Лабораторная работа № 6

Выполнила: Оленина Арина Игоревна
группа 6204-010302D

Оглавление

Задание 1

Я добавила в класс Functions метод, возвращающий значение интеграла функции, вычисленное с помощью численного метода.
В качестве параметров метод получает ссылку типа Function на объект функции, значения левой и правой границы области интегрирования, а также шаг дискретизации.
Если интервал интегрирования выходит за границы области определения функции, метод выбрасывает исключение.

Вычисление значения интеграла выполняется по методу трапеций. Для этого вся область интегрирования разбивается на участки, длина которых (кроме одного) равна шагу дискретизации. На каждом таком участке площадь под кривой, описывающейся заданной функцией, приближается площадью трапеции, две вершины которой расположены на оси абсцисс на границах участка, а ещё две – на кривой в точках границ участка. В область интегрирования необязательно укладывается целое количество шагов дискретизации.

```
public static double integrate(Function function, double
leftBorder, double rightBorder, double step) {
    // Проверка корректности параметров
    if (step <= 0) {
        throw new IllegalArgumentException("Шаг интегрирования
должен быть положительным");
    }
    if (leftBorder >= rightBorder) {
        throw new IllegalArgumentException("Левая граница должна
быть меньше правой");
    }

    // Проверка области определения
    if (leftBorder < function.getLeftDomainBorder() ||
rightBorder > function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Интервал
интегрирования выходит за границы области определения функции");
    }

    double integral = 0.0;
    double currentX = leftBorder;

    // Первая точка
    double prevY = function.getFunctionValue(currentX);

    // Проход по всем отрезкам
    while (currentX < rightBorder) {
        double nextX = Math.min(currentX + step, rightBorder);
        double nextY = function.getFunctionValue(nextX);

        // Площадь трапеции: (a + b) * h / 2
```

```
            integral += (prevY + nextY) * (nextX - currentX) / 2.0;

            currentX = nextX;
            prevY = nextY;
        }

        return integral;
    }
```

В методе main() проверила работу метода интегрирования. Для этого вычислила интеграл для экспоненты на отрезке от 0 до 1. Определила также, какой шаг дискретизации нужен, чтобы рассчитанное значение отличалось от теоретического в 7 знаке после запятой.

```
public static void main(String[] args) {
    // Тестирование интегрирования экспоненты на отрезке [0, 1]
    Function expFunction = new Exp();
    double theoreticalValue = Math.E - 1; // интеграл (e^x) dx
от 0 до 1 = e - 1
    double integralValue = Functions.integrate(expFunction, 0,
1, 0.0001);

    System.out.println("Теоретическое значение: " +
theoreticalValue);
    System.out.println("Значение, полученное при помощи функции:
" + integralValue);
    System.out.println("Шаг = " + 0.0001 + "\n");

}
```

Вывод:

```
Теоретическое значение: 1.718281828459045
Значение, полученное при помощи функции: 1.7182818298909435
Шаг = 1.0E-4
```

При таком шаге (0.0001) значение начинает отличаться от теоретического в 9 знаке после запятой

А при шаге 0,001 значение уже отличается в 7 точке после запятой:

```
Теоретическое значение: 1.718281828459045
Значение, полученное при помощи функции: 1.7182819716491948
Шаг = 0.001
```

Задание 2

Далее в приложении один поток инструкций генерирует задачи для интегрирования, а второй поток – решает их. Для этого я создала объект задания, через который эти потоки будут взаимодействовать. В объекте хранятся параметры задания.

Создала пакет threads, в котором разместила классы, связанные с потоками. В пакете threads описала класс Task, объект которого хранит ссылку на объект интегрируемой функции, границы области интегрирования, шаг

дискретизации, а также целочисленное поле, хранящее количество выполняемых заданий. Т.е. объект описывает одно задание.

```
package threads;
import functions.*;
public class Task {
    private Function function;       // Интегрируемая функция
    private double leftBorder;      // Левая граница
интегрирования
    private double rightBorder;     // Правая граница
интегрирования
    private double step;            // Шаг дискретизации
    private int tasksCount;         // Количество выполняемых
заданий

    // Конструктор
    public Task() {
        this.tasksCount = 0;
    }

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

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

    public double getLeftBorder() {
        return leftBorder;
    }

    public void setLeftBorder(double leftBorder) {
        this.leftBorder = leftBorder;
    }

    public double getRightBorder() {
        return rightBorder;
    }

    public void setRightBorder(double rightBorder) {
        this.rightBorder = rightBorder;
    }

    public double getStep() {
        return step;
    }

    public void setStep(double step) {
        this.step = step;
    }

    public int getTasksCount() {
```

```
        return tasksCount;
    }

    public void setTasksCount(int tasksCount) {
        this.tasksCount = tasksCount;
    }
}
```

В главном классе программы описала метод nonThread(), реализующий последовательную (без применения потоков инструкций) версию программы. В методе создала объект класса Task и установила в нём количество выполняемых заданий (100). После этого в цикле (до количества заданий) выполнила следующие действия:

1. создала и поместила в объект задания объект логарифмической функции, основание которой является случайной величиной, распределённой равномерно на отрезке от 1 до 10;
2. указала в объекте задания левую границу области интегрирования (случайно распределена на отрезке от 0 до 100);
3. указала в объекте задания правую границу области интегрирования (случайно распределена на отрезке от 100 до 200);
4. указала в объекте задания шаг дискретизации (случайно распределён на отрезке от 0 до 1);
5. вывела в консоль сообщение вида "Source <левая граница> <правая граница> <шаг дискретизации>";
6. вычислила значение интеграла для параметров из объекта задания;
7. вывела в консоль сообщение вида "Result <левая граница> <правая граница> <шаг дискретизации> <результат интегрирования>".

```
public static void nonThread() {
    System.out.println("nonThread");
    // Объект задания
    Task task = new Task();
    task.setTasksCount(100); // Минимум 100 заданий

    Random random = new Random();

    for (int i = 0; i < task.getTasksCount(); i++) {
        try {
            // Логарифмическая функция со случайным основанием
от 1 до 10
            double base = 1 + random.nextDouble() * 9;
            Function logFunction = new Log(base);
            task.setFunction(logFunction);

            // Левая граница: [0, 100]
            double leftBorder = random.nextDouble() * 100;
            task.setLeftBorder(leftBorder);
```

```
            // Правая граница: [100, 200]
            double rightBorder = 100 + random.nextDouble() *
100;
            task.setRightBorder(rightBorder);

            // Шаг дискретизации: [0, 1]
            double step = 0.01 + random.nextDouble() * 0.99;
            task.setStep(step);

            // Вывод информации о задании
            System.out.println("Source LeftBorder: " +
task.getLeftBorder()
                    + " RightBorder: " + task.getRightBorder() +
" Step: " + task.getStep() + "\n");

            // Вычисление интеграла
            double result =
Functions.integrate(task.getFunction(),
                    task.getLeftBorder(), task.getRightBorder(),
task.getStep());

            // Вывод результата
            System.out.println("Result LeftBorder: " +
task.getLeftBorder()
                    + " RightBorder: " + task.getRightBorder() +
" Step: " + task.getStep() + " Result: " + result + "\n");

        } catch (Exception e) {
            System.out.println("Ошибка в задании " + i + ": " +
e.getMessage());
        }
    }
}
```

Проверила работу метода, вызвав его в методе main().
Часть вывода:
Первые 10

Source LeftBorder: 5.1827136073217765 RightBorder: 196.35893801051444 Step: 0.3312873995354432

Result LeftBorder: 5.1827136073217765 RightBorder: 196.35893801051444 Step: 0.3312873995354432 Result: 419.6608312948884

Source LeftBorder: 79.0576395573572 RightBorder: 181.74701820149784 Step: 0.24780624611540125

Result LeftBorder: 79.0576395573572 RightBorder: 181.74701820149784 Step: 0.24780624611540125 Result: 985.7267800931035

Source LeftBorder: 98.00275795110575 RightBorder: 181.76108147155992 Step: 0.23719179458714407

Result LeftBorder: 98.00275795110575 RightBorder: 181.76108147155992 Step: 0.23719179458714407 Result: 863.0386346349324

Source LeftBorder: 87.72527476486673 RightBorder: 180.54181149656557 Step: 0.2718130302784922

Result LeftBorder: 87.72527476486673 RightBorder: 180.54181149656557 Step: 0.2718130302784922 Result: 315.4143954662491

Source LeftBorder: 75.18083257061677 RightBorder: 166.44576083488153 Step: 0.679072748997771

Result LeftBorder: 75.18083257061677 RightBorder: 166.44576083488153 Step: 0.679072748997771 Result: 484.0574853855389

Source LeftBorder: 39.93472574285866 RightBorder: 128.47334057659822 Step: 0.6499591893469989

Result LeftBorder: 39.93472574285866 RightBorder: 128.47334057659822 Step: 0.6499591893469989 Result: 1034.3614639152017

Source LeftBorder: 20.690629002164286 RightBorder: 174.0424169084945 Step: 0.5696411773522402

Result LeftBorder: 20.690629002164286 RightBorder: 174.0424169084945 Step: 0.5696411773522402 Result: 575.9611074570195

Source LeftBorder: 60.87656172719803 RightBorder: 134.34594976142418 Step: 0.5010207652963259

Result LeftBorder: 60.87656172719803 RightBorder: 134.34594976142418 Step: 0.5010207652963259 Result: 177.15270692553327

Source LeftBorder: 10.869285595586886 RightBorder: 111.1296934035812 Step: 0.49492117417525144

Result LeftBorder: 10.869285595586886 RightBorder: 111.1296934035812 Step: 0.49492117417525144 Result: 201.51045520869712

Source LeftBorder: 91.25472167615092 RightBorder: 184.448688942461 Step: 0.9842888729335909

Result LeftBorder: 91.25472167615092 RightBorder: 184.448688942461 Step: 0.9842888729335909 Result: 216.15738551044794

Последние 10

Source LeftBorder: 20.00443148882963 RightBorder: 149.49865275982035 Step: 0.2596985144702073

Result LeftBorder: 20.00443148882963 RightBorder: 149.49865275982035 Step: 0.2596985144702073 Result: 256.7194106242582

Source LeftBorder: 24.33050124806828 RightBorder: 130.40263189193524 Step: 0.44817522495765766

Result LeftBorder: 24.33050124806828 RightBorder: 130.40263189193524 Step: 0.44817522495765766 Result: 214.7493824310864

Source LeftBorder: 76.83807202968192 RightBorder: 145.98423586898312 Step: 0.10549294553586276

Result LeftBorder: 76.83807202968192 RightBorder: 145.98423586898312 Step: 0.10549294553586276 Result: 141.28498006816747

Source LeftBorder: 57.40492195960405 RightBorder: 182.0485826145537 Step: 0.5126875455029097

Result LeftBorder: 57.40492195960405 RightBorder: 182.0485826145537 Step: 0.5126875455029097 Result: 316.2254294289223

Source LeftBorder: 92.51770452557918 RightBorder: 101.72106198437781 Step: 0.4423159790480476

Result LeftBorder: 92.51770452557918 RightBorder: 101.72106198437781 Step: 0.4423159790480476 Result: 18.33226800725181

Source LeftBorder: 85.3939034393685 RightBorder: 139.19227118506427 Step: 0.10830117647640812

Result LeftBorder: 85.3939034393685 RightBorder: 139.19227118506427 Step: 0.10830117647640812 Result: 169.86401319479756

Source LeftBorder: 10.432651271638093 RightBorder: 129.87811840318332 Step: 0.6260864033861357

Result LeftBorder: 10.432651271638093 RightBorder: 129.87811840318332 Step: 0.6260864033861357 Result: 242.44825391861056

Source LeftBorder: 53.735721613972956 RightBorder: 117.64761631912543 Step: 0.357431314142735

Result LeftBorder: 53.735721613972956 RightBorder: 117.64761631912543 Step: 0.357431314142735 Result: 148.0805397873114

Source LeftBorder: 96.38147750899826 RightBorder: 198.61844365786783 Step: 0.6913200537264584

Result LeftBorder: 96.38147750899826 RightBorder: 198.61844365786783 Step: 0.6913200537264584 Result: 658.6323398714514

Source LeftBorder: 5.20088272481768 RightBorder: 157.99307355208782 Step: 0.47052831038590626

Result LeftBorder: 5.20088272481768 RightBorder: 157.99307355208782 Step: 0.47052831038590626 Result: 388.1214138421431

Задание 3

В пакете threads создала два следующих класса. При их реализации воспользовалась фрагментами последовательной версии программы из предыдущего задания.
Класс SimpleGenerator реализует интерфейс Runnable, получает в конструкторе и сохраняет в своё поле ссылку на объект типа Task, а в методе run() в цикле формируются задачи и заносятся в полученный объект задания, а также выводятся сообщения в консоль.

```
package threads;

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

public class SimpleGenerator implements Runnable {
    private Task task;

    public SimpleGenerator(Task task) {
        this.task = task;
    }

    public void run() {
        Random random = new Random();

        for (int i = 0; i < task.getTasksCount(); ++i) {
            // Логарифмическая функция со случайным основанием от 1 до 10
            double base = 1 + random.nextDouble() * 9;
```

```
            Function function = new Log(base);

            // Левая граница: [0, 100]
            double leftBorder = random.nextDouble() * 100;

            // Правая граница: [100, 200]
            double rightBorder = 100 + random.nextDouble() *
100;

            // Шаг дискретизации: [0, 1]
            double step = random.nextDouble();

            // Установка данных в задание
            task.setFunction(function);
            task.setLeftBorder(leftBorder);
            task.setRightBorder(rightBorder);
            task.setStep(step);

            // Вывод сообщения
            System.out.println("Source " + leftBorder + " " +
rightBorder + " " + step);
        }
    }
}
```

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 выводятся сообщения в консоль.

```
package threads;

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

public class SimpleIntegrator implements Runnable {
    private Task task;

    public SimpleIntegrator(Task task) {
        this.task = task;
    }

    public void run() {
        for (int i = 0; i < task.getTasksCount(); ++i) {
            // Данные из задания
            double leftBorder = task.getLeftBorder();
            double rightBorder = task.getRightBorder();
            double step = task.getStep();
            if (step <= 0) {
                System.out.println("Пропущено задание с step=" +
step);
                continue;
```

```
            }

            // Вычисление интеграла
            double result =
Functions.integrate(task.getFunction(), leftBorder, rightBorder,
step);

            // Вывод результата
            System.out.println("Result " + leftBorder + " " +
rightBorder + " " + step + " " + result);
        }
    }
}
```

В главном классе программы создала метод simpleThreads(). В нём создала объект задания, указала количество выполняемых заданий (100), создала и запустила два потока вычислений, основанных на описанных классах SimpleGenerator и SimpleIntegrator. Проверила работу метода, вызвав его в методе main().

```
public static void simpleThreads() {
    System.out.println("simpleThreads");
    // Создание объекта задания
    Task task = new Task();
    // Количество выполняемых заданий (минимум 100)
    task.setTasksCount(100);
    // Создание потоков вычислений
    Thread generatorThread = new Thread(new
SimpleGenerator(task));
    Thread integratorThread = new Thread(new
SimpleIntegrator(task));
    // Запуск потоков
    generatorThread.start();
    integratorThread.start();
}
```

Вывод:

Первые 10 вычислений

```
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
```

Последние 10 вычислений

```
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
```

```
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
Result 69.89682291654776 144.5216379170677 0.3213268548701297 190.48098313392887
```

Затем попробовала запускать программу (несколько раз). Попробовала запускать программу, изменяя приоритеты потоков перед их запуском:

```
generatorThread.setPriority(Thread.MAX_PRIORITY);
integratorThread.setPriority(Thread.MIN_PRIORITY);
```

и наоборот:

```
integratorThread.setPriority(Thread.MAX_PRIORITY);
generatorThread.setPriority(Thread.MIN_PRIORITY);
```

Убедилась, что в некоторых случаях интегрирующий поток может прекращать своё выполнение из-за исключения NullPointerException. Определила причину возникновения этого исключения и сделала так, чтобы оно не возникало (добавила проверку функции на null в SimpleIntegrator) :

```
if (function == null) {
    System.out.println("Пропущено задание - функция не готова");
    continue;
}
```

Попробовала запускать программу (несколько раз). Убедилась, что в некоторых случаях интегрирующий поток выводит сообщение с набором данных, который не встречается в сообщениях генерирующего потока (например, левая граница области интегрирования оказывается взята из одного задания, а правая граница и шаг дискретизации – из другого). Определила причину возникновения таких ситуаций (отсутствие синхронизации) и устранила её, используя блоки синхронизации в методах run() генерирующего и интегрирующего потоков. Затем убедилась, что в коде нигде не возникает необоснованно длительной блокировки объектов.

```
public class SimpleIntegrator implements Runnable {
    private Task task;

    public SimpleIntegrator(Task task) {
        this.task = task;
    }

    public void run() {
        for (int i = 0; i < task.getTasksCount(); ++i) {
            synchronized (task) {
                // Данные из задания
                double leftBorder = task.getLeftBorder();
                double rightBorder = task.getRightBorder();
                double step = task.getStep();
                Function function = task.getFunction();

                if (step <= 0) {
                    System.out.println("Пропущено задание с
step=" + step);
                    continue;
```

```
                }
                if (function == null) {
                    System.out.println("Пропущено задание -
функция не готова");
                    continue;
                }

                // Вычисление интеграла
                double result = Functions.integrate(function,
leftBorder, rightBorder, step);

                // Вывод результата
                System.out.println("Result " + leftBorder + " "
+ rightBorder + " " + step + " " + result);
            }
        }
    }
}

public class SimpleGenerator implements Runnable {
    private Task task;

    public SimpleGenerator(Task task) {
        this.task = task;
    }

    public void run() {
        Random random = new Random();

        for (int i = 0; i < task.getTasksCount(); ++i) {
            synchronized (task) {
                // Логарифмическая функция со случайным
основанием от 1 до 10
                double base = 1 + random.nextDouble() * 9;
                Function function = new Log(base);

                // Левая граница: [0, 100]
                double leftBorder = random.nextDouble() * 100;

                // Правая граница: [100, 200]
                double rightBorder = 100 + random.nextDouble() *
100;

                // Шаг дискретизации: [0.01, 1]
                double step = 0.01 + random.nextDouble() * 0.99;

                // Установка данных в задание
                task.setFunction(function);
                task.setLeftBorder(leftBorder);
                task.setRightBorder(rightBorder);
                task.setStep(step);

                // Вывод сообщения
```

```
                System.out.println("Source " + leftBorder + " "
+ rightBorder + " " + step);
            }
        }
    }
}
```

Вывод:
Первый 10 вычислений:

```
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
Пропущено задание с step=0.0
```

Последние 10 вычислений:

```
Source 42.011391170892765 183.20535750340485 0.8339652178046769
Source 31.583505632660692 178.68365064786565 0.837324854285267
Source 3.5801921201075393 165.2445300985409 0.978106319850356
Source 64.73071182423654 169.84159858905025 0.6934417109227501
Source 52.20294971250713 149.89192660142535 0.3618174316902725
Source 73.3748830938698 102.9788473439728 0.577646544692936
Source 45.70914172630419 151.29545891669443 0.2769928926469035
Source 17.544241907620794 169.06918311178828 0.2538911011288483
Source 73.37004627481691 160.3678793712051 0.4086893016424843
Source 53.033799563815755 161.973198238361 0.9977611120267897
```

Задание 4

Затем я определила причину того, что не все сгенерированные задания оказываются выполнены интегрирующим потоком (генератор работает быстрее и часто захватывает монитор для чтения, а интегратор в это время не может работать и постоянно ждет доступа, в результате интегратор обрабатывает меньше заданий).

Для устранения этой проблемы потребуется дополнительный объект, представляющий собой одноместный семафор, различающий операции чтения и записи в защищаемый объект.

В пакете threads создала два следующих класса. При их реализации воспользовалась фрагментами классов из предыдущего задания.

Класс Generator расширяет класс Thread, получает в конструкторе и сохраняет в свои поля ссылки на объект типа Task и на объект семафора, а в методе run() выполняются те же действия, что и в предыдущей версии генерирующего класса.

```
package threads;

import functions.*;
import functions.basic.*;
import java.util.concurrent.Semaphore;

public class Generator extends Thread {
    private Task task;
    private Semaphore writeSemaphore;
    private Semaphore readSemaphore;

    public Generator(Task task, Semaphore writeSemaphore,
Semaphore readSemaphore) {
        this.task = task;
        this.writeSemaphore = writeSemaphore;
        this.readSemaphore = readSemaphore;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < task.getTasksCount(); ++i) {
                // Проверка прерывания
                if (Thread.interrupted()) {
                    System.out.println("Generator был прерван");
                    return;
                }

                // Генерация данных
                double base = 1 + Math.random() * 9;
                Function function = new Log(base);
                double leftBorder = Math.random() * 100;
                double rightBorder = 100 + Math.random() * 100;
                double step = 0.01 + Math.random() * 0.99;

                // Захват семафора записи
                writeSemaphore.acquire();
                try {
                    // Запись данных в задание
                    task.setFunction(function);
                    task.setLeftBorder(leftBorder);
                    task.setRightBorder(rightBorder);
                    task.setStep(step);

                    System.out.println("Source " + leftBorder +
" " + rightBorder + " " + step);
                } finally {
                    // Разрешение чтения
```

```
                    readSemaphore.release();
                }
            }
        } catch (InterruptedException e) {
            System.out.println("Generator был прерван при
ожидании семафора");
        }
    }
}
```

Класс Integrator расширяет класс Thread, получает в конструкторе и сохраняет в свои поля ссылки на объект типа Task и на объект семафора, а в методе run() выполняются те же действия, что и в предыдущей версии интегрирующего класса.

```
package threads;

import functions.Function;
import functions.Functions;
import functions.InappropriateFunctionPointException;
import java.util.concurrent.Semaphore;

public class Integrator extends Thread {
    private Task task;
    private Semaphore writeSemaphore;
    private Semaphore readSemaphore;

    public Integrator(Task task, Semaphore writeSemaphore,
Semaphore readSemaphore) {
        this.task = task;
        this.writeSemaphore = writeSemaphore;
        this.readSemaphore = readSemaphore;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < task.getTasksCount(); ++i) {
                // Проверка прерывания
                if (Thread.interrupted()) {
                    System.out.println("Integrator был
прерван");
                    return;
                }

                // Ожидание данных для чтения
                readSemaphore.acquire();
                try {
                    // Чтение данных из задания
                    double leftBorder = task.getLeftBorder();
                    double rightBorder = task.getRightBorder();
                    double step = task.getStep();
                    Function function = task.getFunction();
```

```
                    // Проверка корректности данных
                    if (function == null || step <= 0) {
                        System.out.println("Пропущено задание -
некорректные данные");
                        continue;
                    }

                    // Вычисление интеграла
                    double result =
Functions.integrate(function, leftBorder, rightBorder, step);
                    System.out.println("Result " + leftBorder +
" " + rightBorder + " " + step + " " + result);
                } finally {
                    // Разрешение записи
                    writeSemaphore.release();
                }
            }
        } catch (InterruptedException e) {
            System.out.println("Integrator был прерван при
ожидании семафора");
        }
    }
}
```

Отличие этих классов от предыдущих версий заключается в том, что вместо средств синхронизации в методах run() используются возможности семафора. В главном классе программы создала метод complicatedThreads(). В нём создала объект задания, указала количество выполняемых заданий (минимум 100), создала и запустила два потока вычислений
классов Generator и Integrator. Проверила работу метода, вызвав его в методе main().

```
public static void complicatedThreads() {
    System.out.println("complicatedThreads");
    // Объект задания
    Task task = new Task();
    task.setTasksCount(100);

    // Два семафора:
    // writeSemaphore начинается с 1 разрешения (можно писать)
    // readSemaphore начинается с 0 разрешений (нельзя читать)
    Semaphore writeSemaphore = new Semaphore(1);
    Semaphore readSemaphore = new Semaphore(0);

    // Создание потоков
    Generator generator = new Generator(task, writeSemaphore,
readSemaphore);
    Integrator integrator = new Integrator(task, writeSemaphore,
readSemaphore);

    // Запуск потоков
    generator.start();
    integrator.start();
```

```
    try {
        // Ожидание завершения потоков
        generator.join();
        integrator.join();

    } catch (InterruptedException e) {
        System.out.println("Основной поток был прерван");
    }
}
```

Вывод:

Первые 10 вычислений

```
Source 82.16297211091796 158.7135304223159 0.8596557671289822
Result 82.16297211091796 158.7135304223159 0.8596557671289822 577.884942785868
Source 71.01763365877565 188.4797192765576 0.5823650128854939
Result 71.01763365877565 188.4797192765576 0.5823650128854939 253.3794802134409
Source 69.27855968388207 154.86435266251033 0.9776346362325224
Result 69.27855968388207 154.86435266251033 0.9776346362325224 259.99717237961715
Source 91.36975600176976 132.53178332488025 0.2579873420899525
Result 91.36975600176976 132.53178332488025 0.2579873420899525 100.37808734096058
Source 26.69238437031307 190.52828586843316 0.16536316775930637
Result 26.69238437031307 190.52828586843316 0.16536316775930637 428.7970333398903
Source 90.77936601222262 138.08494551610195 0.36271914792897514
Result 90.77936601222262 138.08494551610195 0.36271914792897514 100.20020891440768
Source 11.883306576042008 100.14479129484707 0.894493789255077
Result 11.883306576042008 100.14479129484707 0.894493789255077 178.9796674357163
Source 44.846443270646134 199.54589354471648 0.07509768486713198
Result 44.846443270646134 199.54589354471648 0.07509768486713198
329.42801338679516
Source 54.98030603228642 115.07736759699453 0.3100113445299902
Result 54.98030603228642 115.07736759699453 0.3100113445299902 147.20784691350065
Source 68.68666455392417 105.03534259741417 0.6254282208932361
Result 68.68666455392417 105.03534259741417 0.6254282208932361 175.46329071565455
```

Последние 10 вычислений

```
Source 58.13403865841398 150.58981324784128 0.1742742556033076
Result 58.13403865841398 150.58981324784128 0.1742742556033076 338.6024617385616
Source 2.7189849106925235 159.124242363349 0.921521698300266
Result 2.7189849106925235 159.124242363349 0.921521698300266 295.9609568341046
Source 35.54435697270153 164.19665575908127 0.6283573306424445
Result 35.54435697270153 164.19665575908127 0.6283573306424445 296.4539605058025
Source 21.47375649411053 180.35707812729635 0.7053538373937547
Result 21.47375649411053 180.35707812729635 0.7053538373937547 657.3662647257876
Source 41.28401400225702 198.66580766388222 0.742438839281298
Result 41.28401400225702 198.66580766388222 0.742438839281298 513.1556164375671
Source 8.73565953415929 198.2916541636118 0.27148413255824233
Result 8.73565953415929 198.2916541636118 0.27148413255824233 369.89458063542173
Source 84.48419085339138 138.7438035884533 0.4165435060209086
Result 84.48419085339138 138.7438035884533 0.4165435060209086 119.62027981410981
Source 73.40738945121397 155.958169668539 0.3183806215859
Result 73.40738945121397 155.958169668539 0.3183806215859 450.98486380977806
Source 24.052969706958237 108.99416016689558 0.2597234932564231
Result 24.052969706958237 108.99416016689558 0.2597234932564231 160.06089603151318
Source 84.1646320241152 175.758065084913 0.39341446962145854
```

Result 84.1646320241152 175.758065084913 0.39341446962145854 212.3064507990133

Также попробовала запускать программу (несколько раз). И попробовала запускать программу, изменяя приоритеты потоков перед их запуском. Затем сделала так, чтобы после создания потоков основной поток программы выжидал 50 миллисекунд, после чего прерывал работу потов путём вызова метода interrupt(). Изменила код классов потоков так, чтобы прерывание происходило корректно.

```
try {
    // 50 миллисекунд
    Thread.sleep(50);

    // Прерывание потоки
    generator.interrupt();
    integrator.interrupt();

    // Ожидание завершения потоков
    generator.join();
    integrator.join();

} catch (InterruptedException e) {
    System.out.println("Основной поток был прерван");
}
```

Первые 10 вычислений

Source 82.79799000582102 195.3498378706468 0.3403303192157758

Result 82.79799000582102 195.3498378706468 0.3403303192157758 320.85614085051185

Source 51.45319181712382 159.15044439490538 0.18719448161422622

Result 51.45319181712382 159.15044439490538 0.18719448161422622 4345.426250271849

Source 20.911858611655155 100.75459820131385 0.042634602848460046

Result 20.911858611655155 100.75459820131385 0.042634602848460046 620.0683990969567

Source 86.43985800396143 135.84917994485622 0.6775689822473117

Result 86.43985800396143 135.84917994485622 0.6775689822473117 165.6180663283671

Source 99.12960126231795 106.872645093155 0.2929965780029777

Result 99.12960126231795 106.872645093155 0.2929965780029777 17.757251159129442

Source 31.523128314041482 187.3148908439004 0.7359312345344461

Result 31.523128314041482 187.3148908439004 0.7359312345344461 314.3268162817565

Source 97.88542697792964 128.83030317884288 0.817549012732446

Result 97.88542697792964 128.83030317884288 0.817549012732446 78.24842320308541

Source 41.386856442977006 121.16221144183669 0.9402386816176229

Result 41.386856442977006 121.16221144183669 0.9402386816176229 396.86655679563285

Source 79.45751178247812 180.28701933511286 0.8491038282985832

Result 79.45751178247812 180.28701933511286 0.8491038282985832 217.47663731680024

Source 80.7052651226062 153.72152487822098 0.7482586682573283

Result 80.7052651226062 153.72152487822098 0.7482586682573283 202.16909398683035

Последние 10 вычислений

Source 25.150446484132427 139.9037297633476 0.6894441863045505

Result 25.150446484132427 139.9037297633476 0.6894441863045505 237.49466908463194

Source 74.02461614517676 113.94262757118165 0.5205579656710918

Result 74.02461614517676 113.94262757118165 0.5205579656710918 137.69772950194468

Source 11.858182365285275 183.4547407584549 0.6639012161427347

Result 11.858182365285275 183.4547407584549 0.6639012161427347 1204.0851255307598

Source 0.34670493466115726 105.11933985414868 0.503869990728514

Result 0.34670493466115726 105.11933985414868 0.503869990728514 329.4027605026678

Source 67.77810637647517 140.7795440645744 0.3814882493326059

Result 67.77810637647517 140.7795440645744 0.3814882493326059 280.69413061080246

Source 37.77012441941399 161.50357849946843 0.5249932230014428

Result 37.77012441941399 161.50357849946843 0.5249932230014428 323.46202937036105

Source 51.604502352179985 186.97020874915478 0.36624160655059984

Result 51.604502352179985 186.97020874915478 0.36624160655059984 324.30555418482527

Source 22.393412832540783 157.715608453784 0.5864824291242374

Result 22.393412832540783 157.715608453784 0.5864824291242374 293.971592205397

Source 19.56746359273689 114.43903310140641 0.32827491475533305

Result 19.56746359273689 114.43903310140641 0.32827491475533305 516.8470800400288

Source 45.55360318668148 163.3498544501488 0.3494996208795132

Result 45.55360318668148 163.3498544501488 0.3494996208795132 255.66802231576972