МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ

Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра «Программная инженерия»

Отчет по лабораторной работе № 7.

по дисциплине «Информационные технологии и программирование»

по разделу:

«7. Многопоточность»

Выполнила: студентка группы БПИ2401 Рябова Екатерина

Проверил: Харрасов Камиль Раисович

Москва

2025

Содержание

Цель работы:

Познакомиться с многопоточностью в Java, научиться использовать классы Thread и Runnable, библиотеку Executors и лямбда-выражения.

Ход работы:

Задание 1

Реализация многопоточной программы для вычисления суммы элементов массива.

Вариант 1. Создать два потока, которые будут вычислять сумму элементов массива по половинкам, после чего результаты будут складываться в главном потоке.

Реализуем класс ArraySummator, опираясь на предложенный в методичке пример с расширением класса Thread и переопределением метода run. Добавим у экзамепляров класса поля, где будем хранить часть массива, которую мы передаем в поток и счетчик суммы этого потока. В конструктор добавим параметр – массив. В методе run мы будем проходиться по элементам

массива и складывать их. Добавим метод, возвращающий сумму (вернуть в методе run мы ничего не можем, иначе это не будет считаться переопределением). В методе main мы считаем введенный из строки массив чисел, распарсим его в double. Создадим два потока (экземпляры класса ArraySummator), в качестве параметров передадим части массива (от 0 до половины и от половины до конца). Начнем оба потока, сджойним оба потока в главный (после их завершения) и затем вызовем метод getS для обоих потоков и сложим результаты. Обернем это все в блок try-catch.

Листинг 1.1. Программа, которая считывает текстовый файл и выводит на экран топ-10 самых часто встречающихся слов в это файле.

```
package seventhlab;

import java.util.Arrays;
import java.util.Scanner;

public class ArraySummator extends Thread {
    private double[] partOfArray;
    private  double s;
    public ArraySummator(double[] array) {
        partOfArray = array;
    }

    @Override
    public void run(){
        for (double elem : partOfArray){
            s += elem;
        }
        System.out.println(s);
    }

    public double getS(){
        return s;
    }

    public static void main(String[ ] args) throws InterruptedException {
        Scanner s = new Scanner(System.in);
        String input = s.nextLine();
        input = input.trim();
        String[] arr = input.substring(1, input.length() - 1).split(", ");
        double[] nums = new double[arr.length];
        try{
            for (int j = 0; j < arr.length; j++) {
                nums[j] = Double.parseDouble(arr[j].trim());
```

```
            }
            int len = nums.length;
            ArraySummator thread1 = new ArraySummator(Arrays.copyOfRange(nums, 0,
len / 2));
            ArraySummator thread2 = new ArraySummator(Arrays.copyOfRange(nums,
len / 2, len));
            thread1.start();
            thread2.start();
            thread1.join();
            thread2.join();
            double result1 = thread1.getS();
            double result2 = thread2.getS();
            System.out.println(result1 + result2);
        } catch (IllegalArgumentException e) {
            System.out.println("Ошибка данных: " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Ошибка вычисления: " + e.getMessage());
        }
    }
}
```

Результат представлен на рисунке 1:

```
PS C:\Users\user\.vscode\mtuci\java\labs\seventhlab> java ArraySummator.java
[1, 2, 3, 4, 5, 6, 7, 8, 9]
35.0
10.0
45.0
```

Рисунок 1. Результат выполнения программы и промежуточные результаты для каждого потока.

Задание 2

Реализация многопоточной программы для поиска наибольшего элемента в матрице.

Вариант 2. Создать пул потоков с помощью класса ExecutorService и разделить матрицу на равные части, каждую из которых будет обрабатывать отдельный поток. После завершения работы всех потоков результаты будут сравниваться в главном потоке для нахождения наибольшего элемента.

Реализуем класс MaxMatrix. У него будет одно статическое и волатилическое поле – макс. Идея использования пула потоков состоит в том, чтобы взаимодействовать с одной этой переменной max в синхранизированном методе обновления. То есть каждый поток будет проходиться по своему куску матрицы (в данном случае строчке) и сравнивать каждый её элемент с текущим значением максимума, если больше, значит меняем максимум, если нет - значит какой-то поток мог уже найти что-нибудь больше. В методе main соответственно реализуем ввод и парсинг матрицы, создание пулла потоков (с помощью экзекьютера). За максимум берем элемент [0][0] матрицы. Далее создаем построчный цикл, на каждую строчку создаем поток и фактически переопределяем метод run с помощью лямбда-выражения. В лямбда-выражении прописываем проход по заданной строке и выполнение для каждого элемента функции сравнения(обновления), где происходит изменение max, если элемент больше его. Далее заканчиваем процесс выполнения потоков, ждем пока все будет закончено и после этого проверяем, что лежит в max.

Листинг 2.1. Класс, который находит наибольший элемент в матрице без сравнения результатов потока.

```
package seventhlab;

import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MaxMatrix {
    public static volatile double max;

    public static void main(String[] args) throws InterruptedException {
        Scanner s = new Scanner(System.in);
        String input = s.nextLine().trim();
        String[] arrs = input.split("],");
        String[] firstElem = arrs[0].split(",");
        double[][] matrix = new double[arrs.length][firstElem.length];
        try{
            for (int i = 0; i < arrs.length; i ++){
                arrs[i] = arrs[i].trim().substring(1);
                int n = arrs[i].length();
                if (i == 0){
```

```
                arrs[i] = arrs[i].substring(1);
                n -= 1;
                } if (i == arrs.length - 1){
                    arrs[i] = arrs[i].substring(0, n - 2);
                    n -= 2;
                }
                String[] arr = arrs[i].trim().split(",");
                double[] nums = new double[arr.length];
                for (int j = 0; j < arr.length; j++) {
                    if (arr[j].trim().equals("")){
                        nums[j] = 0;
                    } else{
                        nums[j] = Double.parseDouble(arr[j].trim());
                    }
                }
                matrix[i] = nums;
            }
            s.close();
            ExecutorService executor =
Executors.newFixedThreadPool(matrix.length);
            max = matrix[0][0];
            for (double[] line : matrix) {
                executor.execute(() -> {
                    for (double elem : line){
                        updateMax(elem);
                    }
                });
            }
            executor.shutdown();
            while (!executor.isTerminated()) {
                Thread.sleep(10);
            }
            System.out.println(max);
        } catch (IllegalArgumentException e) {
            System.out.println("Ошибка данных: " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Ошибка вычисления: " + e.getMessage());
        }
    }

    public static synchronized void updateMax(double value) {
        if (value > max) {
            max = value;
        }
    }
}
```

Результаты выполнения представлены на рисунке 2.

```
PS C:\Users\user\.vscode\mtuci\java\labs\seventhlab> java MaxMatrix.java
[[1, 2, 3, 0], [-1, -2, 4, 5], [9, 10, 11, 2], [1, 2, 10, 8]]
11.0
PS C:\Users\user\.vscode\mtuci\java\labs\seventhlab> java MaxMatrix.java
[[-1, -3, -4], [-7, -5, -6]]
-1.0
```

Рисунок 2. Результат работы программы с синхронизированным методом и volatilированным максимумом.

Однако заметим, что мы не выполнили прямое требование сравнивая результатов потоков после их выполнения. Для того, чтобы сравнивать результаты, надо их где-то хранить, а экземпляры класса создавать не хочется. Поэтому создадим список (заполним его элементами, равными верхнему левому элементу матрицы) и каждый поток будет работать со своей ячейкой. Тогда перебор строк матрицы заменим циклом по i, но при этом нам придется ввести новую final-переменную row, которой мы каждую итерацию присваиваем значение i. (Так как переменные, используемые в лямбда-выражениях не должны меняться(локально)). После этого можем убрать класс, где мы обновляли максимум. В конце используем встроенный метод поиска максимума для коллекций, чтобы определить максимум среди потоков.

Листинг 2.2. Класс, который сравнивает максиумы по потокам в главном потоке.

```java
package seventhlab;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MaxMatrix2 {
    public static ArrayList<Double> maxArray = new ArrayList<>();

    public static void main(String[] args) throws InterruptedException {
        Scanner s = new Scanner(System.in);
        String input = s.nextLine().trim();
        String[] arrs = input.split("],");
        String[] firstElem = arrs[0].split(",");
        double[][] matrix = new double[arrs.length][firstElem.length];
```

```
        try{
            for (int i = 0; i < arrs.length; i ++){
                arrs[i] = arrs[i].trim().substring(1);
                int n = arrs[i].length();
                if (i == 0){
                arrs[i] = arrs[i].substring(1);
                n -= 1;
                } if (i == arrs.length - 1){
                    arrs[i] = arrs[i].substring(0, n - 2);
                    n -= 2;
                }
                String[] arr = arrs[i].trim().split(",");
                double[] nums = new double[arr.length];
                for (int j = 0; j < arr.length; j++) {
                    if (arr[j].trim().equals("")){
                        nums[j] = 0;
                    } else{
                        nums[j] = Double.parseDouble(arr[j].trim());
                    }
                }
                matrix[i] = nums;
            }
            s.close();
            ExecutorService executor =
Executors.newFixedThreadPool(matrix.length);
            double max = matrix[0][0];
            for (int i = 0; i < matrix.length; i ++){
                maxArray.add(max);
            }
            for (int i = 0; i < matrix.length; i ++) {
                double[] line = matrix[i];
                final int row = i;
                executor.execute(() -> {
                    for (double elem : line){
                        if (elem > maxArray.get(row)){
                            maxArray.set(row, elem);
                        }
                    }
                });
            }
            executor.shutdown();
            while (!executor.isTerminated()) {
                Thread.sleep(10);
            }
            System.out.println(Collections.max(maxArray));
        } catch (IllegalArgumentException e) {
            System.out.println("Ошибка данных: " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Ошибка вычисления: " + e.getMessage());
        }
```

```
    }
}
```

Результат работы программы представлен на рисунке 3.

```
PS C:\Users\user\.vscode\mtuci\java\labs\seventhlab> java MaxMatrix2.java
[[1, 2, 3, 0], [-1, -2, 4, 5], [9, 10, 11, 2], [1, 2, 10, 8]]
11.0
PS C:\Users\user\.vscode\mtuci\java\labs\seventhlab> java MaxMatrix2.java
[[-1, -3, -4], [-7, -5, -6]]
-1.0
```

Рисунок 3. Результат корректной работы программы

Задание 3

У вас есть склад с товарами, которые нужно перенести на другой склад. У каждого товара есть свой вес. На складе работают 3 грузчика. Грузчики могут переносить товары одновременно, но суммарный вес товаров, переносимый ими за одну итерацию, не может превышать 150 кг. Как только грузчики соберут 150 кг товаров, они отправятся на другой склад и начнут разгружать товары.

Напишите программу на Java, используя многопоточность, которая реализует данную ситуацию.

Вариант 3: Использование Lock и Condition. Используйте блокировки и условия для синхронизации работы грузчиков.

Отметим, что вся работа грузчиков состоит из двух этапов: погрузке данных (основная задача) и доставке (имитация). Введем два класса – Склад и Грузчик. Заметим, что грузчик – вложенный класс, то есть, он принадлежит классу. Зададим два conditions для каждой фазы работы. Смысл использования lock и conditions состоит в том, что каждый из грузчиков берет грузы по очереди (пока все остальные спят), как только он взял один груз, он будит всех остальных и (если еще не пора доставлять) ложиться спать. У каждого грузчика есть свой порядковый номер, а у нас есть очередь, каждый грузчик, когда проснулся, сверяет номер по очереди со своим, и, если он не совпал, то

грузчик засыпает. Так они ведут себя, пока 1) не набралось 150 кг ровно, 2) следующий товар уже не влезает (для того, чтобы гарантировать, что в этот заход грузчики точно больше ничего из оставшегося взять не могут, будем работать с отсортированным списком), 3) товары закончились. Во всех этих случаях мы завершаем погрузку и будим всех спящих (во всех фазах). Далее идет фаза доставки. Важно заметить, что едут доставлять только те, кто погрузил какие-то товары, остальные спят. Чтобы обеспечить синхронность во время каждой фазы мы считаем сколько грузчиков из троих погрузили товар (просто мониторим ситуацию, когда груз перестал быть равен 0 у конкретного грузчика) и сравниваем с тех, кто перешел на данную фазу, и погружаем всех в сон, пока на эту фазу не перейдут все грузчики, которым есть, что доставить. После этого все грузчики просыпаются и имитируют доставку. Последний, кто доставил груз, (определяем по кол-ву тех, кто должен был доставить и фактическому счетчику), сбрасывает все флаги так, чтобы перейти снова на фазу погрузки и передает очередь первому грузчику. Затем он будит всех, чтобы все тоже перешли на фазу погрузки. Если товаров после доставки больше нет, грузчики выйдут из фазы погрузки и завершат работу.

Листинг 3.1. Класс Product.

```
package seventhlab;

import java.util.*;
import java.util.concurrent.locks.*;

public class Warehouse {

    static final double MAX = 150;

    static Lock lock = new ReentrantLock();
    static Condition loadPhase = lock.newCondition();
    static Condition deliverPhase = lock.newCondition();

    static List<Double> items;
    static int index = 0;

    static double total = 0;
    static boolean loading = true;
```

```
    static boolean finished = false;

    static int activeLoaders = 0;
    static int delivered = 0;
    static int turn = 1;

    static class Loader implements Runnable {
        String name;
        int myTurn;
        double load = 0;

        Loader(String name, int turn) {
            this.name = name;
            this.myTurn = turn;
        }

        @Override
        public void run() {
            try {
                while (!finished) {
                    while (loading && !finished) {
                        lock.lock();
                        try {
                            while (turn != myTurn && !finished) {
                                loadPhase.await();
                                if (!loading || finished) break;
                            }

                            if (!loading || finished) break;

                            if (index >= items.size()) {
                                System.out.println(name + ": товары кончились");
                                if (total > 0) {
                                    loading = false;
                                    loadPhase.signalAll();
                                    deliverPhase.signalAll();
                                } else {
                                    finished = true;
                                    loadPhase.signalAll();
                                    deliverPhase.signalAll();
                                    return;
                                }
                                break;
                            }

                            double w = items.get(index);
                            if (total + w <= MAX) {
                                total += w;
                                if (load == 0) {
                                    activeLoaders++;
```

```
                            }
                            load += w;
                            index++;
                            System.out.println(name + " взял " + w + " кг.
Всего: " + total);

                            if (total == MAX) {
                                loading = false;
                                loadPhase.signalAll();
                                deliverPhase.signalAll();
                                break;
                            } else if (index < items.size()) {
                                double nextW = items.get(index);
                                if (total + nextW > MAX) {
                                    loading = false;
                                    loadPhase.signalAll();
                                    deliverPhase.signalAll();
                                    break;
                                } else {
                                    turn = (turn % 3) + 1;
                                    loadPhase.signalAll();
                                }
                            } else {
                                turn = (turn % 3) + 1;
                                loadPhase.signalAll();
                            }
                        } else {
                            loading = false;
                            loadPhase.signalAll();
                            deliverPhase.signalAll();
                            break;
                        }

                    } finally {
                        lock.unlock();
                    }
                }

                if (finished) break;

                lock.lock();
                try {
                    while (loading && !finished) {
                        deliverPhase.await();
                    }
                    if (finished) break;
                } finally {
                    lock.unlock();
                }
```

```
                    if (load > 0) {
                        System.out.println(name + " везет " + load + " кг");
                        Thread.sleep(300);
                        System.out.println(name + " вернулся");
                    }

                    lock.lock();
                    try {
                        if (finished) break;

                        if (load > 0) {
                            delivered++;
                        }

                        boolean iAmLast = (delivered >= activeLoaders &&
activeLoaders > 0);

                        if (iAmLast) {
                            delivered = 0;
                            activeLoaders = 0;
                            total = 0;
                            loading = true;
                            turn = 1;
                            deliverPhase.signalAll();
                            loadPhase.signalAll();
                        } else {
                            deliverPhase.await();
                        }

                        load = 0;

                    } finally {
                        lock.unlock();
                    }
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Scanner s = new Scanner(System.in);
        items = new ArrayList<>();

        for (String p : s.nextLine().split(",")) {
            items.add(Double.parseDouble(p.trim()));
        }

        Collections.sort(items);
```

```
        System.out.println("Товары: " + items);

        Thread t1 = new Thread(new Loader("Грузчик 1", 1));
        Thread t2 = new Thread(new Loader("Грузчик 2", 2));
        Thread t3 = new Thread(new Loader("Грузчик 3", 3));

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        System.out.println("Все перевезено");
    }
}
```

Результат работы программы представлен на рисунке 4:

```
PS C:\Users\user\.vscode\mtuci\java\labs\seventhlab> java Warehouse.java
30, 30, 30, 30, 30, 140
Товары: [30.0, 30.0, 30.0, 30.0, 30.0, 140.0]
Грузчик 1 взял 30.0 кг. Всего: 30.0
Грузчик 2 взял 30.0 кг. Всего: 60.0
Грузчик 3 взял 30.0 кг. Всего: 90.0
Грузчик 1 взял 30.0 кг. Всего: 120.0
Грузчик 2 взял 30.0 кг. Всего: 150.0
Грузчик 2 везет 60.0 кг
Грузчик 1 везет 60.0 кг
Грузчик 3 везет 30.0 кг
Грузчик 3 вернулся
Грузчик 1 вернулся
Грузчик 2 вернулся
Грузчик 1 взял 140.0 кг. Всего: 140.0
Грузчик 2: товары кончились
Грузчик 1 везет 140.0 кг
Грузчик 1 вернулся
Грузчик 1: товары кончились
Все перевезено
```

Рисунок 4. Результат корректной работы программы.

Работа загружена на гитхаб по ссылке: https://github.com/Kateriabova/Riaboava_Kate_BPI2401_IT2.git

Ответы на контрольные вопросы:

1. **Как реализуется многопоточностьесть в Java?**

Многопоточность в Java реализуется через:

- Класс Thread (наследование)
- Интерфейс Runnable (реализация)
- Интерфейс Callable (с возвращаемым значением)
- Использование ExecutorService и пулов потоков
- Использование библиотеки Executors и лямбда-выражений

2. **Что такое поток?**

Поток (thread) — это легковесный процесс, наименьшая единица выполнения в Java. Потоки разделяют общее адресное пространство процесса, но имеют собственные стеки вызовов и регистры. JVM позволяет создавать и управлять множеством потоков параллельно.

3. **Для чего нужно ключевое слово synchronized?**

synchronized используется для:

- Обеспечения взаимного исключения (mutual exclusion) доступа к общим ресурсам
- Гарантии атомарности операций
- Синхронизации потоков при доступе к критическим секциям
- Предотвращения состояния гонки (race condition)

public synchronized void method() { ... }

4. **Для чего нужно ключевое слово volatile?**

volatile обеспечивает:

- Видимость изменений переменной между потоками (чтение из основной памяти)
- Запрет на кэширование переменной в регистрах процессора

- Упорядочивание операций (happens-before)

Но не гарантирует атомарность сложных операций.

5. **Зачем нужно синхронизировать потоки?**

Синхронизация нужна для:

- Предотвращения race condition при доступе к общим данным
- Обеспечения консистентности данных
- Реализации взаимодействия между потоками (producer-consumer)
- Избежания deadlock и livelock ситуаций

6. **Какие есть способы синхронизации потоков?**
   - Мониторы объектов (synchronized методы и блоки)
   - Атомарные классы (AtomicInteger, AtomicReference, etc.)
   - Lock интерфейсы (ReentrantLock, ReadWriteLock)
   - Синхронизаторы (CountDownLatch, CyclicBarrier, Semaphore, Exchanger)
   - Потокобезопасные коллекции (ConcurrentHashMap, CopyOnWriteArrayList)
   - Ключевое слово volatile (ограниченная синхронизация)
7. **В чем разница между Thread и Runnable?**

Thread	**Runnable**
Класс	Интерфейс
Наследование (extends)	Реализация (implements)
Одно наследование в Java	Можно реализовать несколько интерфейсов
Менее гибкий	Более гибкий и рекомендуемый подход
start() создает новый поток	Требует передачи в конструктор Thread

Для Runnable нет ограничения единственного наследования, есть возможность использования пулов потоков.

8. **Какие состояния может иметь поток? Опишите жизненный цикл потока.**

Состояния потока:

- NEW — создан, но не запущен (new Thread())
- RUNNABLE — готов к выполнению или выполняется (start())
- BLOCKED — заблокирован, ждет монитор
- WAITING — ждет сигнала без таймаута (wait(), join())
- TIMED_WAITING — ждет с таймаутом (sleep(), wait(timeout))
- TERMINATED — завершил выполнение

Жизненный цикл: создание, выполнение, ожидание, блокировка и завершение.

9. Что такое daemon-поток? Как его создать?

Daemon-поток — фоновый поток, который автоматически завершается когда все не-daemon потоки завершились. Не мешает завершению JVM.

Создание:

```
Thread daemonThread = new Thread(() -> { ... });
daemonThread.setDaemon(true); // ДО start()!
daemonThread.start();
```

10. Как принудительно остановить поток?

Устаревшие методы (deprecated):

- stop() — опасен, может оставить данные в некорректном состоянии
- suspend() и resume() — могут вызвать deadlock

Современные пордходы:

- Флаг завершения:

```
volatile boolean running = true;

public void run() {
   while(running) { ... }
}
```

```
public void stop() {
   running = false;
}
```

- Прерывание:

```
thread.interrupt();

public void run() {
   while(!Thread.currentThread().isInterrupted()) { ... }
}
```

- Future.cancel() при использовании ExecutorService

11.Как работает метод join()? Для чего он используется?

join() заставляет текущий поток ждать завершения другого потока:

```
Thread t = new Thread(() -> { ... });
t.start();
t.join(); // текущий поток ждет завершения t
```

Используется для:

- Ожидания завершения параллельных вычислений
- Синхронизации порядка выполнения потоков
- Сбора результатов из дочерних потоков

12.Что такое «гонка данных» (race condition)?

Race condition — ошибка проектирования многопоточной системы, когда результат выполнения зависит от порядка выполнения потоков:

```
// Два потока инкрементируют одну переменную
if (count == 10) {  // Поток 1: count = 10
   count++;       // Поток 2 тоже считает count = 10
}                 // Результат: 12 вместо 11!
```

13. Что такое deadlock? Как его избежать?

Deadlock — ситуация, когда два или более потока бесконечно ждут друг друга, заблокировав взаимно нужные ресурсы.

Условия deadlock (Коффмана):

- Взаимное исключение
- Удержание и ожидание
- Отсутствие вытеснения
- Циклическое ожидание

Как избежать:

- Упорядочивание блокировок (всегда в одном порядке)
- Использование таймаутов (tryLock(timeout))
- Избегание вложенных блокировок
- Использование Thread.join() вместо взаимных блокировок

14. Что такое wait(), notify() и notifyAll()? В каком классе они объявлены?

Эти методы объявлены в классе Object и используются для взаимодействия потоков:

- wait() — освобождает монитор и переводит поток в WAITING
- notify() — будит один случайный поток, ждущий на этом мониторе
- notifyAll() — будит все потоки, ждущие на этом мониторе

Все эти методы должны вызываться внутри synchronized блока.

```
synchronized(lock) {
   while(!condition) {
      lock.wait();  // освобождает lock и ждет
   }
   // работа
   lock.notifyAll(); // будит ждущие потоки
}
```

15. Что такое ThreadPool? Какие реализации ExecutorService есть в Java?

ThreadPool — пул переиспользуемых потоков для выполнения задач. Эффективнее создания новых потоков для каждой задачи:

Реализации ExecutorService:

- Executors.newFixedThreadPool(n) — фиксированное количество потоков
- Executors.newCachedThreadPool() — создает потоки по мере необходимости, переиспользует
- Executors.newSingleThreadExecutor() — один поток (очередь задач)
- Executors.newScheduledThreadPool(n) — пул для планирования задач
- Executors.newWorkStealingPool() — work-stealing алгоритм (параллелизм)

```
ExecutorService executor = Executors.newFixedThreadPool(4);
executor.submit(() -> { /* задача */ });
executor.shutdown();
```

Преимущества:

- Снижение накладных расходов на создание потоков
- Контроль над количеством параллельных задач
- Очередь задач для обработки
- Управление жизненным циклом потоков

Вывод:

В ходе работы мы изучили многопоточность и её реализации в Java и научились использовать лямбда-выражения.