Федеральное государственное автономное образовательное учреждение
высшего образования «Самарский национальный исследовательский
университет имени академика С.П. Королева»

# Отчёт

## по лабораторной работе №6

Выполнил:

студент группы

6301-030301D

Панин Ю. В.

Проверил:

Борисов Д. С.

Самара, 2025

# Содержание

# Задание 1

В класс Functions был добавлен метод, реализующий нахождение интеграла функции методом трапеций на заданном отрезке с заданным шагом. Далее был преобразован класс, и запущена программа на выполнение. Вывод программы согласуется с поставленными в работе задачами:

```java
// 1. Метод интегрирования по методу трапеций
public static double integrate(Function f, double left, double right, double step) {  1 usage
    if (left < f.getLeftDomainBorder() || right > f.getRightDomainBorder()) {
        throw new IllegalArgumentException("Интервал интегрирования выходит за границы области определения функции");
    }
    if (step <= 0) {
        throw new IllegalArgumentException("Шаг интегрирования должен быть положительным");
    }

    double integral = 0.0;
    double x = left;

    while (x < right) {
        double nextX = Math.min(x + step, right);
        double y1 = f.getFunctionValue(x);
        double y2 = f.getFunctionValue(nextX);

        // Проверка на NaN
        if (Double.isNaN(y1) || Double.isNaN(y2)) {
            return Double.NaN;
        }

        integral += (y1 + y2) * (nextX - x) / 2.0;
        x = nextX;
    }

    return integral;
}
```

```java
    // 2. Метод для определения оптимального шага (для тестирования)
    public static double findOptimalStep(Function f, double left, double right, double targetError) {
        double theoretical = Math.exp(1) - 1; // Для exp: ∫₀¹ e^x dx = e - 1
        double step = 0.1;
        double error = Double.MAX_VALUE;

        while (error > targetError) {
            double numerical = integrate(f, left, right, step);
            error = Math.abs(theoretical - numerical);
            step /= 2;
            if (step < 1e-10) break;
        }

        return step * 2; // Возвращаем последний успешный шаг
    }
}
```

```
PS C:\Users\yura-\Lab-6-2025> cd src
PS C:\Users\yura-\Lab-6-2025\src> javac functions/meta/*.java
PS C:\Users\yura-\Lab-6-2025\src> javac functions/basic/*.java
PS C:\Users\yura-\Lab-6-2025\src> javac functions/*.java
PS C:\Users\yura-\Lab-6-2025\src> javac Main.java
PS C:\Users\yura-\Lab-6-2025\src> java Main
```

```
=== Тестирование задания 1: Метод интегрирования ===

Функция: f(x) = e^x
Интеграл от 0 до 1
Теоретическое значение: Int(e^x dx) на отрезке (0, 1) = e - 1 = 1,7182818285

1. Интегрирование с разными шагами:
==================================
| Шаг      | Результат      | Ошибка          | Относительная ошибка |
|----------|----------------|-----------------|----------------------|
| 1,0000   | 1,8591409142   | 0,1408590858    | 0,0819767069         |
| 0,5000   | 1,7539310925   | 0,0356492640    | 0,0207470413         |
| 0,1000   | 1,7197134914   | 0,0014316629    | 0,0008331945         |
| 0,0500   | 1,7186397889   | 0,0003579605    | 0,0002083247         |
| 0,0100   | 1,7182961475   | 0,0000143190    | 0,0000083333         |
| 0,0050   | 1,7182854082   | 0,0000035798    | 0,0000020833         |
| 0,0010   | 1,7182819716   | 0,0000001432    | 0,0000000833         |
| 0,0005   | 1,7182818643   | 0,0000000358    | 0,0000000208         |
| 0,0001   | 1,7182818299   | 0,0000000014    | 0,0000000008         |
```

```
2. Поиск оптимального шага:
============================
Целевая точность: 0,0000001000

Процесс поиска:
| Итерация | Шаг         | Результат      | Ошибка           |
|----------|-------------|----------------|------------------|
| 0        | 0,10000000 | 1,7197134914   | 0,0014316629     |
| 1        | 0,05000000 | 1,7186397889   | 0,0003579605     |
| 2        | 0,02500000 | 1,7183713214   | 0,0000894929     |
| 3        | 0,01250000 | 1,7183042019   | 0,0000223734     |
| 4        | 0,00625000 | 1,7182874218   | 0,0000055934     |
| 5        | 0,00312500 | 1,7182832268   | 0,0000013983     |
| 6        | 0,00156250 | 1,7182821780   | 0,0000003496     |

 !Достигнута точность 0,0000001000 при шаге 0,0007812500
```

```
3. Проверка граничных случаев:
================================
3.1. Интегрирование за пределами области определения (левая граница):
  Результат: 2,7205012481

3.2. Интегрирование за пределами области определения (правая граница):
  Результат: Infinity

3.3. Некорректный шаг интегрирования:
   !Поймано исключение: Шаг интегрирования должен быть положительным

3.4. Шаг интегрирования = 0:
   !Поймано исключение: Шаг интегрирования должен быть положительным

3.5. Левая граница = правой границе:
  Результат: 0,0000000000
```

```
4. Интегрирование других функций:
==================================
4.1. Int(sin(x))dx на отрезке (0,1):
  Теоретическое: 0,4596976941
  Численное:     0,4596976558
  Ошибка:        0,0000000383

4.2. Int(cos(x)dx) на отрезке (0, 1):
  Теоретическое: 0,8414709848
  Численное:     0,8414709147
  Ошибка:        0,0000000701

4.3. Int(x^2dx) на отрезке (0, 1) (проверка точности метода):
  Шаг 0,5000: результат = 0,3750000000, ошибка = 0,0416666667
  Шаг 0,1000: результат = 0,3350000000, ошибка = 0,0016666667
  Шаг 0,0100: результат = 0,3333500000, ошибка = 0,0000166667
  Шаг 0,0010: результат = 0,3333335000, ошибка = 0,0000001667
```

```
5. Проверка на нецелое количество шагов:
=======================================
5.1. Область интегрирования 0.7, шаг 0.3:
    Количество шагов: (0.7 - 0) / 0.3 = 2.333...
    Результат: 1,0200690377

5.2. Область интегрирования 1.0, шаг 0.7:
    Количество шагов: (1.0 - 0) / 0.7 = 1.428...
    Результат: 1,7646186280


6. Автоматический подбор шага для точности 1e-7:
================================================
Метод половинного деления:
| Шаг         | Результат      | Ошибка          |
|-------------|----------------|-----------------|
| 0,10000000 | 1,7197134914   | 0,0014316629    |
| 0,05000000 | 1,7186397889   | 0,0003579605    |
| 0,02500000 | 1,7183713214   | 0,0000894929    |
| 0,01250000 | 1,7183042019   | 0,0000223734    |
| 0,00625000 | 1,7182874218   | 0,0000055934    |
| 0,00312500 | 1,7182832268   | 0,0000013983    |
| 0,00156250 | 1,7182821780   | 0,0000003496    |

Итоговый шаг: 0,0015625000
Ошибка: 0,0000000874
!Требуемая точность достигнута!
```

```
7. Проверка производительности:
==============================
Время выполнения 100000 интегрирований: 1995 мс
Среднее время одного интегрирования: 0,019950 мс
```

## Задание 2

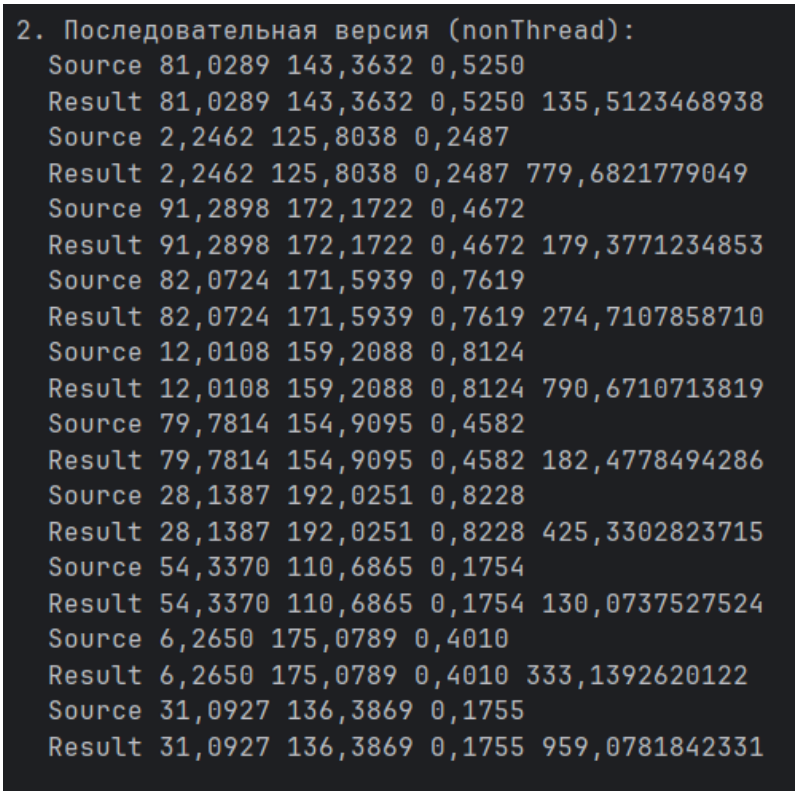
Была создана директория threads, в которой будут описываться методы, связанные с потоками. Далее в пакете threads был описан класс Task, объект которого хранит ссылку на объект интегрируемой функции, границы области интегрирования, шаг дискретизации, а также целочисленное поле, хранящее количество выполняемых заданий.

Затем в главном классе программы описан метод nonThread(), реализующий последовательную (без применения потоков инструкций) версию программы. Данный метод выполняет действия согласно заданию. Программа была запущена на выполнение, её вывод согласуется с поставленными задачами:

```java
package threads;

import functions.Function;

public class Task {  no usages
    private Function function;  2 usages
    private double left;  2 usages
    private double right;  2 usages
    private double step;  2 usages
    private int tasksCount;  2 usages
    private volatile boolean isReady = false;  3 usages
    private volatile boolean isCompleted = false;  3 usages

    // Конструктор
    public Task(int tasksCount) {  12 usages
        this.tasksCount = tasksCount;
    }

    // Геттеры и сеттеры
    public synchronized void setTask(Function function, double left, double right, double step) {
        this.function = function;
        this.left = left;
        this.right = right;
        this.step = step;
        isReady = true;
        isCompleted = false;
    }

    public synchronized void getTask() {  no usages
        isReady = false;
    }
```

```
    public Function getFunction() {
        return function;
    }

    public double getLeft() {  no usages
        return left;
    }

    public double getRight() {  no usages
        return right;
    }

    public double getStep() {  no usages
        return step;
    }

    public int getTasksCount() {  no usages
        return tasksCount;
    }

    public boolean isReady() {  no usages
        return isReady;
    }

    public boolean isCompleted() {  no usages
        return isCompleted;
    }

    public void setCompleted(boolean completed) {
        isCompleted = completed;
    }
}
```

```
private static void nonThread() {  1 usage
    int tasksCount = 10; // Уменьшено для наглядности
    Task task = new Task(tasksCount);

    for (int i = 0; i < tasksCount; i++) {
        // 1. Создаем логарифмическую функцию со случайным основанием
        double base = 1 + random.nextDouble() * 9; // от 1 до 10
        Log logFunc = new Log(base);

        // 2. Левая граница: от 0 до 100
        double left = random.nextDouble() * 100;

        // 3. Правая граница: от 100 до 200
        double right = 100 + random.nextDouble() * 100;

        // 4. Шаг дискретизации: от 0 до 1
        double step = random.nextDouble();

        // 5. Выводим сообщение
        System.out.printf("  Source %.4f %.4f %.4f%n", left, right, step);

        // 6. Вычисляем интеграл
        double result = Functions.integrate(logFunc, left, right, step);

        // 7. Выводим результат
        System.out.printf("  Result %.4f %.4f %.4f %.10f%n", left, right, step, result);
    }
}
```




## Задание 3

В пакете threads были рписаны ещё два класса: SimpleGenerator и SimpleIntegrator. 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 выводится сообщения в консоль.

В главном классе программы создан метод `simpleThreads()`. В нём создан объект задания, указано количество выполняемых заданий (минимум 100), созданы и запущены два потока вычислений, основанных на описанных классах `SimpleGenerator` и `SimpleIntegrator`. Работа метода была проверена путем вызова его в методе main

NullPointerException возникает в интегрирующем потоке, когда он пытается выполнить задание, которое либо:

- Еще не было полностью инициализировано генератором (например, task.getFunction() возвращает null).
- Было перезаписано генератором во время чтения интегрирующим потоком.
- Уже помечено как выполненное, но интегрирующий поток все еще пытается его прочитать.
- Гонка данных (Race Condition): Генератор устанавливает задание (устанавливает поля), но интегрирующий поток начинает читать до того, как все поля установлены.

Несогласованные данные (например, левая граница из одного задания, а правая граница и шаг из другого) возникают из-за состояния гонки (race condition) и неатомарности операций:

- Генератор начинает устанавливать новое задание
- После установки левой границы, но до установки правой границы и шага, происходит переключение контекста. Интегрирующий поток начинает читать задание и получает:
    - Новую левую границу (уже установлена).
    - Старую правую границу (еще не обновлена)
    - Старый шаг (еще не обновлен).
    - Результат: получается "гибридное" задание, которое никогда не генерировалось

.

SimpleGenerator:

```java
package threads;

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

public class SimpleGenerator implements Runnable {  1 usage
    private Task task;  8 usages
    private Random random;  5 usages
    private int generatedCount;  3 usages

    public SimpleGenerator(Task task) {  5 usages
        this.task = task;
        this.random = new Random();
        this.generatedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();

        while (generatedCount < tasksCount) {
            synchronized (task) {
                // Ждем, пока предыдущее задание не будет выполнено
                while (task.isReady() && !task.isCompleted()) {
                    try {
                        task.wait();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return;
                    }
                }
```

```
                // Создаем новое задание
                double base = 1 + random.nextDouble() * 9;
                Log logFunc = new Log(base);
                double left = random.nextDouble() * 100;
                double right = 100 + random.nextDouble() * 100;
                double step = random.nextDouble();

                // Устанавливаем задание
                task.setTask(logFunc, left, right, step);
                generatedCount++;

                // Выводим сообщение
                System.out.printf("Generator: Source %.4f %.4f %.4f%n", left, right, step);

                // Уведомляем интегратор
                task.notifyAll();
            }

            // Небольшая задержка для наглядности
            try {
                Thread.sleep( millis: 10);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                return;
            }
        }
    }
}
```

SimpleIntegrator:

```
package threads;

import functions.Functions;

public class SimpleIntegrator implements Runnable {  1 usage
    private Task task;  12 usages
    private int integratedCount;  3 usages

    public SimpleIntegrator(Task task) {  5 usages
        this.task = task;
        this.integratedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();

        while (integratedCount < tasksCount) {
            synchronized (task) {
                // Ждем, пока появится задание
                while (!task.isReady() || task.isCompleted()) {
                    try {
                        task.wait();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return;
                    }
                }
```

```
                // Получаем задание
                double left = task.getLeft();
                double right = task.getRight();
                double step = task.getStep();

                // Вычисляем интеграл
                double result = Functions.integrate(task.getFunction(), left, right, step);

                // Отмечаем как выполненное
                task.setCompleted(true);
                integratedCount++;

                // Выводим результат
                System.out.printf("Integrator: Result %.4f %.4f %.4f %.10f%n",
                        left, right, step, result);

                // Уведомляем генератор
                task.notifyAll();
            }

            // Небольшая задержка для наглядности
            try {
                Thread.sleep( millis: 10);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                return;
            }
        }
    }
}
```

Main:

```
    private static void simpleThreads() {  1 usage
        int tasksCount = 10; // Уменьшено для наглядности
        Task task = new Task(tasksCount);

        // Создаем потоки
        Thread generatorThread = new Thread(new SimpleGenerator(task));
        Thread integratorThread = new Thread(new SimpleIntegrator(task));

        // Устанавливаем приоритеты (опционально)
        generatorThread.setPriority(Thread.NORM_PRIORITY);
        integratorThread.setPriority(Thread.NORM_PRIORITY);

        // Запускаем потоки
        generatorThread.start();
        integratorThread.start();

        // Ждем завершения потоков
        try {
            generatorThread.join();
            integratorThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("  Все задания выполнены!");
    }
```

```
3. Простая многопоточная версия (simpleThreads):
Generator: Source 5,8290 136,4218 0,9320
Integrator: Result 5,8290 136,4218 0,9320 428,6132336692
Generator: Source 27,2290 127,0935 0,3128
Integrator: Result 27,2290 127,0935 0,3128 283,8337042180
Generator: Source 9,3635 140,2910 0,0538
Integrator: Result 9,3635 140,2910 0,0538 281,7092715216
Generator: Source 49,6925 113,5412 0,0967
Integrator: Result 49,6925 113,5412 0,0967 126,8881412651
Generator: Source 63,9485 169,1855 0,8095
Integrator: Result 63,9485 169,1855 0,8095 260,5072503377
Generator: Source 69,8478 125,5048 0,1777
Integrator: Result 69,8478 125,5048 0,1777 140,4170253681
Generator: Source 52,8440 147,8134 0,1271
Integrator: Result 52,8440 147,8134 0,1271 290,6850459708
Generator: Source 28,0859 184,4688 0,8794
Integrator: Result 28,0859 184,4688 0,8794 592,2834206631
Generator: Source 65,1839 189,0957 0,9050
Integrator: Result 65,1839 189,0957 0,9050 487,0870718053
Generator: Source 47,9570 164,7930 0,7340
Integrator: Result 47,9570 164,7930 0,7340 361,6520153356
  Все задания выполнены!
```

# Задание 4

Для решения проблемы того, что не все сгенерированные задания оказываются выполнены интегрирующим потоком, в пакет threads были добавлены два следующих класса: Generator и Integrator. Их отличие от предыдущих методов заключается в том, что для их реализации используется метод семафора.

Далее в классе Main был добавлен метод complecatedThreads, написанный для выполнения поставленных задач, и программа была запущена на выполнение. В итоге был получен вывод, показывающий правильность работы программы:

Generator:

```java
package threads;

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

public class Generator extends Thread {  4 usages
    private Task task;  3 usages
    private Semaphore semaphore;  3 usages
    private Random random;  5 usages
    private int generatedCount;  4 usages
    private volatile boolean running = true;  3 usages

    public Generator(Task task, Semaphore semaphore) {
        this.task = task;
        this.semaphore = semaphore;
        this.random = new Random();
        this.generatedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();
```

```java
        try {
            while (generatedCount < tasksCount && running) {
                semaphore.beginWrite();

                // Создаем новое задание
                double base = 1 + random.nextDouble() * 9;
                Log logFunc = new Log(base);
                double left = random.nextDouble() * 100;
                double right = 100 + random.nextDouble() * 100;
                double step = random.nextDouble();

                // Устанавливаем задание
                task.setTask(logFunc, left, right, step);
                generatedCount++;

                // Выводим сообщение
                System.out.printf("Generator[%d]: Source %.4f %.4f %.4f%n",
                        generatedCount, left, right, step);

                semaphore.endWrite();

                // Небольшая задержка
                Thread.sleep( millis: 10);
            }
        } catch (InterruptedException e) {
            System.out.println("Generator прерван");
        } finally {
            running = false;
        }
    }

    public void stopGenerator() {  1 usage
        running = false;
        this.interrupt();
    }
```

Integrator:

```java
package threads;

import functions.Functions;

public class Integrator extends Thread {  4 usages
    private Task task;  9 usages
    private Semaphore semaphore;  3 usages
    private int integratedCount;  4 usages
    private volatile boolean running = true;  3 usages

    public Integrator(Task task, Semaphore semaphore) {  6 usages
        this.task = task;
        this.semaphore = semaphore;
        this.integratedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();

        try {
            while (integratedCount < tasksCount && running) {
                semaphore.beginRead();

                // Проверяем, готово ли задание
                if (task.isReady() && !task.isCompleted()) {
                    // Получаем задание
                    double left = task.getLeft();
                    double right = task.getRight();
                    double step = task.getStep();

                    // Вычисляем интеграл
```

```
                // Вычисляем интеграл
                double result = Functions.integrate(task.getFunction(), left, right, step);

                // Отмечаем как выполненное
                task.setCompleted(true);
                integratedCount++;

                // Выводим результат
                System.out.printf("Integrator[%d]: Result %.4f %.4f %.4f %.10f%n",
                        integratedCount, left, right, step, result);
            }

            semaphore.endRead();

            // Небольшая задержка
            Thread.sleep( millis: 10);
        }
    } catch (InterruptedException e) {
        System.out.println("Integrator прерван");
    } finally {
        running = false;
    }
}

public void stopIntegrator() {  1 usage
    running = false;
    this.interrupt();
}
}
```

Main:

```
private static void complicatedThreads() {  1 usage
    int tasksCount = 10; // Уменьшено для наглядности
    Task task = new Task(tasksCount);
    Semaphore semaphore = new Semaphore();

    // Создаем потоки
    Generator generator = new Generator(task, semaphore);
    Integrator integrator = new Integrator(task, semaphore);

    // Устанавливаем приоритеты (опционально)
    generator.setPriority(Thread.NORM_PRIORITY);
    integrator.setPriority(Thread.NORM_PRIORITY);

    // Запускаем потоки
    generator.start();
    integrator.start();

    // Ждем 50 мс и прерываем
    try {
        Thread.sleep( millis: 50);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // Прерываем потоки
    System.out.println("  Прерывание потоков после 50 мс...");
    generator.stopGenerator();
    integrator.stopIntegrator();
```

```
    // Ждем завершения потоков
    try {
        generator.join();
        integrator.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("  Потоки завершены!");
}
```

Также проводим проверку потоков, изменяя приоритеты, для этого также вводим изменения в класс Main:

```java
private static void testWithDifferentPriorities() {  1 usage
    int tasksCount = 20;

    // Тест 1: Генератор с высоким приоритетом
    System.out.println("  a) Генератор с высоким приоритетом:");
    testPriority(tasksCount, Thread.MAX_PRIORITY, Thread.MIN_PRIORITY);

    // Тест 2: Интегратор с высоким приоритетом
    System.out.println("  b) Интегратор с высоким приоритетом:");
    testPriority(tasksCount, Thread.MIN_PRIORITY, Thread.MAX_PRIORITY);

    // Тест 3: Одинаковые приоритеты
    System.out.println("  c) Одинаковые приоритеты:");
    testPriority(tasksCount, Thread.NORM_PRIORITY, Thread.NORM_PRIORITY);
}
```

```java
private static void testPriority(int tasksCount, int generatorPriority, int integratorPriority)
    Task task = new Task(tasksCount);
    Semaphore semaphore = new Semaphore();

    Generator generator = new Generator(task, semaphore);
    Integrator integrator = new Integrator(task, semaphore);

    generator.setPriority(generatorPriority);
    integrator.setPriority(integratorPriority);

    long startTime = System.currentTimeMillis();

    generator.start();
    integrator.start();

    try {
        generator.join();
        integrator.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    long endTime = System.currentTimeMillis();
    System.out.printf("    Время выполнения: %d мс%n", endTime - startTime);
}
}
```

В результате получается следующий вывод, что согласуется с поставленными задачами работы:

```
4. Версия с семафором (complicatedThreads):
Generator[1]: Source 96,7765 115,7889 0,2290
Integrator[1]: Result 96,7765 115,7889 0,2290 46,2496632534
Generator[2]: Source 36,1847 179,7274 0,1242
Generator[3]: Source 0,0160 144,6029 0,0759
Integrator[2]: Result 0,0160 144,6029 0,0759 995,9477090846
Generator[4]: Source 1,2473 107,3945 0,4088
Integrator[3]: Result 1,2473 107,3945 0,4088 180,5073915985
Generator[5]: Source 91,6642 182,6307 0,5498
Integrator[4]: Result 91,6642 182,6307 0,5498 210,0004515772
  Прерывание потоков после 50 мс...
Generator прерван
Integrator прерван
  Потоки завершены!
```

```
5. Тест с разными приоритетами:
  а) Генератор с высоким приоритетом:
Generator[1]: Source 37,3274 143,3788 0,6107
Integrator[1]: Result 37,3274 143,3788 0,6107 216,5578448590
Generator[2]: Source 47,3258 148,6037 0,6517
Integrator[2]: Result 47,3258 148,6037 0,6517 423,0367411490
Generator[3]: Source 37,2686 143,8590 0,8759
Integrator[3]: Result 37,2686 143,8590 0,8759 304,5181977063
Generator[4]: Source 12,8775 170,4153 0,6610
Integrator[4]: Result 12,8775 170,4153 0,6610 346,5054095951
Generator[5]: Source 31,0688 137,3253 0,8461
Integrator[5]: Result 31,0688 137,3253 0,8461 968,9359799513
Generator[6]: Source 95,1195 156,7239 0,1656
Integrator[6]: Result 95,1195 156,7239 0,1656 181,0451674121
Generator[7]: Source 67,0213 186,1031 0,5632
Integrator[7]: Result 67,0213 186,1031 0,5632 317,0196535134
Generator[8]: Source 63,2369 137,8430 0,8858
Integrator[8]: Result 63,2369 137,8430 0,8858 153,2862823648
Generator[9]: Source 93,1389 138,2538 0,0267
Integrator[9]: Result 93,1389 138,2538 0,0267 93,9569941456
Generator[10]: Source 45,6040 118,4786 0,1897
Integrator[10]: Result 45,6040 118,4786 0,1897 583,4431160006
Generator[11]: Source 28,5490 197,8335 0,5815
Integrator[11]: Result 28,5490 197,8335 0,5815 410,0462970511
Generator[12]: Source 18,6398 176,5057 0,5030
Integrator[12]: Result 18,6398 176,5057 0,5030 1191,1377736644
Generator[13]: Source 83,9685 189,0045 0,5070
Generator[14]: Source 1,1866 182,3576 0,0173
Integrator[13]: Result 1,1866 182,3576 0,0173 334,2712495745
Generator[15]: Source 0,3973 173,7676 0,5636
Integrator[14]: Result 0,3973 173,7676 0,5636 454,2829629915
Generator[16]: Source 36,0533 111,2914 0,5753
Integrator[15]: Result 36,0533 111,2914 0,5753 151,1816673579
Generator[17]: Source 0,1707 108,0780 0,4998
Integrator[16]: Result 0,1707 108,0780 0,4998 334,7230545002
Generator[18]: Source 65,3629 145,7969 0,8630
Integrator[17]: Result 65,3629 145,7969 0,8630 256,0147803926
Generator[19]: Source 69,5129 146,1134 0,7735
Integrator[18]: Result 69,5129 146,1134 0,7735 201,8495719041
Generator[20]: Source 31,3464 190,4137 0,2864
Integrator[19]: Result 31,3464 190,4137 0,2864 408,1301975827
```

# Преобразования в связи с замечанием

Для исправления некорректной работы программы (интегратор мог пропускать данные) были сделаны следующие изменения:

1. Task.java:
    a. Добавлены два состояния dataReadyForReading и dataReadyForWriting.
    b. Реализована взаимная блокировка через wait()/notifyAll().
    c. Добавлен внутренний класс TaskData для безопасной передачи данных.
    d. Счетчики для контроля количества заданий.
2. Semaphore.java:
    a. Добавлен флаг dataProcessed для контроля обработки данных.
    b. Читатели ждут, пока данные не будут обработаны.
    c. Писатели устанавливают dataProcessed = false при записи.
    d. Интеграторы устанавливают dataProcessed = true после чтения.
3. Generator.java и Integrator.java:
    a. Используют улучшенные методы Task.setTask() и Task.getTask().
    b. Добавлены счетчики для статистики
    c. Корректная обработка прерываний..
    d. Гарантия обработки всех сгенерированных данных.

Task:

```
package threads;

import functions.Function;

public class Task {  16 usages  yuyurayu *
    private Function function;  3 usages
    private double left;  3 usages
    private double right;  3 usages
    private double step;  3 usages
    private int tasksCount;  2 usages

    // Два состояния для предотвращения пропуска данных
    private volatile boolean dataReadyForReading = false;  5 usages
    private volatile boolean dataReadyForWriting = true;  4 usages
    private volatile boolean isRunning = true;  5 usages

    // Счетчики для контроля
    private int generatedCount = 0;  1 usage
    private int processedCount = 0;  1 usage

    public Task(int tasksCount) {  15 usages  yuyurayu
        this.tasksCount = tasksCount;
    }
```

```java
// Синхронизированный метод установки задания для Generator
public synchronized void setTask(Function function, double left, double right, double step)
        throws InterruptedException {
    // Ждем, пока предыдущее задание не будет обработано
    while (!dataReadyForWriting && isRunning) {
        wait();
    }

    if (!isRunning) {
        throw new InterruptedException("Task stopped");
    }

    this.function = function;
    this.left = left;
    this.right = right;
    this.step = step;
    generatedCount++;

    dataReadyForWriting = false;
    dataReadyForReading = true;
    notifyAll();
}
```

```java
// Синхронизированный метод получения задания для Integrator
public synchronized TaskData getTask() throws InterruptedException {
    // Ждем, пока задание не будет готово
    while (!dataReadyForReading && isRunning) {
        wait();
    }

    if (!isRunning) {
        throw new InterruptedException("Task stopped");
    }

    TaskData data = new TaskData(function, left, right, step);
    processedCount++;

    dataReadyForReading = false;
    dataReadyForWriting = true;
    notifyAll();

    return data;
}

public void stop() {  1 usage  new *
    isRunning = false;
    synchronized (this) {
        notifyAll();
    }
}

public int getTasksCount() {  4 usages  new *
    return tasksCount;
}
```

```java
// Вспомогательные методы для SimpleGenerator/SimpleIntegrator
public Function getFunction() {  yuyurayu
    return function;
}

public double getLeft() {  no usages  yuyurayu
    return left;
}

public double getRight() {  no usages  yuyurayu
    return right;
}

public double getStep() {  yuyurayu
    return step;
}

public boolean isReady() {  2 usages  yuyurayu *
    return dataReadyForReading;
}

public boolean isCompleted() {  2 usages  yuyurayu *
    return !dataReadyForReading && dataReadyForWriting;
}
```

```java
    // Внутренний класс для безопасной передачи данных
    public static class TaskData {  5 usages  yuyurayu *
        public final Function function;
        public final double left;  5 usages
        public final double right;  5 usages
        public final double step;  5 usages

        public TaskData(Function function, double left, double right, double step) {
            this.function = function;
            this.left = left;
            this.right = right;
            this.step = step;
        }
    }
}
```

Semaphore:

```java
package threads;

public class Semaphore {  6 usages  yuyurayu *
    private int readers = 0;  3 usages
    private int writers = 0;  4 usages
    private int writeRequests = 0;  3 usages

    // Флаг для контроля обработки данных
    private boolean dataProcessed = true;  4 usages

    public synchronized void beginRead() throws InterruptedException {  1 usage  yuyurayu *
        // Читатель может начать, если нет писателей, запросов на запись И данные были обработаны
        while (writers > 0 || writeRequests > 0 || !dataProcessed) {
            wait();
        }
        readers++;
    }

    public synchronized void endRead() {  1 usage  yuyurayu *
        readers--;
        dataProcessed = true; // Данные обработаны
        notifyAll();
    }
```

```java
    public synchronized void beginWrite() throws InterruptedException {  1 usage  yuyurayu *
        writeRequests++;
        // Писатель может начать, если нет читателей, других писателей И данные были обработаны
        while (readers > 0 || writers > 0 || !dataProcessed) {
            wait();
        }
        writeRequests--;
        writers++;
        dataProcessed = false; // Данные еще не обработаны
    }

    public synchronized void endWrite() {  1 usage  yuyurayu
        writers--;
        notifyAll();
    }
}
```

Integrator:

```java
package threads;

import functions.Functions;

public class Integrator extends Thread {  2 usages  yuyurayu *
    private Task task;  3 usages
    private Semaphore semaphore;  3 usages
    private volatile boolean running = true;  4 usages
    private int integratedCount = 0;  6 usages

    public Integrator(Task task, Semaphore semaphore) {  5 usages  yuyurayu *
        this.task = task;
        this.semaphore = semaphore;
        setName("Integrator");
    }
```

```java
@Override  yuyurayu *
public void run() {
    try {
        while (running) {
            // Используем семафор для чтения
            semaphore.beginRead();

            try {
                // Получаем задание с гарантией, что оно не пропущено
                Task.TaskData data = task.getTask();
                integratedCount++;

                // Вычисляем интеграл
                double result = Functions.integrate(
                        data.function,
                        data.left,
                        data.right,
                        data.step
                );
```

```java
                // Выводим результат
                System.out.printf("  Integrator[%d]: Result %.4f %.4f %.4f %.10f\n",
                        integratedCount, data.left, data.right, data.step, result);

            } finally {
                semaphore.endRead();
            }

            // Проверяем, все ли задания обработаны
            if (integratedCount >= task.getTasksCount()) {
                break;
            }

            // Небольшая задержка
            if (running) {
                Thread.sleep( millis: 10);
            }
        }
```

```java
        System.out.println("  Integrator: Обработал " + integratedCount + " заданий");

    } catch (InterruptedException e) {
        System.out.println("  Integrator: Поток прерван (обработано: " + integratedCount + ")");
    } finally {
        running = false;
    }
}

public void stopIntegrator() {  1 usage  yuyurayu
    running = false;
    this.interrupt();
}

public int getIntegratedCount() {  2 usages  new *
    return integratedCount;
}
}
```

Generator:

```java
package threads;

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

public class Generator extends Thread {  2 usages  yuyurayu *
    private Task task;  4 usages
    private Semaphore semaphore;  3 usages
    private Random random;  5 usages
    private volatile boolean running = true;  4 usages
    private int generatedCount = 0;  6 usages

    public Generator(Task task, Semaphore semaphore) {  5 usages  yuyurayu *
        this.task = task;
        this.semaphore = semaphore;
        this.random = new Random();
        setName("Generator");
    }
```

```java
    @Override  yuyurayu *
    public void run() {
        int tasksCount = task.getTasksCount();

        try {
            while (generatedCount < tasksCount && running) {
                // Используем семафор для записи
                semaphore.beginWrite();

                try {
                    // Генерируем новое задание
                    double base = 1 + random.nextDouble() * 9;
                    Log logFunc = new Log(base);
                    double left = random.nextDouble() * 100;
                    double right = 100 + random.nextDouble() * 100;
                    double step = random.nextDouble();

                    // Устанавливаем задание с проверкой пропуска
                    task.setTask(logFunc, left, right, step);
                    generatedCount++;

                    // Выводим сообщение
                    System.out.printf("  Generator[%d]: Source %.4f %.4f %.4f (base=%.4f)\n",
                            generatedCount, left, right, step, base);

                } finally {
                    semaphore.endWrite();
                }
```

```java
                // Небольшая задержка для наглядности
                if (running) {
                    Thread.sleep( millis: 10);
                }
            }

            System.out.println("  Generator: Завершил генерацию " + generatedCount + " заданий");

        } catch (InterruptedException e) {
            System.out.println("  Generator: Поток прерван (сгенерировано: " + generatedCount + ")");
        } finally {
            running = false;
            task.stop();
        }
    }

    public void stopGenerator() {  1 usage   yuyurayu
        running = false;
        this.interrupt();
    }

    public int getGeneratedCount() {  2 usages   new *
        return generatedCount;
    }
}
```

В результате работы программы получается следующий вывод:

```
PS C:\Users\yura-\Lab-6-2025\src> javac Main.java
PS C:\Users\yura-\Lab-6-2025\src> java Main
=== Лабораторная работа №6: Многопоточное интегрирование ===
=== Исправленная версия с предотвращением пропуска данных ===

ЧАСТЬ 1: ТЕСТИРОВАНИЕ МЕТОДА ИНТЕГРИРОВАНИЯ

Задание 1: Реализация и тестирование метода интегрирования

Функция: f(x) = e^x
Интеграл от 0 до 1
Теоретическое значение: I(e^x dx) на отрезке (0,1) = e - 1 = 1,7182818285

Интегрирование с разными шагами:
===================================
| Шаг       | Результат      | Ошибка          |
|-----------|----------------|-----------------|
| 1,0000    | 1,8591409142   | 0,1408590858    |
| 0,5000    | 1,7539310925   | 0,0356492640    |
| 0,1000    | 1,7197134914   | 0,0014316629    |
| 0,0100    | 1,7182961475   | 0,0000143190    |
| 0,0010    | 1,7182819716   | 0,0000001432    |
```

```
ЧАСТЬ 2: ПОСЛЕДОВАТЕЛЬНАЯ ВЕРСИЯ (Задание 2)

Задание 2: Последовательная (без потоков) версия программы

Генерация и вычисление 5 задач:
================================================
  Source [ 1]: 58,0501 166,7984 0,6175 (base=2,1141)
  Result [ 1]: 58,0501 166,7984 0,6175 679,8494874662

  Source [ 2]: 32,7996 167,0065 0,1488 (base=1,6797)
  Result [ 2]: 32,7996 167,0065 0,1488 1168,5954716653

  Source [ 3]: 75,0001 162,5942 0,3622 (base=2,5071)
  Result [ 3]: 75,0001 162,5942 0,3622 453,0330173983

  Source [ 4]: 69,7793 178,5678 0,5063 (base=4,4689)
  Result [ 4]: 69,7793 178,5678 0,5063 347,8909866669

  Source [ 5]: 12,0708 117,9736 0,7902 (base=2,6352)
  Result [ 5]: 12,0708 117,9736 0,7902 440,4897437255

  Последовательная версия завершена!
```

```
ЧАСТЬ 3: ПРОСТАЯ МНОГОПОТОЧНАЯ ВЕРСИЯ (Задание 3)

Задание 3: Простая многопоточная версия (синхронизация wait/notify)

Запуск потоков: SimpleGenerator и SimpleIntegrator
Количество задач: 5
======================================================
Запуск потоков...
  SimpleGenerator[1]: Source 72,0685 174,0541 0,4756 (base=1,1235)
  SimpleIntegrator[1]: Result 72,0685 174,0541 0,4756 4186,9707556005
  SimpleGenerator[2]: Source 71,6753 111,2799 0,9011 (base=1,8224)
  SimpleIntegrator[2]: Result 71,6753 111,2799 0,9011 297,5090163253
  SimpleGenerator[3]: Source 70,4898 176,5368 0,7214 (base=4,9041)
  SimpleIntegrator[3]: Result 70,4898 176,5368 0,7214 319,0446914075
  SimpleGenerator[4]: Source 4,4545 159,5779 0,2294 (base=2,3797)
  SimpleIntegrator[4]: Result 4,4545 159,5779 0,2294 747,0675311269
  SimpleGenerator[5]: Source 78,3697 180,5870 0,9179 (base=1,3998)
  SimpleIntegrator[5]: Result 78,3697 180,5870 0,9179 1469,7680830466

  Простая многопоточная версия завершена!
Время выполнения: 71 мс
Все задания выполнены успешно.
```

```
ЧАСТЬ 4: ИСПРАВЛЕННАЯ ВЕРСИЯ С СЕМАФОРОМ (Задание 4)

Задание 4: Исправленная версия с семафором

Запуск потоков с использованием исправленного семафора:
Количество задач: 5
==================================================
Запуск потоков...
  Generator[1]: Source 8,4845 123,4827 0,9975 (base=1,6703)

Прерывание потоков через 50 мс...
  Integrator: Поток прерван (обработано: 0)
  Generator: Поток прерван (сгенерировано: 1)

  Исправленная версия с семафором завершена!
Время выполнения: 54 мс
Статистика:
  Сгенерировано заданий: 1
  Обработано заданий: 0
  Пропущено заданий: 1
```