МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ордена трудового Красного Знамени федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Кафедра Математическая кибернетика и информационные технологии

Отчет по информационным технологиям и программированию

на тему: **«Многопоточность»**

Выполнил: студент группы БПИ2403

ФИО: Сон Владимир Сергеевич

Руководитель: Рыбаков Егор Дмитриевич

Москва, 2025

Цель работы: Изучить основы многопоточности в Java, включая создание и управление потоками с помощью классов Thread и Runnable, освоить механизмы синхронизации для безопасного доступа к общим ресурсам, а также познакомиться с современными подходами к работе с потоками через интерфейсы Callable, Future и фреймворк ExecutorService.

Задание 1.

Реализация многопоточной программы для вычисления суммы элементов массива. Создать два потока, которые будут вычислять сумму элементов массива по половинкам, после чего результаты будут складываться в главном потоке.

Задание 2.

Реализация многопоточной программы для поиска наибольшего элемента в матрице. Создать несколько потоков, каждый из которых будет обрабатывать свою строку матрицы. После завершения работы всех потоков результаты будут сравниваться в главном потоке для нахождения наибольшего элемента.

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.

Ход работы:

```java
public class ArraySumThreads {
    private static int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
    private static int sum1 = 0;
    private static int sum2 = 0;
    private static int totalSum = 0;

    static class FirstHalfThread extends Thread {
        @Override
        public void run() {
            int localSum = 0;
            int halfLength = array.length / 2;

            for (int i = 0; i < halfLength; i++) {
                localSum += array[i];
            }
            sum1 = localSum;
        }
    }

    static class SecondHalfThread extends Thread {
        @Override
        public void run() {
            int localSum = 0;
            int halfLength = array.length / 2;

            for (int i = halfLength; i < array.length; i++) {
                localSum += array[i];
            }
            sum2 = localSum;
        }
    }

    public static void main(String[] args) {
        FirstHalfThread thread1 = new FirstHalfThread();
        SecondHalfThread thread2 = new SecondHalfThread();

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        totalSum = sum1 + sum2;
        System.out.println(totalSum);
    }
```

```java
public class MatrixMaxThreads {
    private static int[][] matrix = {
        {12, 45, 23, 67, 89},
        {34, 78, 91, 15, 42},
        {56, 29, 73, 88, 11},
        {98, 62, 37, 51, 24}
    };

    private static int[] threadMaxValues;
    private static int globalMax;

    static class RowMaxThread extends Thread {
        private int rowIndex;
        private int threadId;

        public RowMaxThread(int rowIndex, int threadId) {
            this.rowIndex = rowIndex;
            this.threadId = threadId;
        }

        @Override
        public void run() {
            int maxInRow = matrix[rowIndex][0];

            for (int j = 1; j < matrix[rowIndex].length; j++) {
                if (matrix[rowIndex][j] > maxInRow) {
                    maxInRow = matrix[rowIndex][j];
                }
            }

            threadMaxValues[threadId] = maxInRow;
        }
    }

    public static void main(String[] args) {
        int rows = matrix.length;
        threadMaxValues = new int[rows];

        RowMaxThread[] threads = new RowMaxThread[rows];

        for (int i = 0; i < rows; i++) {
            threads[i] = new RowMaxThread(i, i);
            threads[i].start();
        }
```

```
        try {
            for (int i = 0; i < rows; i++) {
                threads[i].join();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        globalMax = threadMaxValues[0];
        for (int i = 0; i < threadMaxValues.length; i++) {
            if (threadMaxValues[i] > globalMax) {
                globalMax = threadMaxValues[i];
            }
        }
        System.out.println(globalMax);
    }
}
```

```java
import java.util.concurrent.*;
import java.util.ArrayList;
import java.util.List;

public class WarehouseSimulation {
    static class Product {
        private String name;
        private int weight;

        public Product(String name, int weight) {
            this.name = name;
            this.weight = weight;
        }

        public String getName() {
            return name;
        }

        public int getWeight() {
            return weight;
        }

        @Override
        public String toString() {
            return name + " (" + weight + " кг)";
        }
    }

    static class Warehouse {
        private List<Product> products;
        private int maxWeight = 150;

        public Warehouse() {
            products = new ArrayList<>();
            products.add(new Product("Телевизор", 45));
            products.add(new Product("Холодильник", 80));
            products.add(new Product("Стиральная машина", 70));
            products.add(new Product("Микроволновка", 25));
            products.add(new Product("Диван", 95));
            products.add(new Product("Шкаф", 120));
            products.add(new Product("Стол", 40));
            products.add(new Product("Кресло", 30));
            products.add(new Product("Пылесос", 15));
        }

        public List<Product> getProducts() {
            return products;
        }
```

```
    public int getMaxWeight() {
        return maxWeight;
    }
}

static class Loader implements Callable<LoaderResult> {
    private int loaderId;
    private List<Product> assignedProducts;
    private int maxWeight;

    public Loader(int loaderId, List<Product> assignedProducts, int maxWeight) {
        this.loaderId = loaderId;
        this.assignedProducts = assignedProducts;
        this.maxWeight = maxWeight;
    }

    @Override
    public LoaderResult call() throws Exception {
        int totalWeight = 0;
        List<Product> movedProducts = new ArrayList<>();

        for (Product product : assignedProducts) {
            if (totalWeight + product.getWeight() <= maxWeight) {;
                totalWeight += product.getWeight();
                movedProducts.add(product);

                System.out.println("Грузчик " + loaderId + " перенес " + product);
            } else {
                System.out.println("Грузчик " + loaderId + " превышен лимит веса: " + (totalWeight + product.getWeight()) + " > " + maxWeight + " кг");
            }
        }
        return new LoaderResult(loaderId, movedProducts, totalWeight);
    }
}

static class LoaderResult {
    private int loaderId;
    private List<Product> movedProducts;
    private int totalWeight;

    public LoaderResult(int loaderId, List<Product> movedProducts, int totalWeight) {
        this.loaderId = loaderId;
        this.movedProducts = movedProducts;
        this.totalWeight = totalWeight;
    }
```

```
        public int getLoaderId() {
            return loaderId;
        }

        public List<Product> getMovedProducts() {
            return movedProducts;
        }

        public int getTotalWeight() {
            return totalWeight;
        }
    }

    public static void main(String[] args) {
        Warehouse warehouse = new Warehouse();
        List<Product> products = warehouse.getProducts();

        ExecutorService executor = Executors.newFixedThreadPool(3);
        CompletionService<LoaderResult> completionService = new ExecutorCompletionService<>(executor);

        int productsPerLoader = products.size() / 3;
        int remainder = products.size() % 3;

        int startIndex = 0;
        for (int i = 0; i < 3; i++) {
            int endIndex = startIndex + productsPerLoader + (i < remainder ? 1 : 0);
            List<Product> assignedProducts = products.subList(startIndex, endIndex);

            Loader loader = new Loader(i + 1, assignedProducts, warehouse.getMaxWeight());
            completionService.submit(loader);

            startIndex = endIndex;
        }

        List<LoaderResult> results = new ArrayList<>();
        try {
            for (int i = 0; i < 3; i++) {
                Future<LoaderResult> future = completionService.take();
                LoaderResult result = future.get();
                results.add(result);
            }
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        executor.shutdown();
```

```
        executor.shutdown();

        int totalMovedProducts = 0;
        int totalWeight = 0;

        for (LoaderResult result : results) {
            System.out.println("Грузчик " + result.getLoaderId() + ":");
            System.out.println("Перенесено товаров: " + result.getMovedProducts().size());
            System.out.println("Общий вес: " + result.getTotalWeight() + " кг");
            System.out.println("Список товаров:");
            for (Product product : result.getMovedProducts()) {
                System.out.println("    - " + product);
            }

            totalMovedProducts += result.getMovedProducts().size();
            totalWeight += result.getTotalWeight();
        }
    }
}
```

Вывод: В ходе работы были изучены основные принципы многопоточности в Java, включая создание потоков через наследование класса Thread и реализацию интерфейса Runnable. Рассмотрены механизмы синхронизации (synchronized, Lock, volatile) для предотвращения конфликтов при доступе к общим ресурсам. Знание основ многопоточности позволяет писать более производительные и отзывчивые приложения, эффективно использующие возможности многоядерных процессоров.