

Лабораторная работа №6
Манякин Степан 6204-010302D

Задание 1

Необходимо добавить 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. Вычисление значения интеграла выполняется по методу трапеций.

```
public static double Integrate(Function function, double leftBorder, double
rightBorder, double discretizationStep) {
    // Проверка корректности границ интегрирования
    if (leftBorder >= rightBorder) {
        throw new IllegalArgumentException("Левая граница интегрирования
должна быть меньше правой");
    }

    // Проверка, что границы интегрирования входят в область определения
    функции
    if (leftBorder < function.getLeftDomainBorder() || rightBorder >
function.getRightDomainBorder()) {
        throw new IllegalArgumentException("Границы интегрирования выходят за
область определения функции");
    }

    // Проверка корректности шага дискретизации
    if (discretizationStep <= 0) {
        throw new IllegalArgumentException("Шаг дискретизации должен быть
положительным");
    }

    double integral = 0.0;
    double currentX = leftBorder;

    // Проходим по всем полным шагам
    while (currentX + discretizationStep <= rightBorder) {
        double f1 = function.getFunctionValue(currentX);
        double f2 = function.getFunctionValue(currentX + discretizationStep);
        integral += (f1 + f2) * discretizationStep / 2.0;
        currentX += discretizationStep;
    }

    // Обрабатываем последний неполный шаг (если есть)
    if (currentX < rightBorder) {
        double lastStep = rightBorder - currentX;
        double f1 = function.getFunctionValue(currentX);
        double f2 = function.getFunctionValue(rightBorder);
        integral += (f1 + f2) * lastStep / 2.0;
    }

    return integral;
}
```

В методе main() проверяем работу метода интегрирования.

```
private static void testIntegration() {
    System.out.println("=== TEST INTEGRATION METHOD ===");

    // Создаем экспоненциальную функцию
    Exp expFunction = new Exp();

    // Теоретическое значение интеграла exp(x) от 0 до 1
```

```

        double theoreticalValue = Math.E - 1; //  $\int \exp(x) dx$  от 0 до 1 =  $\exp(1) - \exp(0) = e - 1$ 

        System.out.println("Теоретическое значение интеграла  $\exp(x)$  от 0 до 1: "
+ theoreticalValue);
        System.out.printf("Точное значение: %.10f\n", theoreticalValue);

        // Тестируем с разными шагами дискретизации
        double[] steps = {0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001};

        System.out.println("\nРезультаты интегрирования с разными шагами:");
        System.out.println("Шаг\t\tЗначение интеграла\t\tПогрешность");

        for (double step : steps) {
            try {
                double integralValue = Functions.Integrate(expFunction, 0, 1,
step);
                double error = Math.abs(integralValue - theoreticalValue);
                System.out.printf("%.6f\t%.10f\t%.10f\n", step, integralValue,
error);

                // Проверяем точность до 7 знака после запятой
                if (error < 1e-7) {
                    System.out.println("\n✓ Точность до 7 знака после запятой
достигнута при шаге: " + step);
                    break;
                }
            } catch (Exception e) {
                System.out.println("Ошибка при шаге " + step + ": " +
e.getMessage());
            }
        }

        // Дополнительный тест с очень маленьким шагом для демонстрации
        System.out.println("\n--- Проверка предельной точности ---");
        double fineStep = 1e-6;
        double fineIntegral = Functions.Integrate(expFunction, 0, 1, fineStep);
        double fineError = Math.abs(fineIntegral - theoreticalValue);
        System.out.printf("Шаг %.2e: интеграл = %.10f, погрешность = %.2e\n",
fineStep, fineIntegral, fineError);

        // Тестирование обработки ошибок
        System.out.println("\n=== TEST ERROR HANDLING ===");

        try {
            // Границы вне области определения
            Functions.Integrate(expFunction, -2, 1, 0.1);
        } catch (IllegalArgumentException e) {
            System.out.println("Поймано ожидаемое исключение: " +
e.getMessage());
        }

        try {
            // Неправильный порядок границ
            Functions.Integrate(expFunction, 1, 0, 0.1);
        } catch (IllegalArgumentException e) {
            System.out.println("Поймано ожидаемое исключение: " +
e.getMessage());
        }

        try {
            // Некорректный шаг
            Functions.Integrate(expFunction, 0, 1, -0.1);
        } catch (IllegalArgumentException e) {

```

```
System.out.println("Поймано ожидаемое исключение: " +  
e.getMessage());  
}  
}
```

Для метода трапеций погрешность имеет порядок $O(h^2)$, где h - шаг дискретизации.

Если мы хотим, чтобы погрешность была заметна в 7-м знаке (т.е. $> 1e-7$), то нужно найти такой шаг h , при котором:

$$h^2 > 1e-7$$

$$h > \sqrt{1e-7} \approx 0.000316$$

То есть при шаге больше чем ~ 0.00032 погрешность уже будет заметна в 7-м знаке после запятой.

А при шаге меньше чем 0.00032 погрешность будет меньше $1e-7$ и значения не будут отличаться в 7-м знаке.

Так что шаг > 0.00032 (например, 0.001)

Задание 2

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

В пакете `threads` опишите класс `Task`, объект которого должен хранить 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 functions.threads;  
  
import functions.Function;  
  
public class Task {  
    private Function function;  
    private double leftBorder;  
    private double rightBorder;  
    private double discretizationStep;  
    private int tasksCount;  
    private boolean dataProcessed = true; // true - данные обработаны, можно  
    генерировать новые  
  
    public Task() {  
    }  
  
    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 getDiscretizationStep() {
        return discretizationStep;
    }

    public void setDiscretizationStep(double discretizationStep) {
        this.discretizationStep = discretizationStep;
    }

    public int getTasksCount() {
        return tasksCount;
    }

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

    private volatile boolean newData = false;

    public synchronized void markDataAsNew() {
        newData = true;
    }

    public synchronized void markDataAsProcessed() {
        newData = false;
    }

    public synchronized boolean isNewData() {
        return newData;
    }

    public synchronized boolean isDataProcessed() {
        return dataProcessed;
    }

    public synchronized void setDataProcessed(boolean processed) {
        this.dataProcessed = processed;
    }
}

```

В главном классе программы опишите метод `nonThread()`, реализующий последовательную (без применения потоков инструкций) версию программы. В методе необходимо создать объект класса `Task` и установить в нём количество выполняемых заданий (минимум 100).

```

public static void nonThread() {
    System.out.println("=== NON-THREADED ===");

    // Создаем объект задания
    Task task = new Task();
    task.setTasksCount(100); // минимум 100 заданий

    for (int i = 0; i < task.getTasksCount(); i++) {
        // Случайное основание логарифма от 1 до 10 (исключая 1)
        double base = 1 + Math.random() * 9; // от 1.000... до 10.000...
        if (Math.abs(base - 1.0) < 1e-10) {
            base = 1.1; // гарантируем, что основание != 1
        }

        // Создаем логарифмическую функцию
        Log logFunction = new Log(base);

        // Устанавливаем параметры задания
        task.setFunction(logFunction);
        task.setLeftBorder(Math.random() * 100); // от 0 до 100
        task.setRightBorder(100 + Math.random() * 100); // от 100 до 200
        task.setDiscretizationStep(Math.random()); // от 0 до 1

        // Выводим исходные данные
        System.out.printf("Source %.6f %.6f %.6f\n",
            task.getLeftBorder(),
            task.getRightBorder(),
            task.getDiscretizationStep());

        try {
            // Вычисляем интеграл
            double result = Functions.Integrate(
                task.getFunction(),
                task.getLeftBorder(),
                task.getRightBorder(),
                task.getDiscretizationStep()
            );

            // Выводим результат
            System.out.printf("Result %.6f %.6f %.6f %.6f\n",
                task.getLeftBorder(),
                task.getRightBorder(),
                task.getDiscretizationStep(),
                result);

        } catch (IllegalArgumentException e) {
            System.out.printf("Error: %s for bounds [%.6f, %.6f] step
%.6f\n",
                e.getMessage(),
                task.getLeftBorder(),
                task.getRightBorder(),
                task.getDiscretizationStep());
        }
    }
}

```

Задание 3

В пакете threads создаем два следующих класса:

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 functions.threads;

import functions.basic.Log;

public class SimpleGenerator implements Runnable {
    private final Task task;

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

    @Override
    public void run() {
        for (int i = 0; i < task.getTasksCount(); i++) {
            // Генерируем случайные параметры
            double base = 1 + Math.random() * 9;
            if (Math.abs(base - 1.0) < 1e-10) {
                base = 1.1;
            }

            Log logFunction = new Log(base);
            double leftBorder = Math.random() * 100;
            double rightBorder = 100 + Math.random() * 100;
            double step = Math.random();

            // Синхронизация для устранения рассогласования данных
            synchronized (task) {
                // Устанавливаем параметры задания
                task.setFunction(logFunction);
                task.setLeftBorder(leftBorder);
                task.setRightBorder(rightBorder);
                task.setDiscretizationStep(step);

                // Выводим сообщение
                System.out.printf("Source %.6f %.6f %.6f\n", leftBorder,
rightBorder, step);
            }

            try {
                // Небольшая задержка для наглядности
                Thread.sleep(2);
            } catch (InterruptedException e) {
                System.out.println("Generator was interrupted");
                return;
            }
        }
    }
}
```

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 functions.threads;

import functions.Function;
import functions.Functions;

public class SimpleIntegrator implements Runnable {
    private final Task task;

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

    @Override
    public void run() {
        for (int i = 0; i < task.getTasksCount(); i++) {
            Function function;
            double leftBorder, rightBorder, step;

            // Синхронизация для устранения рассогласования данных
            synchronized (task) {
                // Получаем параметры задания
                function = task.getFunction();
                leftBorder = task.getLeftBorder();
                rightBorder = task.getRightBorder();
                step = task.getDiscretizationStep();
            }

            // Простое решение для избежания NullPointerException
            if (function == null) {
                // Если функция еще не установлена, пропускаем итерацию
                continue;
            }

            try {
                // Вычисляем интеграл
                double result = Functions.Integrate(function, leftBorder,
rightBorder, step);

                // Выводим результат
                System.out.printf("Result %.6f %.6f %.6f %.6f\n",
                    leftBorder, rightBorder, step, result);
            } catch (IllegalArgumentException e) {
                System.out.printf("Error: %s for bounds [%.6f, %.6f] step
%.6f\n",
                    e.getMessage(), leftBorder, rightBorder, step);
            }

            try {
                // Небольшая задержка для наглядности
                Thread.sleep(2);
            } catch (InterruptedException e) {
                System.out.println("Integrator was interrupted");
                return;
            }
        }
    }
}

```

В главном классе программы создаем метод `simpleThreads()`. В нём создаем объект задания, указываем количество выполняемых заданий (минимум 100),

создайте и запустите два потока вычислений, основанных на описанных классах SimpleGenerator и SimpleIntegrator.

```
public static void simpleThreads() {
    System.out.println("=== SIMPLE THREADS EXECUTION ===");

    // Создаем объект задания
    Task task = new Task();
    task.setTasksCount(100); // минимум 100 заданий

    // Создаем потоки
    Thread generatorThread = new Thread(new SimpleGenerator(task));
    Thread integratorThread = new Thread(new SimpleIntegrator(task));

    // Можно экспериментировать с приоритетами:
    //generatorThread.setPriority(Thread.MAX_PRIORITY);
    //integratorThread.setPriority(Thread.MIN_PRIORITY);

    // Запускаем потоки
    generatorThread.start();
    integratorThread.start();

    // Ожидаем завершения потоков
    try {
        generatorThread.join();
        integratorThread.join();
    } catch (InterruptedException e) {
        System.out.println("Main thread was interrupted");
    }

    System.out.println("=== SIMPLE THREADS EXECUTION COMPLETED ===");
}
```

Решенные проблемы:

1. NullPointerException - проверка if (function == null) в интеграторе
2. Рассинхронизация данных - использование synchronized (task) блоков
3. Короткие блокировки - синхронизация только на время чтения/записи параметров

В программе также есть возможность смены приоритетов:

```
// Создаем потоки
Thread generatorThread = new Thread(new SimpleGenerator(task));
Thread integratorThread = new Thread(new SimpleIntegrator(task));

// Можно экспериментировать с приоритетами:
//generatorThread.setPriority(Thread.MAX_PRIORITY);
//integratorThread.setPriority(Thread.MIN_PRIORITY);
```

Задание 4

В пакете threads создаем два следующих класса:

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 functions.threads;

import functions.Function;
import functions.basic.Log;
import java.util.concurrent.Semaphore;

public class Generator extends Thread {
    private final Task task;
    private final Semaphore semaphore;

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

    @Override
    public void run() {
        try {
            for (int i = 0; i < task.getTasksCount(); i++) {
                if (isInterrupted()) {
                    System.out.println("Generator was interrupted");
                    return;
                }

                // Ждем, пока предыдущие данные не будут обработаны
                synchronized (task) {
                    long waitStart = System.currentTimeMillis();
                    while (!task.isDataProcessed() && !isInterrupted()) {
                        long elapsed = System.currentTimeMillis() -
waitStart;

                        if (elapsed > 100) { // Максимум 100ms ждем
                            break; // Выходим из ожидания
                        }
                        task.wait(10);
                    }
                    if (isInterrupted()) return;
                }

                // Генерируем случайные параметры
                double base = 1 + Math.random() * 9;
                if (Math.abs(base - 1.0) < 1e-10) {
                    base = 1.1;
                }

                Log logFunction = new Log(base);
                double leftBorder = Math.random() * 100;
                double rightBorder = 100 + Math.random() * 100;
                double step = Math.random();

                // Захватываем семафор для записи
                semaphore.acquire();
                try {
                    // Устанавливаем параметры задания
                    task.setFunction(logFunction);
                    task.setLeftBorder(leftBorder);
                    task.setRightBorder(rightBorder);
                    task.setDiscretizationStep(step);
                }
            }
        }
    }
}
```

```

        task.setDataProcessed(false); // Помечаем как
необработанные

        // Выводим сообщение
        System.out.printf("Source %.6f %.6f %.6f\n", leftBorder,
rightBorder, step);
    } finally {
        semaphore.release();
    }
    //после semaphore.release():
    synchronized (task) {
        task.setDataProcessed(false);
    }

    Thread.sleep(2);
}
} catch (InterruptedException e) {
    System.out.println("Generator was interrupted during sleep or
semaphore acquisition");
}
}
}

```

Класс Integrator должен расширять 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 functions.threads;

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

public class Integrator extends Thread {
    private final Task task;
    private final Semaphore semaphore;

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

    @Override
    public void run() {
        try {
            for (int i = 0; i < task.getTasksCount(); i++) {
                if (isInterrupted()) {
                    System.out.println("Integrator was interrupted");
                    return;
                }

                Function function;
                double leftBorder, rightBorder, step;
                boolean shouldProcess = false;

                // Захватываем семафор для чтения
                semaphore.acquire();
                try {
                    // Получаем параметры задания

```

```

        function = task.getFunction();
        leftBorder = task.getLeftBorder();
        rightBorder = task.getRightBorder();
        step = task.getDiscretizationStep();

        // Проверяем, что данные еще не обработаны
        if (!task.isDataProcessed()) {
            shouldProcess = true;
            task.setDataProcessed(true); // Помечаем как
обработанные
        }
    } finally {
        semaphore.release();
    }
    // После semaphore.release():
    synchronized (task) {
        task.setDataProcessed(true);
        task.notifyAll();
    }

    // Пропускаем, если данные уже обработаны
    if (!shouldProcess || function == null) {
        continue;
    }

    try {
        // Вычисляем интеграл
        double result = Functions.Integrate(function, leftBorder,
rightBorder, step);

        // Выводим результат
        System.out.printf("Result %.6f %.6f %.6f %.6f\n",
            leftBorder, rightBorder, step, result);

    } catch (IllegalArgumentException e) {
        System.out.printf("Error: %s for bounds [%.6f, %.6f] step
%.6f\n",
            e.getMessage(), leftBorder, rightBorder, step);
    }

    Thread.sleep(2);
}
} catch (InterruptedException e) {
    System.out.println("Integrator was interrupted during sleep or
semaphore acquisition");
}
}
}

```

Отличие этих классов от предыдущих версий заключается в том, что вместо средств синхронизации в методах `run()` должны использоваться возможности семафора.

В главном классе программы создаем метод `complicatedThreads()`. В нём создаем объект задания, указываем количество выполняемых заданий (минимум 100), создаем и запускаем два потока вычислений классов `Generator` и `Integrator`.

```

public static void complicatedThreads() {
    System.out.println("=== COMPLICATED THREADS EXECUTION ===");

    // Создаем объект задания
    Task task = new Task();
    task.setTasksCount(100);

    // Создаем семафор (1 разрешение = взаимное исключение)
    Semaphore semaphore = new Semaphore(1);

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

    // Устанавливаем приоритеты (можно экспериментировать)
    generator.setPriority(Thread.MAX_PRIORITY);
    integrator.setPriority(Thread.MIN_PRIORITY);

    // Запускаем потоки
    generator.start();
    integrator.start();

    // Ждем 50ms и прерываем потоки
    try {
        Thread.sleep(50);
    } catch (InterruptedException e) {
        System.out.println("Main thread was interrupted");
    }

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

    // Ожидаем завершения потоков
    try {
        generator.join();
        integrator.join();
    } catch (InterruptedException e) {
        System.out.println("Main thread was interrupted while joining");
    }

    System.out.println("=== COMPLICATED THREADS EXECUTION COMPLETED ===");
}

```

Основной поток программы выжидает 50 миллисекунд, после чего прерывает работу потоков путём вызова метода `interrupt()`

Далее представлен вывод `main` с первыми 10 этапами работы каждого этапа, а `ComplicatedThreads` прерывается через 50 мс:

=== TEST INTEGRATION METHOD ===

Теоретическое значение интеграла $\exp(x)$ от 0 до 1: 1.718281828459045

Точное значение: 1,7182818285

Результаты интегрирования с разными шагами:

Шаг	Значение интеграла	Погрешность
-----	--------------------	-------------

0,100000	1,7197134914	0,0014316629
0,010000	1,7182961475	0,0000143190
0,001000	1,7182819716	0,0000001432
0,000100	1,7182818299	0,0000000014

✓ Точность до 7 знака после запятой достигнута при шаге: 1.0E-4

--- Проверка предельной точности ---

Шаг 1,00e-06: интеграл = 1,7182818284, погрешность = 2,11e-11

=== TEST ERROR HANDLING ===

Поймано ожидаемое исключение: Левая граница интегрирования должна быть меньше правой

Поймано ожидаемое исключение: Шаг дискретизации должен быть положительным

=== NON-THREADED ===

Source 5,164281 119,454344 0,188978
Result 5,164281 119,454344 0,188978 368,864783
Source 62,937039 188,355372 0,089930
Result 62,937039 188,355372 0,089930 275,142072
Source 84,287962 168,039331 0,206256
Result 84,287962 168,039331 0,206256 227,146989
Source 4,131297 167,214632 0,573175
Result 4,131297 167,214632 0,573175 626,026367
Source 72,233605 182,069564 0,921975
Result 72,233605 182,069564 0,921975 314,717558
Source 68,275463 179,461649 0,613940
Result 68,275463 179,461649 0,613940 321,272898
Source 20,655112 152,355557 0,998245
Result 20,655112 152,355557 0,998245 524,058655
Source 16,622509 170,864671 0,607303
Result 16,622509 170,864671 0,607303 306,017528
Source 96,508126 145,708458 0,232937
Result 96,508126 145,708458 0,232937 107,811420
Source 70,611496 179,979772 0,462856
Result 70,611496 179,979772 0,462856 942,575639

=== SIMPLE THREADS EXECUTION ===

Source 83,740409 109,823340 0,699343
Result 83,740409 109,823340 0,699343 105,565745
Source 79,442348 124,958591 0,030646
Result 79,442348 124,958591 0,030646 994,674259
Source 83,645570 115,937158 0,580480
Result 83,645570 115,937158 0,580480 68,792102
Source 35,728048 116,607277 0,122164
Result 35,728048 116,607277 0,122164 180,978660

Source 43,958223 197,464466 0,811313
Result 43,958223 197,464466 0,811313 314,511707
Source 31,802321 118,399784 0,864623
Result 31,802321 118,399784 0,864623 316,049853
Source 71,768760 167,814071 0,892180
Result 71,768760 167,814071 0,892180 214,235406
Source 74,260982 199,148627 0,376853
Result 74,260982 199,148627 0,376853 428,789258
Source 10,335787 125,589689 0,486712
Result 10,335787 125,589689 0,486712 251,344301
Source 99,315042 149,036264 0,005058
=== SIMPLE THREADS EXECUTION COMPLETED ===
=== COMPLICATED THREADS EXECUTION ===
Source 84,874432 180,377786 0,667505
Result 84,874432 180,377786 0,667505 3151,105364
Source 56,533702 154,710154 0,606732
Result 56,533702 154,710154 0,606732 27759,719527
Source 47,398576 157,325692 0,922849
Result 47,398576 157,325692 0,922849 236,415491
Source 53,555384 154,144695 0,621016
Result 53,555384 154,144695 0,621016 212,726877
Generator was interrupted during sleep or semaphore acquisition
=== COMPLICATED THREADS EXECUTION COMPLETED ===

Process finished with exit code 0

Спасибо за внимание