Лабораторная работа №6

Выполнил: Магера Никита Алексеевич

Студент группы 6203-010302D

Ход выполнения

Задание 1

В 1 задании необходимо в классе FunctionPoint добавить метод Integral, реализующий численное интегрирование методом трапеций. Метод должен принимать функцию, границы интегрирования и шаг дискретизации, проверять корректность интервала и вычислять интеграл. Затем протестировать метод на функции e^x от 0 до 1, определить шаг для точности 7 знаков после запятой. Теоретическое значение: e-1 ≈ 1.718281828. Экспериментально установлено, что для достижения точности в 7 знаке после запятой требуется шаг дискретизации порядка 0.001. (Рис.1-2)

```
public static double Integral(Function function, double left, double right, double step) {  1 usage  new *
    // не выходим ли за границы области определения
    if (left < function.getLeftDomainBorder() || right > function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Интервал интегрирования выходит за границы области определения функции");
    }
    // вычисление интеграла методом трапеций
    double integral = 0.0;
    double need_x = left;
    double need_y = function.getFunctionValue(need_x);
    // идем пока не дойдем до правой границы
    while (need_x < right) {
        // следующая точка
        double next_x = need_x + step;
        if (next_x > right) {
            next_x = right;
        }
        // значение функции в следующей точке
        double next_y = function.getFunctionValue(next_x);
        // площадь трапеции: (основание1 + основание2) * высота / 2
        double trapezoid = (need_y + next_y) * (next_x - need_x) / 2.0;
        integral += trapezoid;
        // переходим к следующему отрезку
        need_x = next_x;
        need_y = next_y;
    }
    return integral;
}
```

Рис. 1

```
проверка интегрирования
теоретическое значение ∫e^x dx от 0 до 1 = 1.718281828459045
Шаг: 1.0  Результат: 1.8591409142295225, Ошибка: 0.14085908577047745
не удовлетворяет условию
Шаг: 0.1  Результат: 1.7197134913893144, Ошибка: 0.0014316629302693062
не удовлетворяет условию
Шаг: 0.01  Результат: 1.7182961474504181, Ошибка: 1.431899137305237E-5
не удовлетворяет условию
Шаг: 0.001  Результат: 1.7182819716491948, Ошибка: 1.4319014973729338E-7
не удовлетворяет условию
Шаг: 1.0E-4  Результат: 1.7182818298909435, Ошибка: 1.4318983776462346E-9
7 знаков, результат удовлетворяет условию
```

Рис. 2

Задание 2

В 2 задании нужно создать класс `Task` в пакете `threads`, который будет хранить параметры задания для вычисления интеграла. Класс должен содержать поля для функции (логарифм), левой и правой границ интегрирования, шага и результата вычислений. Функция-логарифм генерируется со случайным основанием от 1 до 10, левая граница — от 0 до 100, правая граница — от 100 до 200, шаг — от 0 до 1. Затем реализуется метод `nonThread()`, который последовательно выполняет 100 таких заданий. Для каждого задания метод вычисляет определенный интеграл функции логарифма на заданном интервале с указанным шагом, после чего выводит параметры задания в формате "Source <левая граница> <правая граница> <шаг>" и результат в формате "Result <левая граница> <правая граница> <шаг> <значение интеграла>". Все вычисления выполняются в одном потоке, что позволяет оценить базовое время выполнения без использования многопоточности.(Рис.3-4)

```
// последовательная версия программы
public static void nonThread() {  1 usage  new *
    //объект Task с 100 заданиями
    Task task = new Task( tasksCount: 100);
    for (int i = 0; i < task.getTasksCount(); i++) {
        // логарифм со случайным основанием от 1 до 10
        double log = 1 + Math.random() * 9;
        task.setFunction(new functions.basic.Log(log));
        // левая граница: 0 - 100
        task.setLeftX(Math.random() * 100);
        // правая граница: 100 - 200
        task.setRightX(100 + Math.random() * 100);
        // шаг 0 1
        task.setStep(Math.random());
        // вывод
        System.out.println("Source " + task.getLeftX() + " " + task.getRightX() + " " + task.getStep());
        try {
            // вычисление интеграла
            double integral = Functions.Integral(task.getFunction(), task.getLeftX(), task.getRightX(), task.getStep());
            System.out.println("Result " + task.getLeftX() + " " + task.getRightX() + " " + task.getStep() + " " + integral);

        } catch (Exception e) {
            System.out.println("ошибка интегрирования: " + e.getMessage());
        }
    }
    System.out.println("Выполнено " + task.getTasksCount() + " заданий.");
}
```

Рис.3

```
Result 44.23974750196164 175.6570664456022 0.7659439871056373 297.93316158790276
Source 69.59979750774976 131.59885926414213 0.702045436341746
Result 69.59979750774976 131.59885926414213 0.702045436341746 177.46363347279345
Source 57.216444301984204 146.67211601196257 0.8816765190957578
Result 57.216444301984204 146.67211601196257 0.8816765190957578 183.76451468262871
Source 10.679198694298853 136.94596743066978 0.5352139837067169
Result 10.679198694298853 136.94596743066978 0.5352139837067169 236.05528044978536
Source 51.94376785741108 110.30059204872232 0.850693034664562
Result 51.94376785741108 110.30059204872232 0.850693034664562 201.56148310269438
Source 37.65462764127958 184.82108533950338 0.8581333169986722
Result 37.65462764127958 184.82108533950338 0.8581333169986722 426.3963989671267
Source 26.95159207457083 112.08493045969523 0.7280123505416273
Result 26.95159207457083 112.08493045969523 0.7280123505416273 488.9372971629669
Source 34.74784681068738 181.83536083901845 0.0039362577420447575
Result 34.74784681068738 181.83536083901845 0.0039362577420447575 1298.2114330300767
Source 16.283229194597236 175.25559450266348 0.04834721584055235
Result 16.283229194597236 175.25559450266348 0.04834721584055235 524.3053018549571
Source 93.69804265073849 102.90653832526519 0.770145495126178
Result 93.69804265073849 102.90653832526519 0.770145495126178 77.79580832265432
Source 53.439008731863034 110.36327880340431 0.5243933952262243
Result 53.439008731863034 110.36327880340431 0.5243933952262243 254.91027599711018
Source 14.500444051695672 159.68326193424872 0.9268076241546478
Result 14.500444051695672 159.68326193424872 0.9268076241546478 286.1850744698833
Source 91.82009150656752 198.06175681965158 0.20953500959985594
Result 91.82009150656752 198.06175681965158 0.20953500959985594 268.44221928540196
Source 56.59049793317321 106.37786603770137 0.6597465410793324
Result 56.59049793317321 106.37786603770137 0.6597465410793324 111.07422905955643
Source 25.813170762370685 179.80697573620625 0.6288726812190684
Result 25.813170762370685 179.80697573620625 0.6288726812190684 368.38215061729227
Source 97.11423402351043 130.27319835023795 0.5778428864601848
Result 97.11423402351043 130.27319835023795 0.5778428864601848 80.80561776177751
Source 81.65664120692135 116.23804039784142 0.63248076567469
Result 81.65664120692135 116.23804039784142 0.63248076567469 192.84060064346139
Source 77.92542742247424 199.16915841899765 0.16324554446353845
Result 77.92542742247424 199.16915841899765 0.16324554446353845 269.2263776337073
Source 72.22892298390063 198.00571633824734 0.9747997631561297
Result 72.22892298390063 198.00571633824734 0.9747997631561297 374.20945358296217
Source 47.95019771340237 163.0345646039474 0.7934216130684173
Result 47.95019771340237 163.0345646039474 0.7934216130684173 331.23210048886904
Source 34.899486396699494 118.81404955323694 0.8740702882845427
Result 34.899486396699494 118.81404955323694 0.8740702882845427 184.78515433794738
Source 32.559656941338 132.12995557897008 0.7139508314419414
Result 32.559656941338 132.12995557897008 0.7139508314419414 403.22090403529063
Source 96.58328450891915 145.4407606436056 0.055714713297390084
Result 96.58328450891915 145.4407606436056 0.055714713297390084 213.22541034986796
Source 86.59125043867424 178.86696851917685 0.590971095220135
Result 86.59125043867424 178.86696851917685 0.590971095220135 236.84464692717998
```

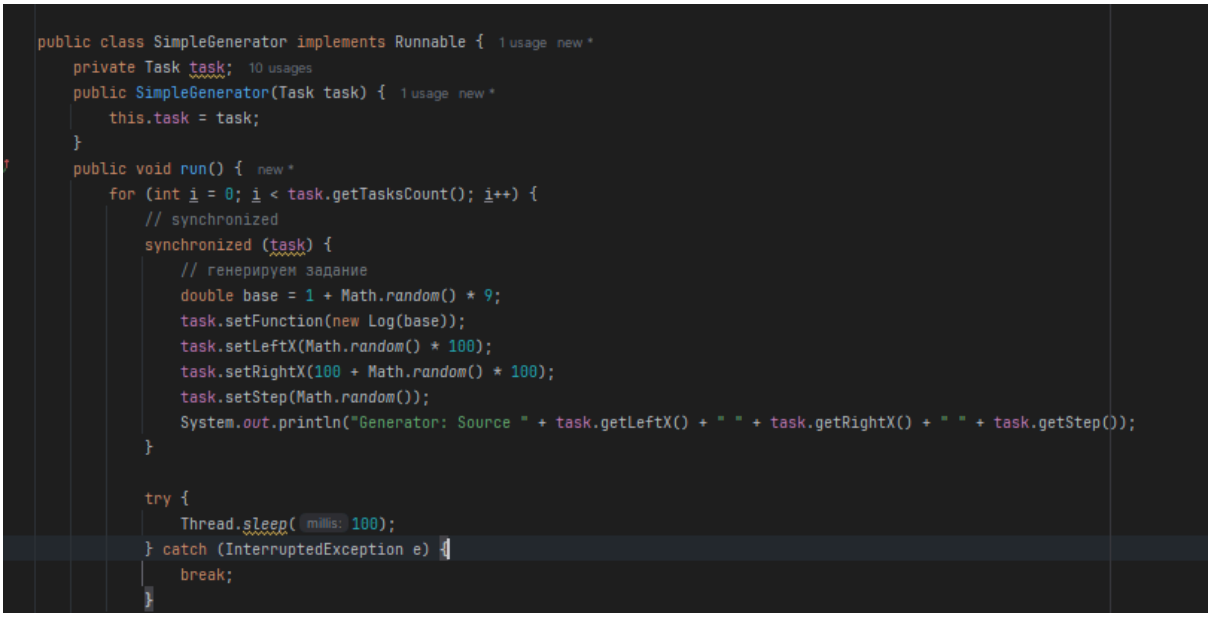
Рис.4(и так 100 раз)

Задание 3

В третьем задании должны быть созданы классы SimpleGenerator и SimpleIntegrator, реализующие интерфейс Runnable. SimpleGenerator формирует задания в цикле, занося параметры в объект Task и выводя сообщения "Source". SimpleIntegrator в цикле извлекает данные из Task, вычисляет интегралы через Functions.Integral и выводит "Result". При тестировании выявлены проблемы многопоточности: NullPointerException возникал при попытке интегратора получить данные до их генерации, также наблюдалось смешивание параметров из разных заданий. Для устранения добавлена проверка на null с ожиданием и блоки синхронизации synchronized(task) в методах run(). Проведены эксперименты с приоритетами потоков: при высоком приоритете генератора задания создавались быстрее, чем обрабатывались; при высоком приоритете интегратора учащались попытки чтения неготовых данных. После синхронизации исключения исчезли, данные передаются целостно, потоки корректно завершают все 100 заданий.(Рис.5-7)

```
public class SimpleIntegrator implements Runnable {  1 usage  new *
    private Task task;  8 usages
    public SimpleIntegrator(Task task) {  1 usage  new *
        this.task = task;
    }

    public void run() {  new *
        for (int i = 0; i < task.getTasksCount(); i++) {
            // synchronized
            synchronized (task) {
                try {
                    if (task.getFunction() == null) {
                        Thread.sleep( millis: 30);
                        continue;
                    }
                    // берем текущие данные
                    double leftX = task.getLeftX();
                    double rightX = task.getRightX();
                    double step = task.getStep();

                    // вычисляем интеграл
                    double integral = Functions.Integral(task.getFunction(), leftX, rightX, step);
                    // выводим результат
                    System.out.println("Integrator: Result " + leftX + " " + rightX + " " + step + " " + integral);

                } catch (Exception e) {
                    System.out.println("Integrator error: " + e.getMessage());
                }
            }

            try {
                Thread.sleep( millis: 30);
            } catch (InterruptedException e) {
                break;
            }
```

Рис.5



```
public class SimpleGenerator implements Runnable {  1 usage  new *
    private Task task;  10 usages
    public SimpleGenerator(Task task) {  1 usage  new *
        this.task = task;
    }
    public void run() {  new *
        for (int i = 0; i < task.getTasksCount(); i++) {
            // synchronized
            synchronized (task) {
                // генерируем задание
                double base = 1 + Math.random() * 9;
                task.setFunction(new Log(base));
                task.setLeftX(Math.random() * 100);
                task.setRightX(100 + Math.random() * 100);
                task.setStep(Math.random());
                System.out.println("Generator: Source " + task.getLeftX() + " " + task.getRightX() + " " + task.getStep());
            }

            try {
                Thread.sleep( millis: 100);
            } catch (InterruptedException e) {
                break;
            }
```

Рис.6

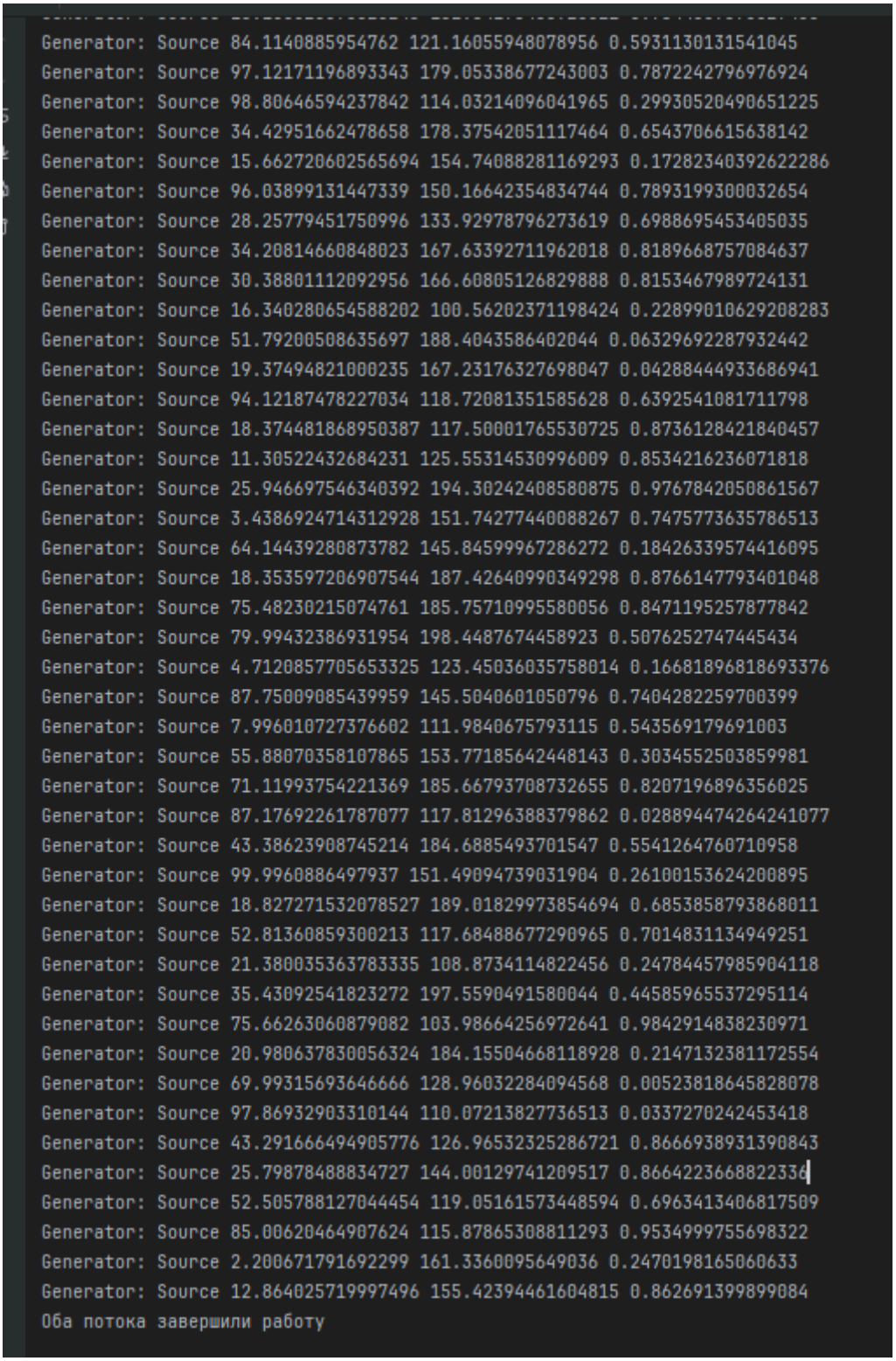

Generator: Source 84.1140885954762 121.16055948078956 0.5931130131541045
Generator: Source 97.12171196893343 179.05338677243003 0.7872242796976924
Generator: Source 98.80646594237842 114.03214096041965 0.29930520490651225
Generator: Source 34.42951662478658 178.37542051117464 0.6543706615638142
Generator: Source 15.662720602565694 154.74088281169293 0.17282340392622286
Generator: Source 96.03899131447339 150.16642354834744 0.7893199300032654
Generator: Source 28.25779451750996 133.92978796273619 0.6988695453405035
Generator: Source 34.20814660848023 167.63392711962018 0.8189668757084637
Generator: Source 30.38801112092956 166.60805126829888 0.8153467989724131
Generator: Source 16.340280654588202 100.56202371198424 0.22899010629208283
Generator: Source 51.79200508635697 188.4043586402044 0.06329692287932442
Generator: Source 19.37494821000235 167.23176327698047 0.04288444933686941
Generator: Source 94.12187478227034 118.72081351585628 0.6392541081711798
Generator: Source 18.374481868950387 117.50001765530725 0.8736128421840457
Generator: Source 11.30522432684231 125.55314530996009 0.8534216236071818
Generator: Source 25.946697546340392 194.30242408580875 0.9767842050861567
Generator: Source 3.4386924714312928 151.74277440088267 0.7475773635786513
Generator: Source 64.14439280873782 145.84599967286272 0.18426339574416095
Generator: Source 18.353597206907544 187.42640990349298 0.8766147793401048
Generator: Source 75.48230215074761 185.75710995580056 0.8471195257877842
Generator: Source 79.99432386931954 198.4487674458923 0.5076252747445434
Generator: Source 4.7120857705653325 123.45036035758014 0.16681896818693376
Generator: Source 87.75009085439959 145.5040601050796 0.7404282259700399
Generator: Source 7.996010727376602 111.9840675793115 0.543569179691003
Generator: Source 55.88070358107865 153.77185642448143 0.3034552503859981
Generator: Source 71.11993754221369 185.66793708732655 0.8207196896356025
Generator: Source 87.17692261787077 117.81296388379862 0.028894474264241077
Generator: Source 43.38623908745214 184.6885493701547 0.5541264760710958
Generator: Source 99.9960886497937 151.49094739031904 0.26100153624200895
Generator: Source 18.827271532078527 189.01829973854694 0.6853858793868011
Generator: Source 52.81360859300213 117.68488677290965 0.7014831134949251
Generator: Source 21.380035363783335 108.8734114822456 0.24784457985904118
Generator: Source 35.43092541823272 197.5590491580044 0.44585965537295114
Generator: Source 75.66263060879082 103.98664256972641 0.9842914838230971
Generator: Source 20.980637830056324 184.15504668118928 0.2147132381172554
Generator: Source 69.99315693646666 128.96032284094568 0.00523818645828078
Generator: Source 97.86932903310144 110.07213827736513 0.0337270242453418
Generator: Source 43.291666494905776 126.96532325286721 0.8666938931390843
Generator: Source 25.79878488834727 144.00129741209517 0.8664223668822336
Generator: Source 52.505788127044454 119.05161573448594 0.6963413406817509
Generator: Source 85.00620464907624 115.87865308811293 0.9534999755698322
Generator: Source 2.200671791692299 161.3360095649036 0.2470198165060633
Generator: Source 12.864025719997496 155.42394461604815 0.862691399899084
Оба потока завершили работу

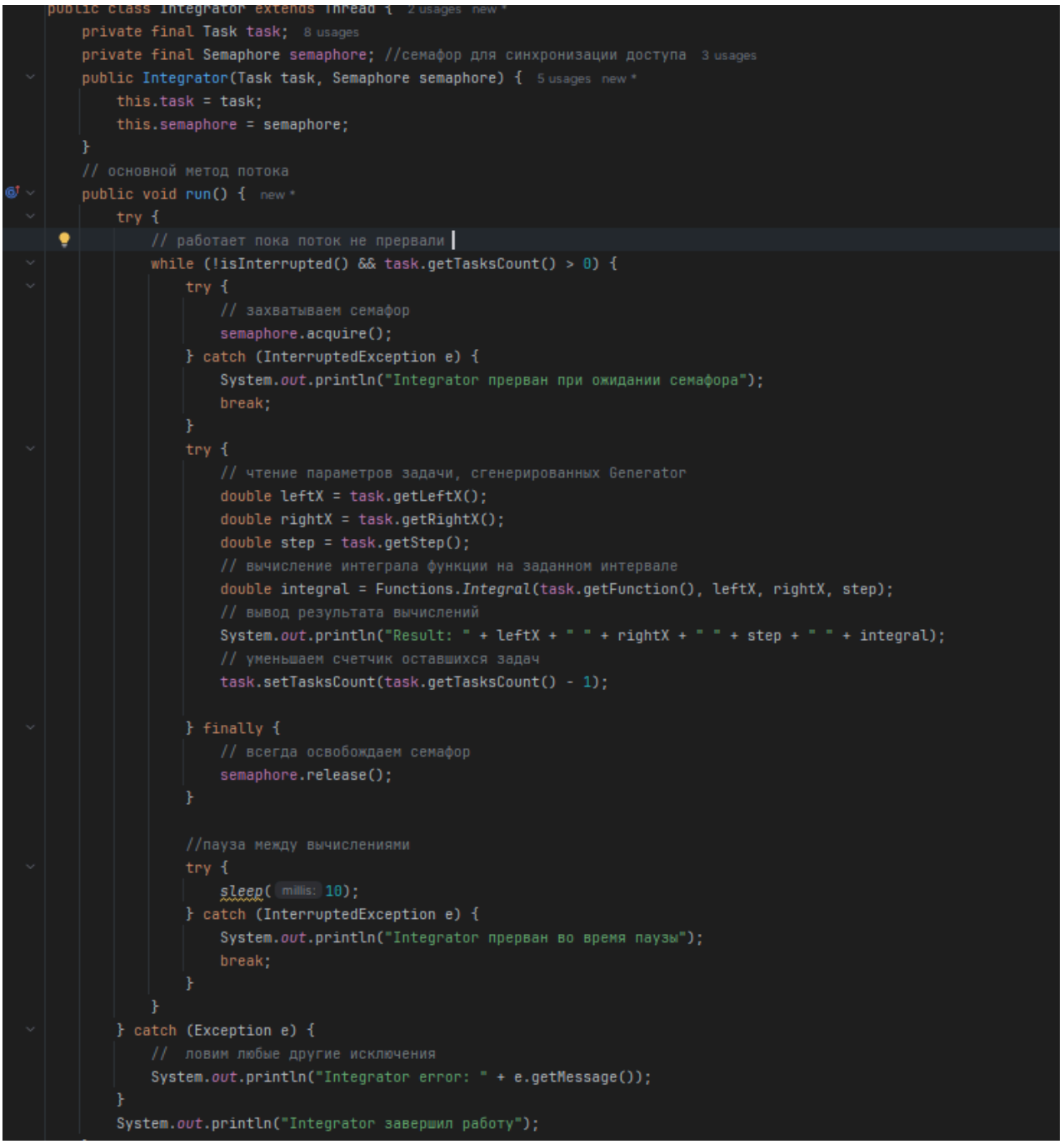
Рис.7

Задание 4

В 4 задании реализованы два потока Generator и Integrator, использующие общий семафор java.util.concurrent.Semaphore для синхронизации доступа к объекту Task. Generator генерирует случайные параметры задач, Integrator вычисляет интегралы. Семафор с одним разрешением обеспечивает взаимоисключающий доступ: пока один поток работает с данными, второй ожидает. Добавлена корректная обработка прерывания потоков через проверку isInterrupted() и обработку InterruptedException. Программа отрабатывает минимум 100 заданий с гарантированной синхронизацией без потерь данных.(Рис.8-10)

```java
public Generator(Task task, Semaphore semaphore) {  5 usages  new *
    this.task = task;
    this.semaphore = semaphore;
}

// основной метод потока
public void run() {  new *
    Random random = new Random(); // генератор случайных чисел

    try {
        // Работаем пока поток не прервали и есть задачи для обработки
        while (!isInterrupted() && task.getTasksCount() > 0) {
            try {
                // захватываем семафор
                semaphore.acquire();
            } catch (InterruptedException e) {
                System.out.println("Generator прерван при ожидании семафора");
                break;
            }
            try {
                // генерация случайных параметров для задачи:
                double a = random.nextDouble() * 9 + 1;
                task.setFunction(new Log(a));
                task.setLeftX(random.nextDouble() * 100);
                task.setRightX(100 + random.nextDouble() * 100);
                task.setStep(random.nextDouble());

                System.out.println("Source: " + task.getLeftX() + " " + task.getRightX() + " " + task.getStep());

            } finally {
                // всегда освобождаем семафор
                semaphore.release();
            }

            // пауза между генерациями
            try {
                sleep( millis: 10);
            } catch (InterruptedException e) {
                System.out.println("Generator прерван во время паузы"); break;
            }
        }
    } catch (Exception e) {
        // ловим любые другие исключения
        System.out.println("Generator: " + e.getMessage());
    }

    System.out.println("Generator завершил работу");
}
```

Рис.8



```
public class Integrator extends Thread {  2 usages  new *
    private final Task task;  8 usages
    private final Semaphore semaphore; //семафор для синхронизации доступа  3 usages
    public Integrator(Task task, Semaphore semaphore) {  5 usages  new *
        this.task = task;
        this.semaphore = semaphore;
    }
    // основной метод потока
    public void run() {  new *
        try {
            // работает пока поток не прервали
            while (!isInterrupted() && task.getTasksCount() > 0) {
                try {
                    // захватываем семафор
                    semaphore.acquire();
                } catch (InterruptedException e) {
                    System.out.println("Integrator прерван при ожидании семафора");
                    break;
                }
                try {
                    // чтение параметров задачи, сгенерированных Generator
                    double leftX = task.getLeftX();
                    double rightX = task.getRightX();
                    double step = task.getStep();
                    // вычисление интеграла функции на заданном интервале
                    double integral = Functions.Integral(task.getFunction(), leftX, rightX, step);
                    // вывод результата вычислений
                    System.out.println("Result: " + leftX + " " + rightX + " " + step + " " + integral);
                    // уменьшаем счетчик оставшихся задач
                    task.setTasksCount(task.getTasksCount() - 1);

                } finally {
                    // всегда освобождаем семафор
                    semaphore.release();
                }

                //пауза между вычислениями
                try {
                    sleep( millis: 10);
                } catch (InterruptedException e) {
                    System.out.println("Integrator прерван во время паузы");
                    break;
                }
            }
        } catch (Exception e) {
            //  ловим любые другие исключения
            System.out.println("Integrator error: " + e.getMessage());
        }
        System.out.println("Integrator завершил работу");
```

Рис.9

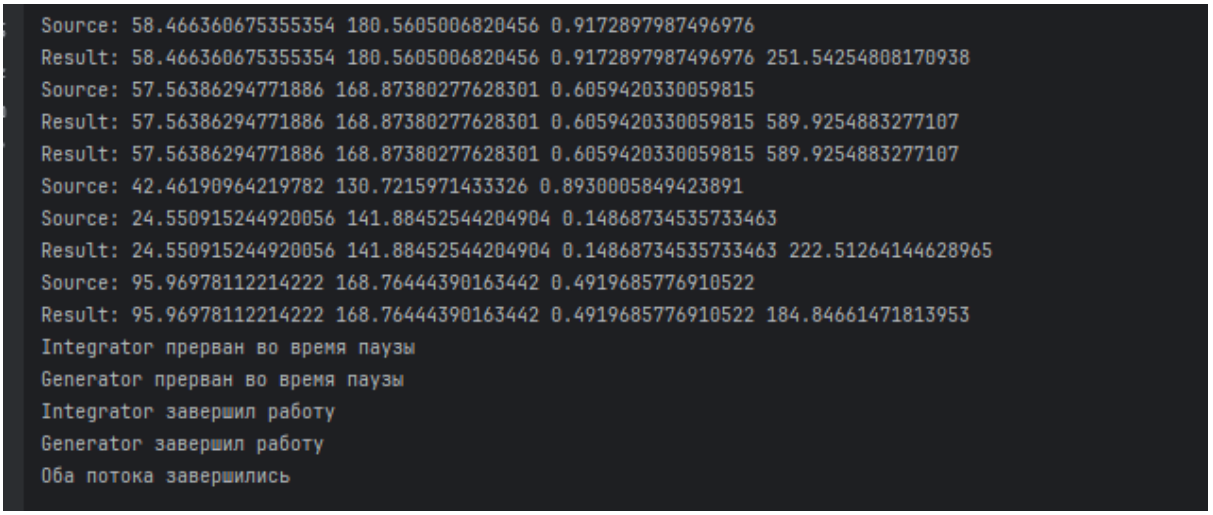

Source: 58.466360675355354 180.5605006820456 0.9172897987496976
Result: 58.466360675355354 180.5605006820456 0.9172897987496976 251.54254808170938
Source: 57.56386294771886 168.87380277628301 0.6059420330059815
Result: 57.56386294771886 168.87380277628301 0.6059420330059815 589.9254883277107
Result: 57.56386294771886 168.87380277628301 0.6059420330059815 589.9254883277107
Source: 42.46190964219782 130.7215971433326 0.8930005849423891
Source: 24.550915244920056 141.88452544204904 0.14868734535733463
Result: 24.550915244920056 141.88452544204904 0.14868734535733463 222.51264144628965
Source: 95.96978112214222 168.76444390163442 0.4919685776910522
Result: 95.96978112214222 168.76444390163442 0.4919685776910522 184.84661471813953
Integrator прерван во время паузы
Generator прерван во время паузы
Integrator завершил работу
Generator завершил работу
Оба потока завершились

Рис.10