левая граница = %.6f%n" + " правая граница = %.6f%n" +
                              " шаг дискретизации  = %.6f%n" + " значение интеграла = %.6f%n%n",
                    taskNumber, left, right, step, result);
        }
    } catch (InterruptedException e) {
        System.out.println("Integrator: прерван при ожидании семафора");
        Thread.currentThread().interrupt();
    }
    System.out.println("Integrator: завершение работы потока");
}
```

Рисунок 18,19,20 – реализация класса Integrator

```
=== complicatedThreads():  многопоточная версия с семафорами ===
[GEN] task=  1 base=95,673043 left=121,401172 right=0,656954 step=5,077085
[INT] task=  1 left=95,673043 right=121,401172 step=0,656954 value=74,1840150349
[GEN] task=  2 base=50,163930 left=167,103304 right=0,318354 step=8,671064
[GEN] task=  3 base=27,913945 left=122,499029 right=0,034351 step=9,375174
[INT] task=  2 left=50,163930 right=167,103304 step=0,318354 value=250,9226944315
[GEN] task=  4 base=70,875071 left=110,834508 right=0,554188 step=4,455340
[INT] task=  3 left=27,913945 right=122,499029 step=0,034351 value=179,3843370493
[INT] task=  4 left=70,875071 right=110,834508 step=0,554188 value=120,3802884517
[GEN] task=  5 base=30,799866 left=172,903835 right=0,253997 step=7,812182
[GEN] task=  6 base=81,985982 left=149,958356 right=0,357803 step=6,252285
[INT] task=  5 left=30,799866 right=172,903835 step=0,253997 value=312,9161346768
```

```
[INT] task= 93 left=14,528921 right=154,650586 step=0,553265 value=308,3019155850
[GEN] task= 94 base=8,394522 left=106,146086 right=0,148599 step=5,324683
[GEN] task= 95 base=16,456990 left=104,811302 right=0,485752 step=4,842042
[INT] task= 94 left=8,394522 right=106,146086 step=0,148599 value=226,9498109413
[INT] task= 95 left=16,456990 right=104,811302 step=0,485752 value=223,8912723719
[GEN] task= 96 base=97,688712 left=130,925221 right=0,777689 step=9,196963
[INT] task= 96 left=97,688712 right=130,925221 step=0,777689 value=70,9306452307
[GEN] task= 97 base=90,479050 left=135,023938 right=0,814071 step=9,054536
[INT] task= 97 left=90,479050 right=135,023938 step=0,814071 value=95,3990817562
[GEN] task= 98 base=96,441714 left=103,307780 right=0,748298 step=8,403613
[INT] task= 98 left=96,441714 right=103,307780 step=0,748298 value=14,8494298814
[GEN] task= 99 base=98,718540 left=155,961713 right=0,024083 step=3,427900
[GEN] task=100 base=41,491712 left=165,965361 right=0,349800 step=5,122829
Generator: завершение работы потока
[INT] task= 99 left=98,718540 right=155,961713 step=0,024083 value=224,8149447981
[INT] task=100 left=41,491712 right=165,965361 step=0,349800 value=348,4880486800
Integrator: завершение работы потока
complicatedThreads(): оба потока завершили работу

Process finished with exit code 0
```

Рисунок 21,22 - пример консольного вывода работы complicatedThreads(),с исправленным выводом в 1 строчку