Лабораторная n6

Задание 1

Добавьте в класс Functions метод, возвращающий значение интеграла функции, вычисленное с помощью численного метода.

```
public static double integrate(Function function, double leftLimit, double rightLimit, double step) {  3 usages
    // Проверка входных параметров
    if (step <= 0) {
        throw new IllegalArgumentException("Шаг интегрирования должен быть положительным: " + step);
    }

    if (Double.isNaN(leftLimit) || Double.isNaN(rightLimit)) {
        throw new IllegalArgumentException("Границы интегрирования не могут быть NaN");
    }

    if (leftLimit >= rightLimit) {
        throw new IllegalArgumentException(
                String.format("Левая граница (%f) должна быть меньше правой (%f)", leftLimit, rightLimit)
        );
    }

    // Проверка области определения
    if (leftLimit < function.getLeftDomainBorder() || rightLimit > function.getRightDomainBorder()) {
        throw new IllegalArgumentException(
                String.format("Интервал интегрирования [%f, %f] выходит за границы области определения [%f, %f]",
                        leftLimit, rightLimit,
                        function.getLeftDomainBorder(), function.getRightDomainBorder())
        );
    }

    double integral = 0.0;
    double currentX = leftLimit;
    double nextX;
```

Задание 2

Создайте пакет threads, в котором будут размещены классы, связанные с потоками.

```java
package threads;

import functions.Function;

public class Task {  7 usages
    private Function function;  3 usages
    private double leftBound;  4 usages
    private double rightBound;  4 usages
    private double step;  4 usages
    private int taskCount;  3 usages
    private int generatedCount;  5 usages
    private int processedCount;  5 usages

    public Task() {  1 usage
        this.generatedCount = 0;
        this.processedCount = 0;
    }

    public Task(Function function, double leftBound, double rightBound, double step) {
        this.function = function;
        this.leftBound = leftBound;
        this.rightBound = rightBound;
        this.step = step;
        this.generatedCount = 0;
        this.processedCount = 0;
    }

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

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

    public double getLeftBound() {  1 usage
        return leftBound;
    }

    public void setLeftBound(double leftBound) {  1 usage
```

```java
    public static void nonThread() {  1 usage  1eprosy
        // ... реализация nonThread ...
    }
```

Задание 3.

В пакете threads создайте два следующих класса. При их реализации воспользуйтесь

фрагментами последовательной версии программы из предыдущего задания.

```java
package threads;

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

public class SimpleGenerator implements Runnable {  2 usages
    private final Task task;  15 usages
    private final Random random;  5 usages

    public SimpleGenerator(Task task) {  1 usage
        this.task = task;
        this.random = new Random();
    }

    @Override
    public void run() {
        try {
            int taskCount = task.getTaskCount();

            for (int i = 0; i < taskCount; i++) {
                synchronized (task) {
                    // Генерация параметров
                    double base = 1 + random.nextDouble() * 9;
                    if (Math.abs(base - 1.0) < 1e-10) base = 1.1;

                    double leftBound = random.nextDouble() * 99.9 + 0.1;
                    double rightBound = 100 + random.nextDouble() * 100;

                    double step = random.nextDouble();
                    if (step < 1e-10) step = 0.01;

                    // Установка параметров
                    task.setFunction(new Log(base));
                    task.setLeftBound(leftBound);
                    task.setRightBound(rightBound);
                    task.setStep(step);
```

```java
package threads;

import functions.Function;
import functions.Functions;

public class SimpleIntegrator implements Runnable { 1 usage  1eprosy
    private final Task task;  10 usages

    public SimpleIntegrator(Task task) {  2 usages  1eprosy
        this.task = task;
    }

    @Override  1eprosy
    public void run() {
        int taskCount = task.getTaskCount();
        int processed = 0;

        while (processed < taskCount) {
            // Проверяем, не прерван ли поток
            if (Thread.currentThread().isInterrupted()) {
                return;
            }

            double leftBound = 0, rightBound = 0, step = 0;
            Function function = null;

            // Чтение параметров с блокировкой
            synchronized (task) {
                // Проверяем, есть ли новые задания для обработки
                if (task.getGeneratedCount() <= processed) {
                    // Нет новых заданий - короткая пауза
                    try {
                        task.wait( timeoutMillis: 10);
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
```

```java
public static void simpleThreads() {  1 usage  1eprosy
    // ... реализация simpleThreads ...
}

public static void complicatedThreads() {  1 usage  1eprosy
    // ... реализация complicatedThreads ...
}
```

Задание 4.

```
=== ТЕСТИРОВАНИЕ МЕТОДА ИНТЕГРИРОВАНИЯ ===
Функция: e^x (экспонента)
Отрезок: [0, 1]
Область определения: [-Infinity, Infinity]
Левая граница области определения: -Infinity
Правая граница области определения: Infinity

--- Проверка вычисления функции ---
  f(0,00) = 1,000000 (ожидается: 1,000000)
  f(0,25) = 1,284025 (ожидается: 1,284025)
  f(0,50) = 1,648721 (ожидается: 1,648721)
  f(0,75) = 2,117000 (ожидается: 2,117000)
  f(1,00) = 2,718282 (ожидается: 2,718282)

Теоретическое значение интеграла: 1,7182818285

--- Тестирование интегрирования с разными шагами ---
Шаг: 1,000000 | Результат: 1,8591409142 | Погрешность: 1,41e-01 (8,198%) | Точность: ~0 знаков
Шаг: 0,500000 | Результат: 1,7539310925 | Погрешность: 3,56e-02 (2,075%) | Точность: ~1 знаков
Шаг: 0,100000 | Результат: 1,7197134914 | Погрешность: 1,43e-03 (0,083%) | Точность: ~2 знаков
Шаг: 0,050000 | Результат: 1,7186397889 | Погрешность: 3,58e-04 (0,021%) | Точность: ~3 знаков
Шаг: 0,010000 | Результат: 1,7182961475 | Погрешность: 1,43e-05 (0,001%) | Точность: ~4 знаков
Шаг: 0,005000 | Результат: 1,7182854082 | Погрешность: 3,58e-06 (0,000%) | Точность: ~5 знаков
Шаг: 0,001000 | Результат: 1,7182819716 | Погрешность: 1,43e-07 (0,000%) | Точность: ~6 знаков
Шаг: 0,000500 | Результат: 1,7182818643 | Погрешность: 3,58e-08 (0,000%) | Точность: ~7 знаков
Шаг: 0,000100 | Результат: 1,7182818299 | Погрешность: 1,43e-09 (0,000%) | Точность: ~8 знаков
Шаг: 0,000050 | Результат: 1,7182818288 | Погрешность: 3,58e-10 (0,000%) | Точность: ~9 знаков
Шаг: 0,000010 | Результат: 1,7182818285 | Погрешность: 1,43e-11 (0,000%) | Точность: ~10 знаков
```

```
--- Поиск шага для точности 7 знака после запятой ---
Цель: погрешность < 0.0000001 (10^-7)
Это означает точность до 7 знаков после запятой
Итерация  1: Шаг = 0,01000000, Интеграл = 1,7182961475, Погрешность = 1,43e-05 (4 знаков точности) (лучший)
Итерация  2: Шаг = 0,00300000, Интеграл = 1,7182831154, Погрешность = 1,29e-06 (5 знаков точности) (лучший)
Итерация  3: Шаг = 0,00150000, Интеграл = 1,7182821504, Погрешность = 3,22e-07 (6 знаков точности) (лучший)
Итерация  4: Шаг = 0,00120000, Интеграл = 1,7182820345, Погрешность = 2,06e-07 (6 знаков точности) (лучший)
Итерация  5: Шаг = 0,00096000, Интеграл = 1,7182819603, Погрешность = 1,32e-07 (6 знаков точности) (лучший)
Итерация  6: Шаг = 0,00076800, Интеграл = 1,7182819129, Погрешность = 8,44e-08 (7 знаков точности) (лучший) ✓ ДОСТИГНУТА ЦЕЛЬ (7+ знаков)

--- РЕЗУЛЬТАТЫ ПОИСКА ---
Лучший найденный шаг: 0,000768000000
Вычисленное значение: 1,718281912908
Теоретическое значение: 1,718281828459
Достигнутая погрешность: 0,000000084448
Целевая погрешность: 0,000000100000
Достигнутая точность: ~7 знаков после запятой
✅ УСПЕХ: Найден шаг, обеспечивающий точность 7 знака!

--- АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ---
Для шага 0,00076800 потребуется:
  ~1304 вычислений функции
  ~1304 операций умножения/сложения

--- СРАВНЕНИЕ МЕТОДОВ ---
Метод прямоугольников (используемый) дает линейную сходимость
Т.е. для увеличения точности в 10 раз нужно уменьшить шаг в 10 раз
Это соответствует необходимости ~10^7 операций для 7 знаков
```

```
=== БИНАРНЫЙ ПОИСК ОПТИМАЛЬНОГО ШАГА ===
Поиск оптимального шага методом бинарного поиска...
Диапазон поиска: [0,00000001, 0,10000000]
Итерация  1: Шаг = 0,0500000050, Погрешность = 3,58e-04 (3 знаков) (лучший) ×
Итерация  2: Шаг = 0,0750000025, Погрешность = 7,77e-04 (3 знаков) ×
Итерация  3: Шаг = 0,0875000013, Погрешность = 1,04e-03 (2 знаков) ×
Итерация  4: Шаг = 0,0937500006, Погрешность = 1,19e-03 (2 знаков) ×
Итерация  5: Шаг = 0,0968750003, Погрешность = 1,29e-03 (2 знаков) ×
Итерация  6: Шаг = 0,0984375002, Погрешность = 1,35e-03 (2 знаков) ×
Итерация  7: Шаг = 0,0992187501, Погрешность = 1,39e-03 (2 знаков) ×
Итерация  8: Шаг = 0,0996093750, Погрешность = 1,41e-03 (2 знаков) ×
Итерация  9: Шаг = 0,0998046875, Погрешность = 1,42e-03 (2 знаков) ×
Итерация 10: Шаг = 0,0999023438, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 11: Шаг = 0,0999511719, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 12: Шаг = 0,0999755859, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 13: Шаг = 0,0999877930, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 14: Шаг = 0,0999938965, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 15: Шаг = 0,0999969482, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 16: Шаг = 0,0999984741, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 17: Шаг = 0,0999992371, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 18: Шаг = 0,0999996185, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 19: Шаг = 0,0999998093, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 20: Шаг = 0,0999999046, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 21: Шаг = 0,0999999523, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 22: Шаг = 0,0999999762, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 23: Шаг = 0,0999999881, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 24: Шаг = 0,0999999940, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 25: Шаг = 0,0999999970, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 26: Шаг = 0,0999999985, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 27: Шаг = 0,0999999993, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 28: Шаг = 0,0999999996, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 29: Шаг = 0,0999999998, Погрешность = 1,43e-03 (2 знаков) ×
Итерация 30: Шаг = 0,0999999999, Погрешность = 1,43e-03 (2 знаков) ×
```

```
--- РЕЗУЛЬТАТ БИНАРНОГО ПОИСКА ---
Оптимальный шаг: 0,050000005000
Вычисленное значение: 1,718639788886
Теоретическое значение: 1,718281828459
Достигнутая погрешность: 0,000357960427
Достигнутая точность: 3 знаков после запятой
⚠ Лучшая достигнутая точность: 3 знаков

--- ПРАКТИЧЕСКИЕ РЕКОМЕНДАЦИИ ---
Для большинства практических задач:
1. Шаг 0.001 дает точность ~3 знака
2. Шаг 0.0001 дает точность ~4 знака
3. Шаг 0.00001 дает точность ~5 знака
4. Для 7 знаков нужен шаг ~0,05000001

⚠ ВАЖНО: Очень маленький шаг может вызвать:
  - Накопление ошибок округления
  - Большое время вычислений
  - Проблемы с памятью
Рекомендуется найти баланс между точностью и производительностью

=== ТОЧНЫЙ РАСЧЕТ ДЛЯ ДЕМОНСТРАЦИИ ===
Демонстрация расчета с шагом 1.0E-6:
Вычисленное значение: 1,718281828459
Теоретическое значение: 1,718281828459
Абсолютная погрешность: 1,83e-13
Относительная погрешность: 0,0000%
Точность: 12 знаков после запятой
Время вычисления: 17,415 мс
Количество операций: ~1000000
Скорость: ~57423 операций/мс
```

```
============================================================
ПЕРЕХОД К МНОГОПОТОЧНЫМ ТЕСТАМ
============================================================



=== ИТОГОВАЯ СТАТИСТИКА ===
1. nonThread (последовательная):
   Время: 0,00 сек
   Скорость: Infinity задач/сек

2. simpleThreads (простая многопоточная):
   Время: 0,00 сек
   Скорость: Infinity задач/сек
   Ускорение: NaN%

3. complicatedThreads (сложная многопоточная):
   Время: 0,00 сек
   Скорость: Infinity задач/сек
   Ускорение: NaN%

Общее время выполнения всех тестов: 2.001 сек

=== РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ ИНТЕГРИРОВАНИЯ ===
Для функции e^x на отрезке [0, 1]:
Теоретическое значение интеграла: 1.718281828459045

Для достижения точности 7 знаков после запятой:
Рекомендуемый шаг дискретизации: ~0.000001 (10^-6)
Это даст погрешность < 0.0000001 (10^-7)

Примечание: фактический шаг зависит от используемого
метода интегрирования (прямоугольники, трапеции и т.д.)

=== ПРОГРАММА ЗАВЕРШЕНА ===

Process finished with exit code 0
```