

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

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

ОТЧЕТ ПО
ЛАБОРАТОРНОЙ РАБОТЕ № 5

**«Переопределение и использование
стандартных функций класса Object»**

по курсу
Объектно-ориентированное программирование

Выполнил: Петрухин Роман,
студент группы 6203-010302D

Оглавление

Пункт №1	3
Пункт №2	3-4
Пункт №3	5-8
Пункт №4	8
Пункт №5	9-11
Пункт №6	11-14

Пункт №1

Реализация данной задачи представлена на рисунке №1. Она полностью соответствует описанию тз.

```
public static double integral(Function f, double leftX, double rightX, double step) { // 7 usages & zhetsok1
    if (f.getLeftDomainBorder() > leftX) throw new IllegalArgumentException("Incorrect left border!");
    if (f.getRightDomainBorder() < rightX) throw new IllegalArgumentException("Incorrect right domain border!");
    if (leftX > rightX) throw new IllegalArgumentException("Incorrect function!");
    if (step <= 0) throw new IllegalArgumentException("Step must be greater than 0!");

    if (leftX == rightX)
        return 0;

    double integralValue = 0;
    double currentX = leftX;

    while(currentX < rightX) {
        double nextX = Math.min(currentX + step, rightX); // Отлавливаем промежуток последнего шага, чтобы не вылезти за границы
        double currentStep = nextX - currentX; // Текущий шаг

        double f1 = f.getFunctionValue(currentX);
        double f2 = f.getFunctionValue(nextX);
        integralValue += (f1 + f2) * currentStep / 2.0; // Вычисление интеграла при помощи площадей трапеции

        currentX = nextX;
    }

    return integralValue;
}
```

Рисунок 1

Применение функции integral() и сравнение вычислений с теоретическим интегралом от exp на [0,1] представлены на рисунке №2.

```
Function exp = new Exp();
double theoryIntegral = Math.E - 1; // Значение интеграла от exp на [0,1]

double[] steps = {0.1, 0.01, 0.001, 0.0001};

System.out.println("Шаг\tИнтеграл\tПогрешность");
System.out.println("-----");

for (double step : steps) {
    double myIntegral = integral(exp, leftX: 0, rightX: 1, step);
    double error = Math.abs(myIntegral - theoryIntegral);

    System.out.printf("%.6f\t%.8f\t%.2e", step, myIntegral, error);

    if (error < 1e-7) {
        System.out.println(" <- Точность достигнута");
        break;
    } else {
        System.out.println();
    }
}
```

Рисунок 2

Пункт №2

После создания пакета threads я создал класс Task, реализация которого представлена на рисунке №3

```
public class Task { 14 usages  ↗ zhestok1

    private Function function;  2 usages
    private double leftX;  2 usages
    private double rightX;  2 usages
    private double step;  2 usages
    private int taskCount;  3 usages
```

Рисунок 3

Также я создал сеттеры и геттеры для переменных этого класса. На рисунке №4 представлена реализация функции nonThread().

```
public static void nonThread() { 1 usage  ↗ zhestok1
    System.out.println("-----NonThread Выполняется-----");

    Task task = new Task(taskCount: 100);
    for (int i = 0; i < task.getTaskCount(); i++) {
        double logBase = 1 + Math.random() * 9; // [1, 10)
        Function log = new Log(logBase);
        double leftX = Math.random() * 100;
        double rightX = 100 + Math.random() * 100;
        double step = Math.random();

        task.setFunction(log);
        task.setLeftX(leftX);
        task.setRightX(rightX);
        task.setStep(step);

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

        try {
            double integralValue = integral(log, leftX, rightX, step);
            System.out.println("Result " + "<" + task.getLeftX() + "> " +
                               "<" + task.getRightX() + "> " + "<" + task.getStep() + "> " + integralValue + ">");
        } catch (Exception e) {
            System.out.println("Error!" + e.getMessage());
        }
        System.out.println();
    }
}
```

Рисунок 4

Результат работы функции nonThread представлен на рисунке №5.

```
-----NonThread Выполняется-----
Source <53.91588454315184> <177.34068398715527> <0.6097379731859447>
Result <53.91588454315184> <177.34068398715527> <0.6097379731859447> <283.2884347237962>

Source <52.67294530754769> <110.73050211575327> <0.6933707056187429>
Result <52.67294530754769> <110.73050211575327> <0.6933707056187429> <234.61297042019567>

Source <26.2720951337153> <191.05141305847286> <0.7958831022348586>
Result <26.2720951337153> <191.05141305847286> <0.7958831022348586> <329.246473432046>
```

Рисунок 5

Пункт №3

Реализация классов SimpleGenerator и SimpleIntegrator на рисунках 6, 7, 8, 9.

```
public class SimpleGenerator implements Runnable { 1 usage  ↗zhestok1
    private Task task;  10 usages

    public SimpleGenerator(Task task) { 5 usages  ↗zhestok1
        this.task = task;
    }

    @Override  ↗zhestok1
    public void run() {
        System.out.println(Thread.currentThread().getName() + " запущен");

        for (int i = 0; i < task.getTaskCount(); i++) {
            double logBase = 1 + Math.random() * 9;
            Function log = new Log(logBase);
            double leftX = Math.random() * 100;
            double rightX = 100 + Math.random() * 100;
            double step = Math.random();

            task.setFunction(log);
            task.setLeftX(leftX);
            task.setRightX(rightX);
            task.setStep(step);
        }
    }
}
```

Рисунок 6

```

        System.out.println(Thread.currentThread().getName() + " Source "
            + "<" + task.getLeftX() + "> " +
            "<" + task.getRightX() + "> " +
            "<" + task.getStep() + "> ");

        // ★ ВАЖНО: Добавить задержку! ★
        try {
            Thread.sleep( millis: 10); // Даем время Integrator'у
        } catch (InterruptedException e) {
            break;
        }
    }

    task.setEnd(true);
    System.out.println(Thread.currentThread().getName() + " завершен");
}

```

Рисунок 7

Забегая вперед скажу, что флаг `isEnd` - метод которым я разрешил проблему ошибки `null`, которая будет представлена на рисунке 11. Каким образом? Сама ошибка возникала из-за того, что потоки рассинхронизированы и не обеспечивают корректный доступ к данным. В свою же очередь метод `isEnd()` обеспечивает безопасную проверку достижения конца данных, предотвращая обращение к `null`-объектам и гарантируя корректность многопоточных операций.

```

public class SimpleIntegrator implements Runnable { 1 usage  ↳ zhestok1
    private Task task; 12 usages

    public SimpleIntegrator(Task task) { 5 usages  ↳ zhestok1
        this.task = task;
    }

    @Override  ↳ zhestok1
    public void run() {
        System.out.println(Thread.currentThread().getName() + " запущен");
        int processed = 0;

        while (!task.isEnd() || processed < task.getTaskCount()) {
            if (task.getFunction() != null) {
                try {
                    double integralValue = integral(task.getFunction(),
                        task.getLeftX(), task.getRightX(), task.getStep());

                    System.out.println(Thread.currentThread().getName() + " Result " +
                        "<" + task.getLeftX() + "> " +
                        "<" + task.getRightX() + "> " +
                        "<" + task.getStep() + "> " +
                        "<" + integralValue + ">");
                }
            }
        }
    }
}

```

Рисунок 8

```
        processed++;
        task.setFunction(null);

    } catch (Exception e) {
        System.out.println(Thread.currentThread().getName() + " Error: " + e.getMessage());
        processed++;
    }
}

try {
    Thread.sleep( millis: 1); // Даем время Generator'у
} catch (InterruptedException e) {
    break;
}
}

System.out.println(Thread.currentThread().getName() + " завершен. Обработано: " + processed);
}
```

Рисунок 9

Реализация метода simpleThreads() представлена на рисунке 10.

```
public static void simpleThread() { 1 usage  & zhestok1
    System.out.println("===== SimpleThreads Выполняется =====");

    Task task = new Task( taskCount: 100);

    Thread generator = new Thread(new SimpleGenerator(task), name: "SimpleGenerator");
    Thread integrator = new Thread(new SimpleIntegrator(task), name: "SimpleIntegrator");

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

    try {
        generator.join( millis: 100);
        integrator.join( millis: 100);

    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }

    System.out.println("===== SimpleThreads Завершен =====");
}
```

Рисунок 10

Как и писал выше на рисунке №11 представлен результат ошибки NullPointerException. Решение по её исправлению тоже представлено выше.

```
Integrator запущен
Error: Cannot invoke "functions.Function.getLeftDomainBorder()" because "f" is null
```

Рисунок 11

Реализация классов Generator и Integrator представлено на рисунках 12 - 15.

```
public class Generator extends Thread { 2 usages  ↗ zhestok1
    private Task task; 9 usages
    private Semaphore semaphore; 3 usages

    public Generator(Task task, Semaphore semaphore) { 5 usages  ↗ zhestok1
        this.task = task;
        this.semaphore = semaphore;
    }

    @Override  ↗ zhestok1
    public void run() {
        System.out.println(getName() + " запущен");

        for (int i = 0; i < task.getTaskCount() && !isInterrupted(); i++) {
            try {
                // Используем tryAcquire с таймаутом
                if (semaphore.tryAcquire( timeout: 100, TimeUnit.MILLISECONDS)) {
                    try {
                        // Проверяем прерывание после захвата семафора
                        if (isInterrupted()) {
                            break;
                        }

                        double logBase = 1 + Math.random() * 9;
                        functions.Function log = new functions.basic.Log(logBase);
                        double leftX = Math.random() * 100;
                        double rightX = 100 + Math.random() * 100;
                        double step = Math.random();

                        task.setFunction(log);
                        task.setLeftX(leftX);
                        task.setRightX(rightX);
                        task.setStep(step);
                    }
                }
            }
        }
    }
}
```

Рисунок 12

```

        System.out.println(getName() + " Source " +
            "<" + task.getLeftX() + "> " +
            "<" + task.getRightX() + "> " +
            "<" + task.getStep() + "> ");

    } finally {
        semaphore.release();
    }
} else {
    // Таймаут - проверяем прерывание
    if (isInterrupted()) {
        break;
    }
}
} catch (InterruptedException e) {
    System.out.println(getName() + " был прерван");
    break;
} catch (Exception e) {
    System.out.println(getName() + " Error: " + e.getMessage());
}
}
System.out.println(getName() + " завершен");
}

```

Рисунок 13

```

public class Integrator extends Thread { 2 usages & zhestok1
    private Task task; 11 usages
    private Semaphore semaphore; 3 usages

    public Integrator(Task task, Semaphore semaphore) { 5 usages & zhestok1
        this.task = task;
        this.semaphore = semaphore;
    }

    @Override & zhestok1
    public void run() {
        System.out.println(getName() + " запущен");
        int processed = 0;

        while (processed < task.getTaskCount() && !isInterrupted()) {
            try {
                // Используем tryAcquire с таймаутом
                if (semaphore.tryAcquire(timeout: 100, TimeUnit.MILLISECONDS)) {
                    try {
                        // Проверяем прерывание после захвата семафора
                        if (isInterrupted()) {
                            break;
                        }

                        if (task.getFunction() != null) {
                            double integralValue = functions.Functions.integral(
                                task.getFunction(),
                                task.getLeftX(),
                                task.getRightX(),
                                task.getStep()
                            );

```

Рисунок 14

```

        System.out.println(getName() + " Result " +
            "<" + task.getLeftX() + "> " +
            "<" + task.getRightX() + "> " +
            "<" + task.getStep() + "> " +
            "<" + integralValue + ">");

        processed++;
        task.setFunction(null);
    }
} finally {
    semaphore.release();
}
} else {
    // Проверяем прерывание
    if (isInterrupted()) {
        break;
    }
}
} catch (InterruptedException e) {
    System.out.println(getName() + " был прерван");
    break;
} catch (Exception e) {
    System.out.println(getName() + " Error: " + e.getMessage());
    processed++; // Увеличиваем счетчик даже при ошибке
}
}
System.out.println(getName() + " завершен");
}

```

Рисунок 15

Кратенько о семафоре и его методах: Semaphore в Java — это примитив синхронизации, который позволяет ограничить количество потоков, одновременно получающих доступ к общему ресурсу. Он работает на основе счетчика: когда поток запрашивает доступ (acquire()), значение счетчика уменьшается; когда поток освобождает ресурс (release()), счетчик увеличивается. Если счетчик равен нулю, поток блокируется до тех пор, пока другой поток не освободит ресурс.

Реализация метода complicatedThreads() представлена на рисунках 16-17.

```

public static void complicatedThreads() { 1 usage & zhestok1
    System.out.println("===== ComplicatedThreads Выполняется =====");

    Task task = new Task(taskCount: 100);
    Semaphore semaphore = new Semaphore(permits: 1);

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

    // Установка приоритетов (экспериментируем)
    generator.setPriority(Thread.MAX_PRIORITY);      // Высший приоритет
    integrator.setPriority(Thread.NORM_PRIORITY);     // Обычный приоритет

    System.out.println("Приоритет Generator: " + generator.getPriority());
    System.out.println("Приоритет Integrator: " + integrator.getPriority());

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

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

```

Рисунок 16

```

System.out.println("Прерываем потоки после 50ms...");
generator.interrupt();
integrator.interrupt();

// Ждем завершения потоков после прерывания
try {
    generator.join(millis: 100);
    integrator.join(millis: 100);
} catch (InterruptedException e) {
    System.out.println("Основной поток был прерван");
}

// Проверяем, завершились ли потоки
System.out.println("Generator состояние: " + (generator.isAlive() ? "ЖИВ" : "ЗАВЕРШЕН"));
System.out.println("Integrator состояние: " + (integrator.isAlive() ? "ЖИВ" : "ЗАВЕРШЕН"));

System.out.println("===== ComplicatedThreads Завершен =====");

```

Рисунок 17

На рисунке 20 представлен кусочек результата выполнения данной функции

```
SimpleGenerator Source <48.13606759336958> <115.4905726078157> <0.25912159715011696>
SimpleIntegrator Result <48.13606759336958> <115.4905726078157> <0.25912159715011696> <156.70128706053563>
SimpleGenerator Source <70.91658868511043> <140.78010535122712> <0.4622727083578779>
SimpleIntegrator Result <70.91658868511043> <140.78010535122712> <0.4622727083578779> <297.7803852704798>
SimpleGenerator Source <96.87950081913044> <151.7872946278252> <0.880183260488174>
SimpleIntegrator Result <96.87950081913044> <151.7872946278252> <0.880183260488174> <137.41157551184463>
SimpleGenerator Source <28.946746945033897> <118.0311753605119> <0.8285140129631866>
SimpleIntegrator Result <28.946746945033897> <118.0311753605119> <0.8285140129631866> <176.8190577649053>
```

Рисунок 20