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

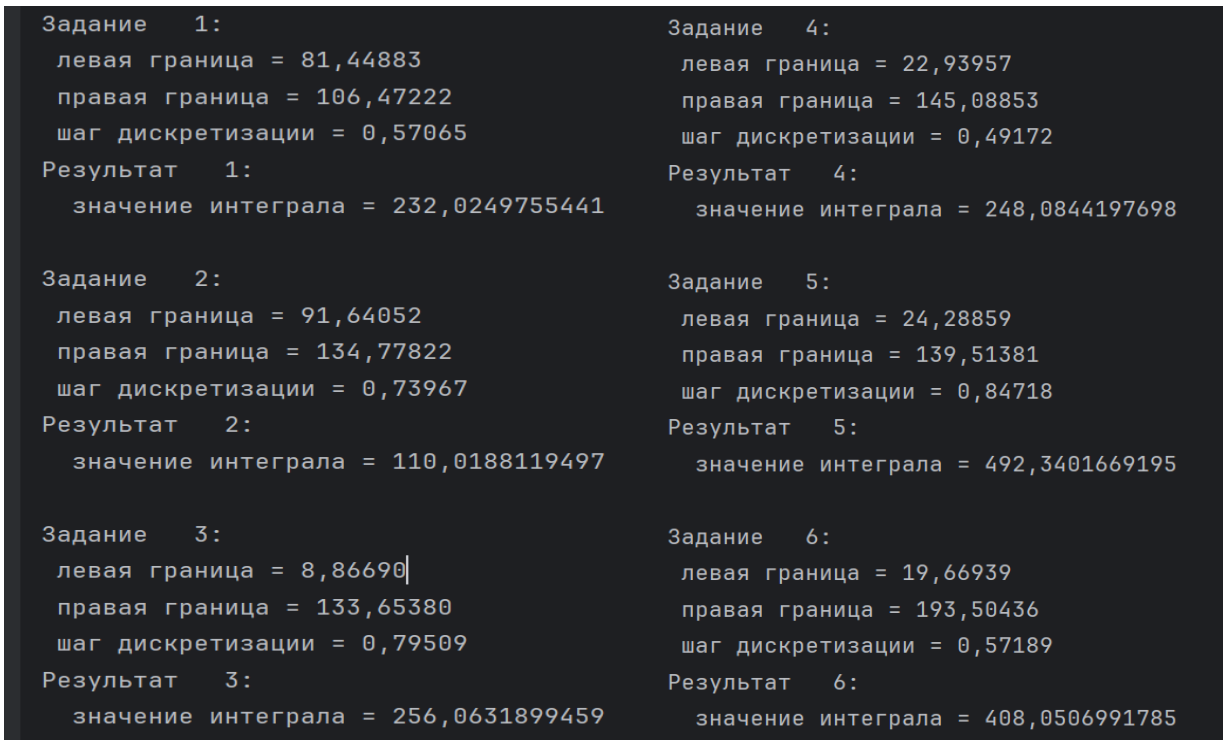
Я реализовал последовательную версию программы в методе nonThread(), где все вычисления выполняются в одном основном потоке.

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 6 и последние 5 заданий, чтобы не было избыточного количества картинок.



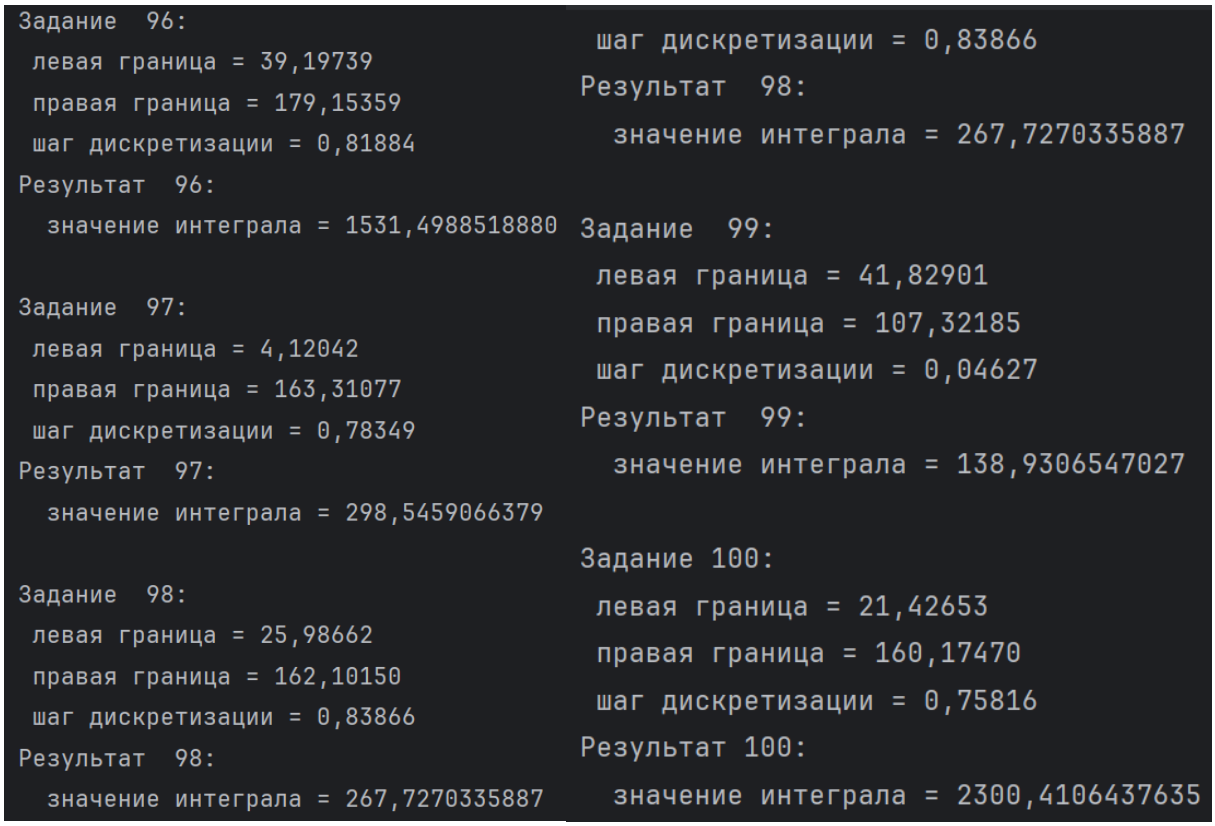


Рисунок 5,6 - вывод работы nonThread: 6 первых и 5 последних заданий

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

Проблема: Поскольку потоки работают одновременно без синхронизации, интегрирующий поток может начать читать данные до того, как генератор их полностью подготовит. Это приводит к гонке данных - интегратор может получить "смешанные" параметры от разных заданий.

```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  new *
    private final Task task;  6 usages
    private final Random random = new Random();  // Генератор случайных чисел  4 usages

    public SimpleGenerator(Task task) {  5 usages  new *
        if (task == null)
            throw new IllegalArgumentException("Task не должен быть null");
        this.task = task;}
    @Override  new *
    public void run() {
        for (int i = 0; i < task.getTasksCount(); ++i) {   // Генерируем указанное количество задач
            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();
```

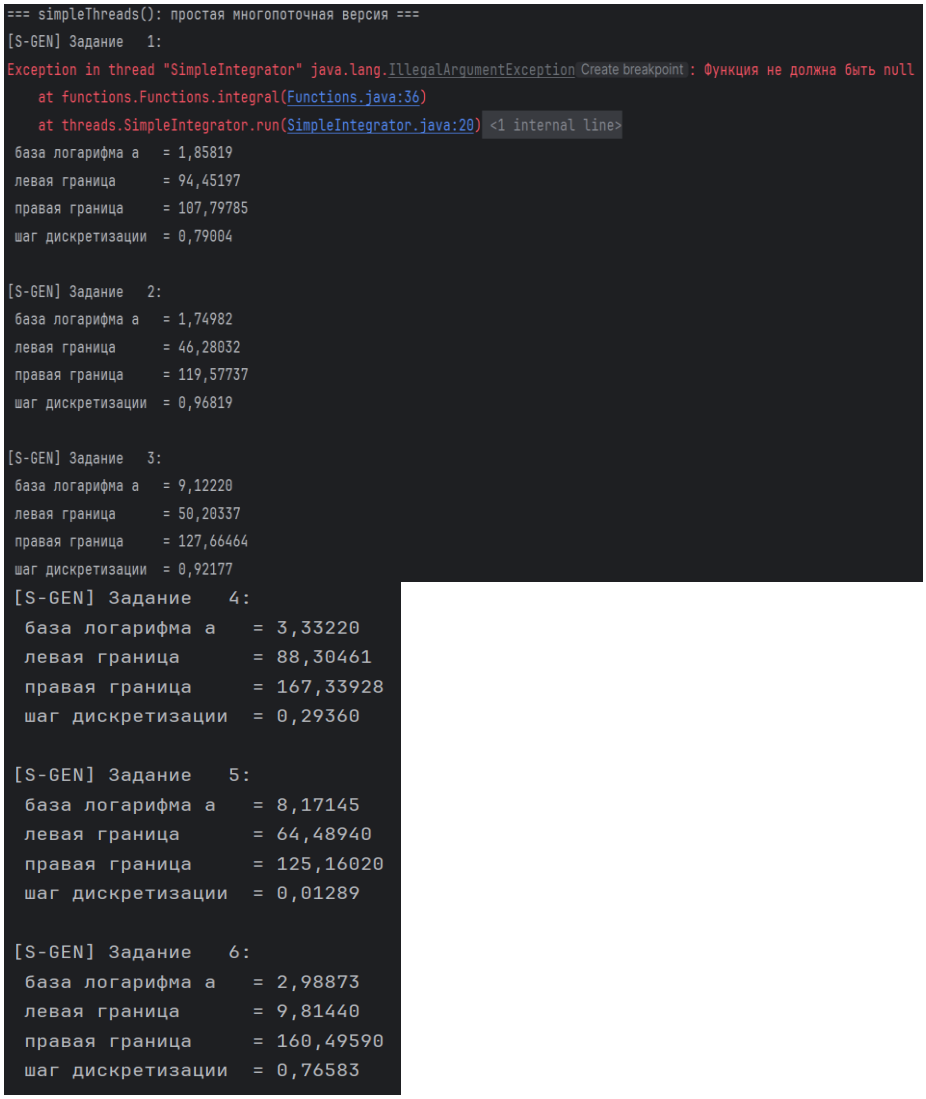
```
    double left = 100 * random.nextDouble();
    if (left <= 0)
        left = Double.MIN_VALUE; // Гарантируем положительное значение
    // Правая граница в диапазоне (100, 200), обязательно больше левой
    double right = 100 + 100 * random.nextDouble();
    if (right <= left)
        right = left + 1; // Гарантируем что right > left
    // Шаг интегрирования в диапазоне (0, 1]
    double step = random.nextDouble();
    if (step <= 0)
        step = 1; // Минимальный шаг = 1
    // Заполняем общую задачу сгенерированными параметрами
    task.setFunction(logFunction);
    task.setLeft(left);
    task.setRight(right);
    task.setStep(step);
    // Выводим информацию о сгенерированной задаче
     System.out.printf( "[S-GEN] Задание %3d:%n" + " база логарифма a    = %.5f%n" + " левая граница      = %.5f%n" +
            " правая граница     = %.5f%n" + " шаг дискретизации  = %.5f%n%n",
            taskNumber, base, left, right, step);
}
```

Рисунок 7,8 – реализация класса SimpleGenerator

```
package threads;
import functions.Function;
import functions.Functions;
// Вычислитель интеграла
public class SimpleIntegrator implements Runnable {  1 usage   new *
    private final Task task;  6 usages
    public SimpleIntegrator(Task task) {  4 usages   new *
        if (task == null)
            throw new IllegalArgumentException("Task не должен быть null");
        this.task = task;}
    @Override  new *
    public void run() {
        for (int i = 0; i < task.getTasksCount(); ++i) { // Выполняем вычисления для указанного количества задач
            int taskNumber = i + 1;
            // Читаем текущие параметры из общей задачи
            Function f = task.getFunction(); // Может быть null если генератор еще не установил функцию
            double left = task.getLeft();
            double right = task.getRight();
            double step = task.getStep();
            double value = Functions.integral(f, left, right, step); // Вычисляем интеграл с полученными параметрами
            // Выводим результат вычисления
            System.out.printf(
                    "[S-INT] Задание %3d:%n" +
                    "       левая граница      = %.5f%n" +
```

```
            int taskNumber = i + 1;
            // Читаем текущие параметры из общей задачи
            Function f = task.getFunction(); // Может быть null если генератор еще не установил функцию
            double left = task.getLeft();
            double right = task.getRight();
            double step = task.getStep();
            double value = Functions.integral(f, left, right, step); // Вычисляем интеграл с полученными параметрами
            // Выводим результат вычисления
            System.out.printf(
                    "[S-INT] Задание %3d:%n" +
                    "       левая граница      = %.5f%n" +
                    "       правая граница     = %.5f%n" +
                    "       шаг дискретизации  = %.5f%n" +
                    "       значение интеграла = %.10f%n%n",
                    taskNumber, left, right, step, value
            );
        }
    }
}
```

Рисунок 9,10 – реализация класса SimpleIntegrator



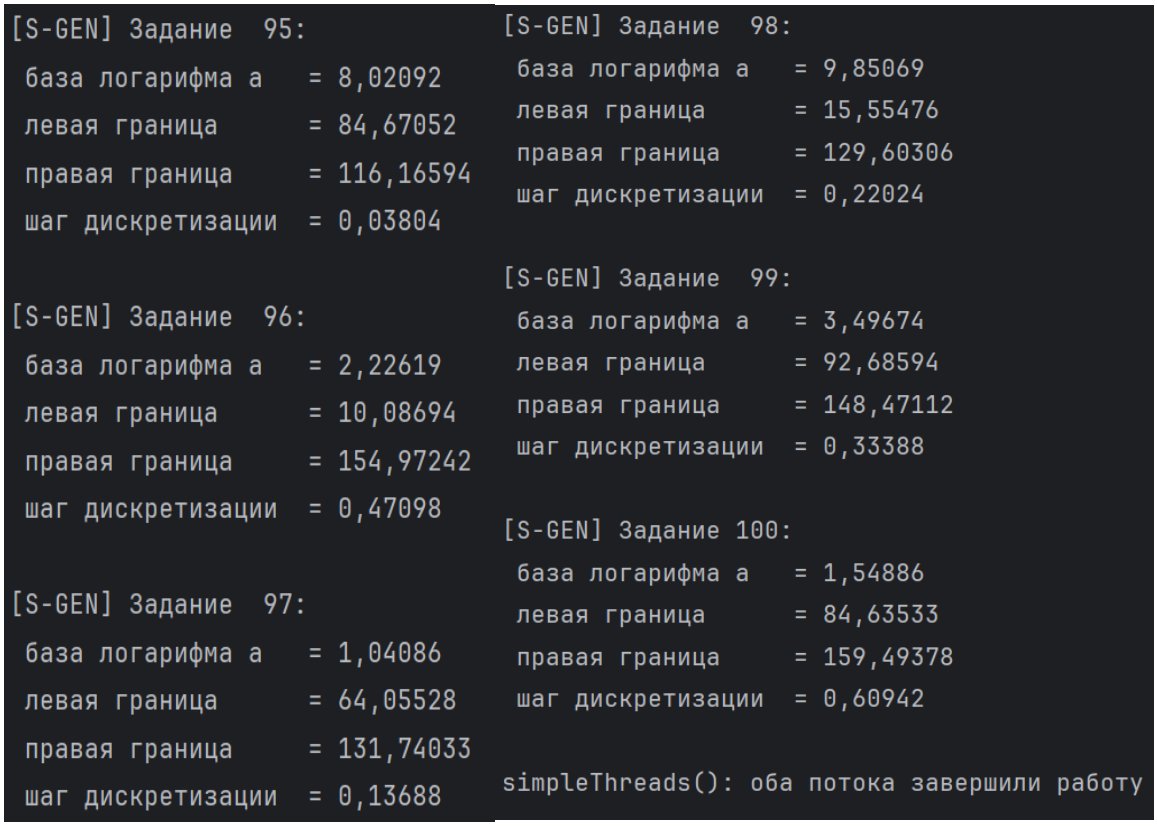


Рисунок 11.12.13 - пример консольного вывода работы simpleThreads первые и последние задания

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 14,15,16 – 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 17,18,19 – реализация класса Integrator

```
=== complicatedThreads():  многопоточная версия с семафорами ===
[GEN] Задание   1:
 левая граница = 57,211518
 правая граница = 141,534929
 шаг дискретизации = 0,184645
 основание log a    = 3,625708

[GEN] Задание   2:
 левая граница = 48,306701
 правая граница = 102,690281
 шаг дискретизации = 0,124080
 основание log a    = 6,429154

[INT] Задание   1:
 левая граница = 57,211518
 правая граница = 141,534929
 шаг дискретизации  = 0,184645
 значение интеграла = 298,989942

[GEN] Задание   3:
 левая граница = 71,242407
 правая граница = 144,560817
 шаг дискретизации = 0,541507
 основание log a    = 3,417653

[INT] Задание   2:
 левая граница = 48,306701
 правая граница = 102,690281
 шаг дискретизации  = 0,124080
 значение интеграла = 125,715143

[GEN] Задание   4:
 левая граница = 62,596320
 правая граница = 133,998062
 шаг дискретизации = 0,187870
 основание log a    = 6,220758
```

```
[INT] Задание   3:
 левая граница = 71,242407
 правая граница = 144,560817
 шаг дискретизации  = 0,541507
 значение интеграла = 278,087875

[GEN] Задание   5:
 левая граница = 25,133023
 правая граница = 115,585924
 шаг дискретизации = 0,513251
 основание log a    = 6,396589

[INT] Задание   4:
 левая граница = 62,596320
 правая граница = 133,998062
 шаг дискретизации  = 0,187870
 значение интеграла = 178,322778
```

```
[INT] Задание   5:
 левая граница = 25,133023
 правая граница = 115,585924
 шаг дискретизации  = 0,513251
 значение интеграла = 203,445897

[GEN] Задание   6:
 левая граница = 87,196503
 правая граница = 111,410957
 шаг дискретизации = 0,231241
 основание log a    = 4,018754

[GEN] Задание   7:
 левая граница = 66,702501
 правая граница = 179,452360
 шаг дискретизации = 0,100587
 основание log a    = 9,759871
```

```
[GEN] Задание  57:
 левая граница = 33,065665
 правая граница = 198,400771
 шаг дискретизации = 0,990076
 основание log a    = 2,357747

[INT] Задание  56:
 левая граница = 22,647446
 правая граница = 144,128332
 шаг дискретизации  = 0,586495
 значение интеграла = 295,175006

[GEN] Задание  58:
 левая граница = 93,179423
 правая граница = 153,945218
 шаг дискретизации = 0,283975
 основание log a    = 5,690100

[INT] Задание  59:
 левая граница = 44,224444
 правая граница = 165,704683
 шаг дискретизации  = 0,992158
 значение интеграла = 303,321130

[GEN] Задание  60:
 левая граница = 10,319823
 правая граница = 193,707666
 шаг дискретизации = 0,110158
 основание log a    = 7,009026

[GEN] Задание  61:
 левая граница = 99,759548
 правая граница = 163,709421
 шаг дискретизации = 0,882369
 основание log a    = 8,749867
```

```
[INT] Задание  57:
 левая граница = 33,065665
 правая граница = 198,400771
 шаг дискретизации  = 0,990076
 значение интеграла = 896,087449

[GEN] Задание  59:
 левая граница = 44,224444
 правая граница = 165,704683
 шаг дискретизации = 0,992158
 основание log a    = 6,288532

[INT] Задание  58:
 левая граница = 93,179423
 правая граница = 153,945218
 шаг дискретизации  = 0,283975
 значение интеграла = 167,978894

[INT] Задание  60:
 левая граница = 10,319823
 правая граница = 193,707666
 шаг дискретизации  = 0,110158
 значение интеграла = 417,346865

[INT] Задание  61:
 левая граница = 99,759548
 правая граница = 163,709421
 шаг дискретизации  = 0,882369
 значение интеграла = 143,605654

[GEN] Задание  62:
 левая граница = 77,868586
 правая граница = 147,436285
 шаг дискретизации = 0,000922
 основание log a    = 7,425893
```

```
[GEN] Задание  63:
 левая граница = 47,364117
 правая граница = 144,003507
 шаг дискретизации = 0,078250
 основание log a    = 4,866823

Generator: прерван при ожидании семафора
Generator: завершение работы потока
[INT] Задание  62:
 левая граница = 77,868586
 правая граница = 147,436285
 шаг дискретизации  = 0,000922
 значение интеграла = 163,354130

Integrator: обнаружен флаг прерывания, выходим из цикла
Integrator: завершение работы потока
complicatedThreads(): оба потока завершили работу

Process finished with exit code 0
```

Рисунок 20 - пример консольного вывода работы complicatedThreads