Министерство науки и высшего образования Российской Федерации

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

Институт информатики и кибернетики Кафедра

технической кибернетики

Отчет по лабораторной работе №6

Дисциплина: «ООП»

Выполнил: Дорофеева Александра Алексеевна

Группа: 6201-120303D

Проверил: преподаватель Борисов Д.С.

Самара, 2025

Задание 1

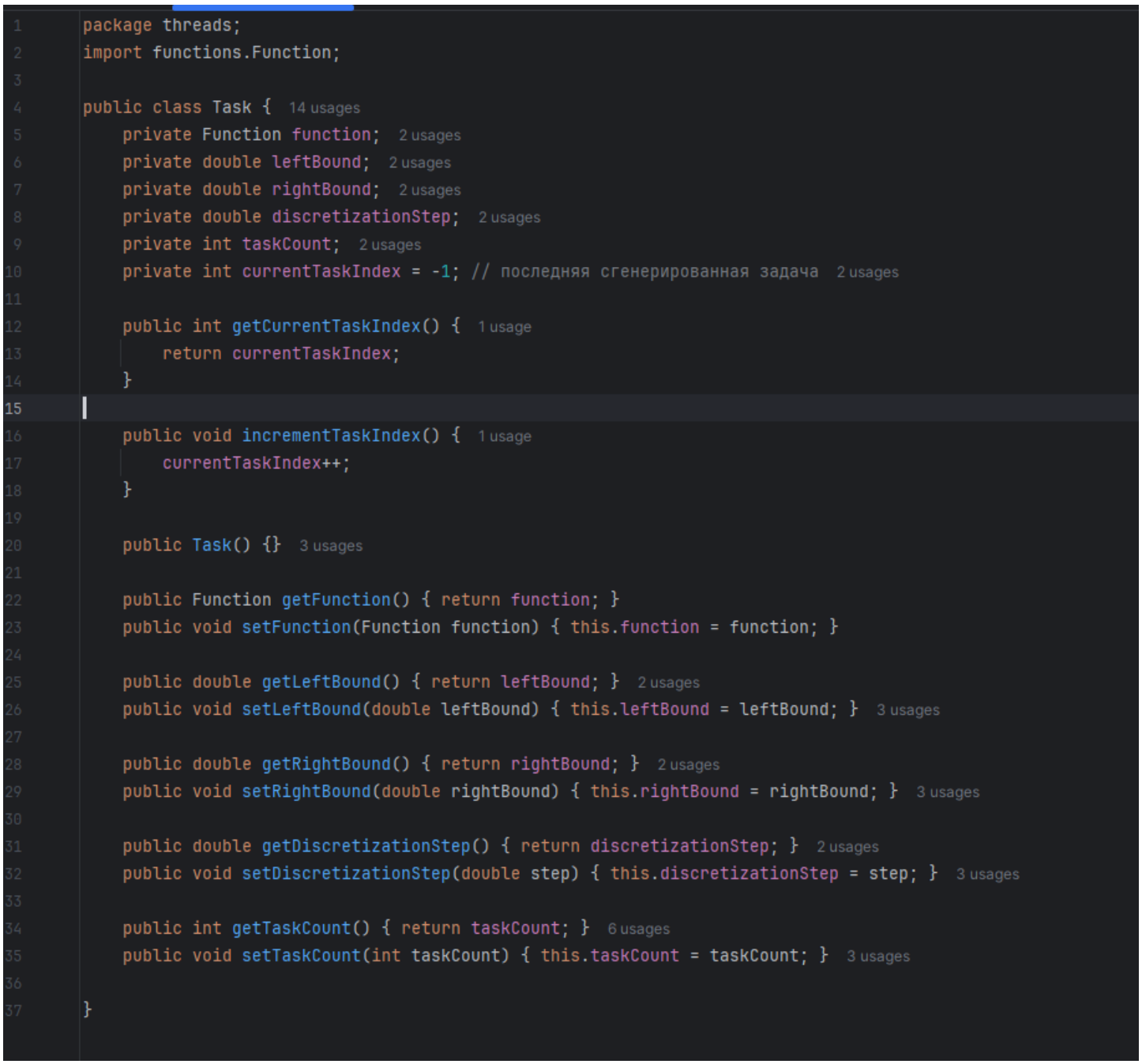
```java
public static double integrate(Function f, double leftX, double rightX, double step) {  4 usages
    if (leftX < f.getLeftDomainBorder() || rightX > f.getRightDomainBorder()) {
        throw new IllegalArgumentException("Интервал интегрирования выходит за границы функции");
    }
    if (step <= 0) {
        throw new IllegalArgumentException("Шаг интегрирования должен быть положительным");
    }

    double sum = 0.0;
    double x = leftX;

    while (x < rightX) {
        double nextX = Math.min(x + step, rightX); // учёт последнего участка, если меньше шага
        double y1 = f.getFunctionValue(x);
        double y2 = f.getFunctionValue(nextX);
        sum += (y1 + y2) / 2 * (nextX - x); // площадь трапеции
        x = nextX;
    }

    return sum;
}
```

Считает интеграл методом трапеций: проверяет границы и шаг, делит интервал на части, для каждой части находит площадь трапеции и суммирует их.

```
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:G:\intellij IDE
=== Задание 1: Интеграл exp(x) от 0 до 1 ===
Интеграл exp(x) от 0,00 до 1,00 с шагом 0,0003906250 = 1,7182819159
```

Результат

Задание 2



```java
package threads;
import functions.Function;

public class Task {  14 usages
    private Function function;  2 usages
    private double leftBound;  2 usages
    private double rightBound;  2 usages
    private double discretizationStep;  2 usages
    private int taskCount;  2 usages
    private int currentTaskIndex = -1; // последняя сгенерированная задача  2 usages

    public int getCurrentTaskIndex() {  1 usage
        return currentTaskIndex;
    }

    public void incrementTaskIndex() {  1 usage
        currentTaskIndex++;
    }

    public Task() {}  3 usages

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

    public double getLeftBound() { return leftBound; }  2 usages
    public void setLeftBound(double leftBound) { this.leftBound = leftBound; }  3 usages

    public double getRightBound() { return rightBound; }  2 usages
    public void setRightBound(double rightBound) { this.rightBound = rightBound; }  3 usages

    public double getDiscretizationStep() { return discretizationStep; }  2 usages
    public void setDiscretizationStep(double step) { this.discretizationStep = step; }  3 usages

    public int getTaskCount() { return taskCount; }  6 usages
    public void setTaskCount(int taskCount) { this.taskCount = taskCount; }  3 usages

}
```

Этот код представляет собой класс Task, который служит контейнером для передачи данных между потоками. Он хранит: функцию для интегрирования (function), границы интегрирования (leftBound, rightBound), шаг дискретизации (discretizationStep), общее количество задач (taskCount) и индекс текущей готовой задачи (currentTaskIndex). Класс предоставляет стандартные геттеры и сеттеры для доступа к полям, а также специальный метод incrementTaskIndex() для атомарного увеличения счётчика готовых задач, что используется для синхронизации между потоком-генератором (который создаёт задачи) и потоком-интегратором (который их обрабатывает).

```java
// Метод последовательного выполнения (Задание 2)
public static void nonThread() {  1 usage
    Random rand = new Random();
    Task task = new Task();
    task.setTaskCount(100);

    System.out.println("Количество заданий: " + task.getTaskCount());

    for (int i = 0; i < task.getTaskCount(); i++) {
        int taskNumber = i + 1;

        double base = 1 + 9 * rand.nextDouble();
        double left = 100 * rand.nextDouble();
        double right = left + 100 * rand.nextDouble();
        double step = 0.01 + 0.99 * rand.nextDouble();

        task.setFunction(new Log(base));
        task.setLeftBound(left);
        task.setRightBound(right);
        task.setDiscretizationStep(step);

        System.out.printf("Task %d - Source: %.6f %.6f %.6f%n", taskNumber, left, right, step);

        try {
            double result = Functions.integrate(task.getFunction(), left, right, step);
            System.out.printf("Task %d - Result: %.6f %.6f %.6f %.6f%n", taskNumber, left, right, step, result);
        } catch (IllegalArgumentException e) {
            System.out.printf("Task %d - Ошибка интегрирования: %s%n", taskNumber, e.getMessage());
        }
    }
}
```

Этот код реализует последовательное (без многопоточности) выполнение 100 задач интегрирования: создаётся объект Task, затем в цикле генерируются случайные параметры для логарифмической функции (основание от 1 до 10, левая граница от 0 до 100, правая граница на 100 единиц дальше левой, шаг от 0.01 до 1), эти параметры устанавливаются в объект задачи, выводится информация о задаче, вычисляется интеграл с помощью Functions.integrate() и выводится результат, при этом перехватываются возможные исключения (например, если интервал интегрирования выходит за область определения функции) с выводом сообщения об ошибке.

```
 Задание 2: 100 случайных логарифмических интегралов
Количество заданий: 100
Task 1 - Source: 94,072932 106,962773 0,922706
Task 1 - Result: 94,072932 106,962773 0,922706 28,437391
Task 2 - Source: 34,358598 35,469236 0,635519
Task 2 - Result: 34,358598 35,469236 0,635519 1,718861
Task 3 - Source: 1,945724 88,771172 0,528126
Task 3 - Result: 1,945724 88,771172 0,528126 191,249232
Task 4 - Source: 65,191399 97,305868 0,478184
Task 4 - Result: 65,191399 97,305868 0,478184 70,088210
Task 5 - Source: 57,815008 137,060964 0,504501
Task 5 - Result: 57,815008 137,060964 0,504501 172,090039
Task 6 - Source: 44,484294 100,721464 0,654971
Task 6 - Result: 44,484294 100,721464 0,654971 163,915078
Task 7 - Source: 91,147728 122,563700 0,763016
Task 7 - Result: 91,147728 122,563700 0,763016 100,051481
Task 8 - Source: 68,338957 106,833859 0,360193
Task 8 - Result: 68,338957 106,833859 0,360193 101,045038
Task 9 - Source: 18,120406 18,137658 0,185999
Task 9 - Result: 18,120406 18,137658 0,185999 0,057717
Task 10 - Source: 70,899551 132,406789 0,990248
Task 10 - Result: 70,899551 132,406789 0,990248 127,766488
Task 11 - Source: 7,042832 18,297004 0,498651
Task 11 - Result: 7,042832 18,297004 0,498651 13,457678
Task 12 - Source: 20,028480 21,411482 0,558421
```

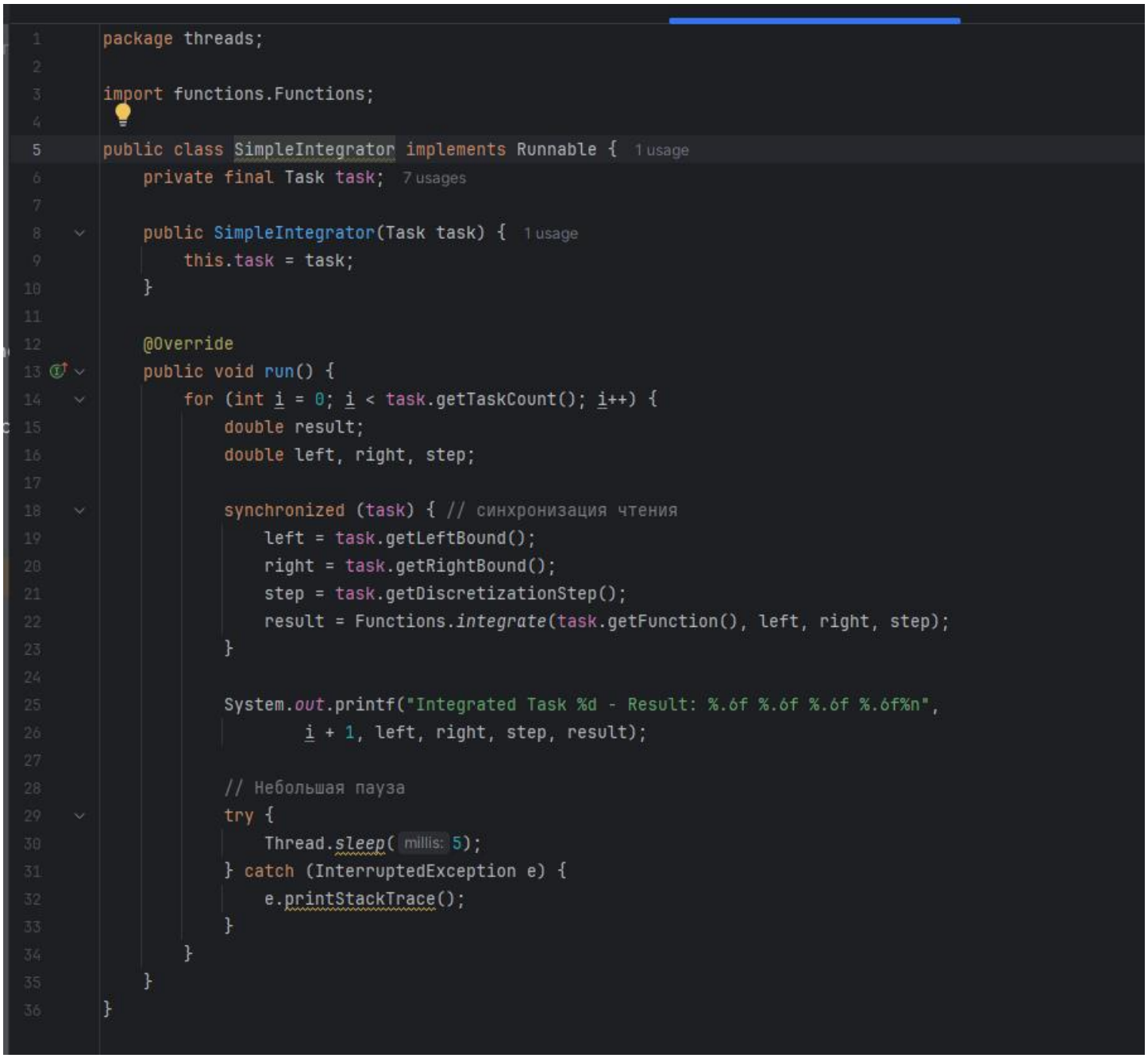
Кусочек результатов

Задание 3

```java
package threads;

import ...

public class SimpleGenerator implements Runnable {  1 usage
    private final Task task;  7 usages
    private final Random rand = new Random();  4 usages

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

    @Override
    public void run() {
        for (int i = 0; i < task.getTaskCount(); i++) {
            synchronized (task) { // синхронизация записи
                double base = 1 + 9 * rand.nextDouble();        // [1,10]
                double left = 100 * rand.nextDouble();         // [0,100]
                double right = left + 100 * rand.nextDouble(); // [left, left+100]
                double step = 0.01 + 0.99 * rand.nextDouble(); // [0.01,1]

                task.setFunction(new Log(base));
                task.setLeftBound(left);
                task.setRightBound(right);
                task.setDiscretizationStep(step);

                System.out.printf("Generated Task %d - Source: %.6f %.6f %.6f%n",
                        i + 1, left, right, step);
            }

            // Небольшая пауза, чтобы дать интегратору шанс прочитать данные
            try {
                Thread.sleep( millis: 5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Этот код представляет класс `SimpleGenerator`, который реализует интерфейс `Runnable` и работает в отдельном потоке. Он получает в конструкторе общий объект `Task` и в методе `run()` в цикле генерирует 100 задач: создаёт случайные параметры для логарифмической функции (основание от 1 до 10, границы интегрирования от 0 до 200 со смещением, шаг от 0.01 до 1) и атомарно записывает их в объект `Task` внутри блока `synchronized`, чтобы избежать race condition. После записи данных каждого задания выводится сообщение, а

затем поток делает короткую паузу `Thread.sleep(5)`, чтобы дать возможность другому потоку (интегратору) прочитать данные.



```java
package threads;

import functions.Functions;

public class SimpleIntegrator implements Runnable {  1 usage
    private final Task task;  7 usages

    public SimpleIntegrator(Task task) {  1 usage
        this.task = task;
    }

    @Override
    public void run() {
        for (int i = 0; i < task.getTaskCount(); i++) {
            double result;
            double left, right, step;

            synchronized (task) { // синхронизация чтения
                left = task.getLeftBound();
                right = task.getRightBound();
                step = task.getDiscretizationStep();
                result = Functions.integrate(task.getFunction(), left, right, step);
            }

            System.out.printf("Integrated Task %d - Result: %.6f %.6f %.6f %.6f%n",
                    i + 1, left, right, step, result);

            // Небольшая пауза
            try {
                Thread.sleep( millis: 5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Этот код реализует класс `SimpleIntegrator`, который работает в отдельном потоке и обрабатывает задачи интегрирования. Он получает общий объект `Task` в конструкторе и в методе `run()` в цикле читает параметры очередной задачи (границы интегрирования и шаг) атомарно внутри блока `synchronized`, вычисляет интеграл с помощью `Functions.integrate()` и выводит результат. После обработки каждой задачи поток делает короткую паузу `Thread.sleep()` для передачи управления другому потоку (генератору), обеспечивая поочерёдную работу потоков.

```java
// Метод потокового выполнения (Задание 3)
public static void simpleThreads() {  1 usage
    Task task = new Task();
    task.setTaskCount(100);

    Thread generatorThread = new Thread(new SimpleGenerator(task));
    Thread integratorThread = new Thread(new SimpleIntegrator(task));

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

    generatorThread.start();
    integratorThread.start();

    try {
        generatorThread.join();
        integratorThread.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Все задачи сгенерированы и интегрированы потоками.");
}
```

Этот код запускает два параллельных потока для многопоточной обработки: создаёт объект `Task` на 100 задач, оборачивает `SimpleGenerator` и `SimpleIntegrator` в потоки, назначает генератору максимальный приоритет, а интегратору — минимальный для тестирования приоритетов, запускает оба потока методом `start()`, затем ожидает их завершения с помощью `join()` и выводит итоговое сообщение после обработки всех задач.

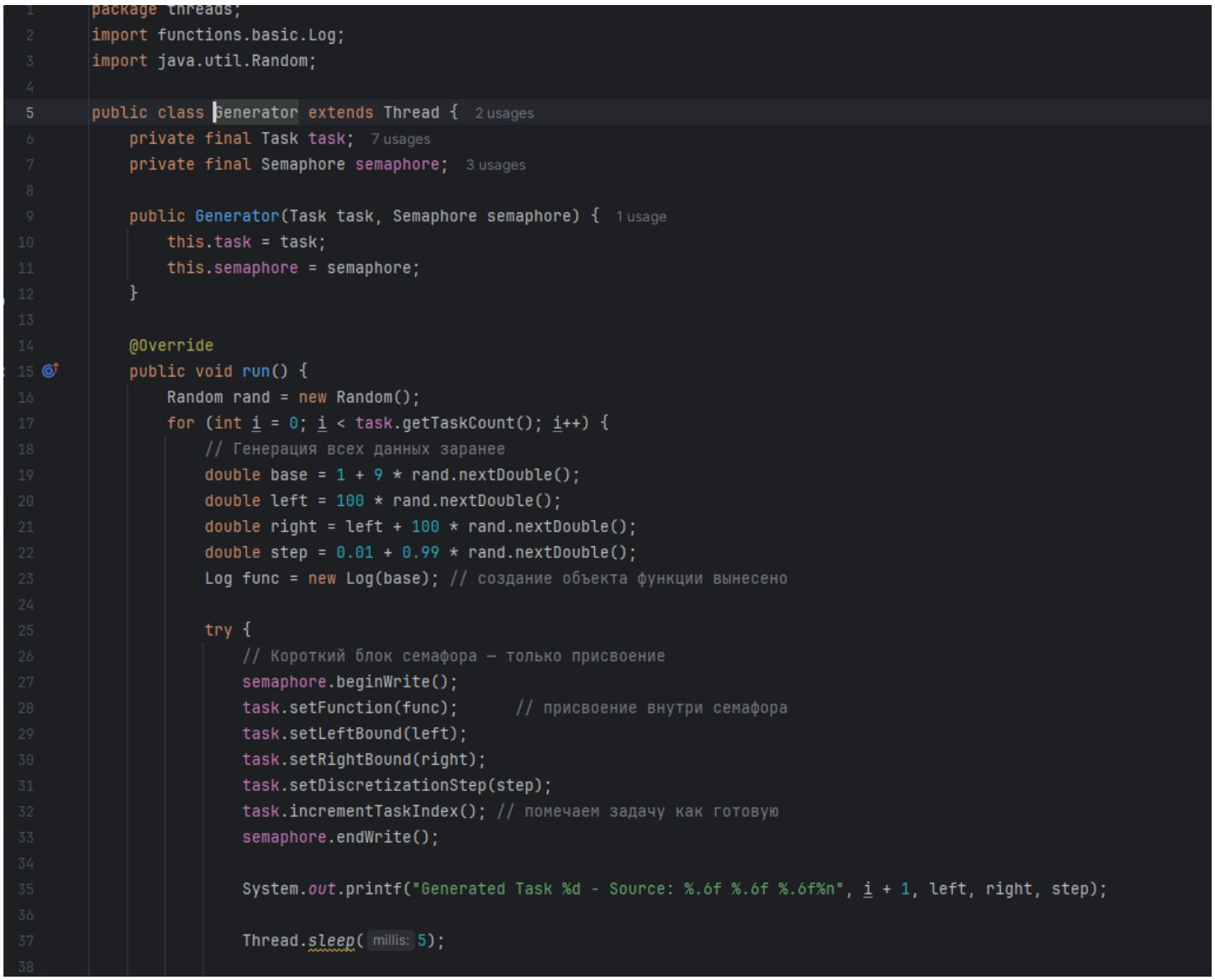
```
 Задание 3: Потоковая генерация и интегрирование
Generated Task 1 - Source: 7,861532 41,501610 0,036806
Integrated Task 1 - Result: 7,861532 41,501610 0,036806 104,091184
Integrated Task 2 - Result: 7,861532 41,501610 0,036806 104,091184
Generated Task 2 - Source: 91,410281 130,131987 0,887566
Integrated Task 3 - Result: 91,410281 130,131987 0,887566 84,095566
Generated Task 3 - Source: 55,899472 99,368042 0,562582
Integrated Task 4 - Result: 55,899472 99,368042 0,562582 100,411190
Generated Task 4 - Source: 80,267711 154,535530 0,977555
Integrated Task 5 - Result: 80,267711 154,535530 0,977555 355,451248
Generated Task 5 - Source: 65,420751 95,630700 0,416136
Integrated Task 6 - Result: 65,420751 95,630700 0,416136 74,535826
Generated Task 6 - Source: 99,276931 147,253570 0,408782
Integrated Task 7 - Result: 99,276931 147,253570 0,408782 339,114578
Generated Task 7 - Source: 64,448087 154,626478 0,669482
Integrated Task 8 - Result: 64,448087 154,626478 0,669482 335,636731
Generated Task 8 - Source: 67,123392 106,665867 0,874264
Integrated Task 9 - Result: 67,123392 106,665867 0,874264 89,228893
Generated Task 9 - Source: 91,602540 140,239419 0,493349
Integrated Task 10 - Result: 91,602540 140,239419 0,493349 4942,518668
Generated Task 10 - Source: 13,928641 104,154693 0,442626
Integrated Task 11 - Result: 13,928641 104,154693 0,442626 298,044723
```

Кусочек результатов

Задание 4

```java
package threads;

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

    // Генератор начинает запись
    public synchronized void beginWrite() throws InterruptedException {  1 usage
        writeRequests++;
        while (readers > 0 || writers > 0) {
            wait();
        }
        writeRequests--;
        writers++;
    }

    // Генератор заканчивает запись
    public synchronized void endWrite() {  1 usage
        writers--;
        notifyAll(); // уведомляем интегратор
    }

    // Интегратор начинает чтение
    public synchronized void beginRead() throws InterruptedException {  1 usage
        while (writers > 0 || writeRequests > 0) {
            wait();
        }
        readers++;
    }

    // Интегратор заканчивает чтение
    public synchronized void endRead() {  1 usage
        readers--;
        notifyAll(); // уведомляем генератор
    }
}
```

Этот код реализует семафор с разделением на читателей и писателей: класс `Semaphore` отслеживает количество активных читателей (`readers`), активных писателей (`writers`) и запросов на запись (`writeRequests`). Методы `beginWrite()`/`endWrite()` предоставляют эксклюзивный доступ писателю, заставляя его ждать, пока есть другие читатели или писатели. Методы `beginRead()`/`endRead()` позволяют нескольким читателям работать одновременно, но блокируют их, если есть активные писатели или запросы на запись. Использование `synchronized`, `wait()` и `notifyAll()` обеспечивает корректную синхронизацию между потоками.



```java
package threads;
import functions.basic.Log;
import java.util.Random;

public class Generator extends Thread {  2 usages
    private final Task task;  7 usages
    private final Semaphore semaphore;  3 usages

    public Generator(Task task, Semaphore semaphore) {  1 usage
        this.task = task;
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        Random rand = new Random();
        for (int i = 0; i < task.getTaskCount(); i++) {
            // Генерация всех данных заранее
            double base = 1 + 9 * rand.nextDouble();
            double left = 100 * rand.nextDouble();
            double right = left + 100 * rand.nextDouble();
            double step = 0.01 + 0.99 * rand.nextDouble();
            Log func = new Log(base); // создание объекта функции вынесено

            try {
                // Короткий блок семафора — только присвоение
                semaphore.beginWrite();
                task.setFunction(func);      // присвоение внутри семафора
                task.setLeftBound(left);
                task.setRightBound(right);
                task.setDiscretizationStep(step);
                task.incrementTaskIndex(); // помечаем задачу как готовую
                semaphore.endWrite();

                System.out.printf("Generated Task %d - Source: %.6f %.6f %.6f%n", i + 1, left, right, step);

                Thread.sleep( millis: 5);
```

```java
            } catch (InterruptedException e) {
                System.out.println("Generator прерван.");
                break;
            }
        }
    }
}
```

Этот код реализует поток-генератор, который создаёт задачи для интегрирования: класс `Generator` расширяет `Thread`, получает в конструкторе общий объект `Task` и семафор. В методе `run()` он в цикле генерирует случайные параметры логарифмической функции (как в предыдущих заданиях), затем использует семафор для атомарной записи данных в `Task`: вызывает `beginWrite()` для получения эксклюзивного доступа, устанавливает параметры задачи, инкрементирует индекс готовой задачи через `incrementTaskIndex()`, завершает

запись через `endWrite()`, выводит информацию и делает паузу. Код корректно обрабатывает прерывание потока, выводя сообщение и завершая цикл.

```java
package threads;
import functions.Functions;

public class Integrator extends Thread {  2 usages
    private final Task task;  7 usages
    private final Semaphore semaphore;  3 usages

    public Integrator(Task task, Semaphore semaphore) {  1 usage
        this.task = task;
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        for (int i = 0; i < task.getTaskCount(); i++) {
            boolean done = false;

            while (!done) {
                try {
                    semaphore.beginRead();

                    // Проверяем, подготовил ли генератор i-ю задачу
                    if (task.getCurrentTaskIndex() >= i) {
                        double left = task.getLeftBound();
                        double right = task.getRightBound();
                        double step = task.getDiscretizationStep();

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

                        System.out.printf("Integrator Task %d - Result: %.6f %.6f %.6f %.6f%n",
                                i + 1, left, right, step, result);

                        done = true; // интегрирование выполнено
                    }

                    semaphore.endRead();
```

```java
                    if (!done) {
                        Thread.sleep( millis: 1); // ждём генератора
                    }

                } catch (InterruptedException e) {
                    System.out.println("Integrator прерван.");
                    done = true;
                    break;
                }
            }
        }

    }
}
```

Этот код реализует поток-интегратор, который обрабатывает подготовленные задачи: класс `Integrator` расширяет `Thread`, получает общий `Task` и семафор. В методе `run()` для каждой из 100 задач он входит в цикл, где с помощью `beginRead()` получает доступ на чтение, проверяет, готова ли текущая задача (по индексу `currentTaskIndex`), и если да — читает параметры, вычисляет интеграл, выводит результат и помечает задачу как выполненную. Если задача ещё не готова, поток освобождает семафор через `endRead()` и ждёт 1 мс, повторяя проверку. Код корректно обрабатывает прерывание, завершая выполнение с выводом сообщения.

```
    // Метод для Задания 4
    public static void complicatedThreads() {  1 usage
        Task task = new Task();
        task.setTaskCount(100);

        Semaphore semaphore = new Semaphore();

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

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

        try {
            // Ждём завершения потоков (без прерывания)
            generator.join();
            integrator.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Все задачи успешно обработаны с использованием семафора.");
    }

}
```

Этот код запускает многопоточную обработку с использованием семафора: создаёт объект `Task` на 100 задач и семафор `Semaphore`, затем создаёт и запускает два потока — `Generator` (генератор задач) и `Integrator` (интегратор), которые используют общий семафор для синхронизации доступа к объекту задачи. После запуска потоков главный поток ожидает их завершения с помощью `join()` и выводит сообщение об успешной обработке всех задач с использованием семафора.

```
 Задание 4: Потоки с семафором и прерыванием
Generated Task 1 - Source: 79,901553 171,187160 0,176873
Integrator Task 1 - Result: 79,901553 171,187160 0,176873 219,804991
Generated Task 2 - Source: 42,318476 93,336577 0,756878
Integrator Task 2 - Result: 42,318476 93,336577 0,756878 138,455844
Generated Task 3 - Source: 14,612083 61,387019 0,078965
Integrator Task 3 - Result: 14,612083 61,387019 0,078965 75,511262
Generated Task 4 - Source: 48,940737 94,199943 0,129447
Integrator Task 4 - Result: 48,940737 94,199943 0,129447 125,197987
Generated Task 5 - Source: 79,379138 164,241166 0,287495
Integrator Task 5 - Result: 79,379138 164,241166 0,287495 186,352568
Generated Task 6 - Source: 83,404716 139,047216 0,447327
Integrator Task 6 - Result: 83,404716 139,047216 0,447327 152,933974
Generated Task 7 - Source: 69,578111 156,150605 0,781377
Integrator Task 7 - Result: 69,578111 156,150605 0,781377 209,249916
Generated Task 8 - Source: 15,346593 25,732779 0,678935
Integrator Task 8 - Result: 15,346593 25,732779 0,678935 18,053925
```

Кусочек результатов