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»

Отчёт по Лабораторной работе №7

По дисциплине: «Информационные технологии и программирование»

«Многопоточность»

Выполнила: студентка группы БПИ2401

Алексеева Татьяна Игоревна

Проверил: Харрасов Камиль Раисович

Москва

2025

Цель работы

Освоить практические навыки многопоточного программирования в Java, изучить создание и управление потоками выполнения через наследование от класса Thread, синхронизацию доступа к общим ресурсам и разработку параллельных алгоритмов для эффективной обработки данных в многопоточной среде.

Задание 1 (вариант 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.

```
import java.util.Arrays;

public class ArraySummator extends Thread {
    private double[] part;
    private double result;

    public ArraySummator(double[] arr) {
        part = arr;
    }

    @Override
    public void run() {
        result = 0;
        for (double elem : part) {
            result += elem;
        }
        System.out.println("Поток " + getName() + " посчитал сумму. Она равна: " + result);
    }

    public double getSum() {
        return result;
    }

    public static void main(String[] args) {
    double[] array = {12.0, 11.0, 15.9, 20.0, 22.22, 67.7};

    try {
        int middle = array.length / 2;
        double[] firstPart = Arrays.copyOfRange(array, 0, middle);
        double[] secondPart = Arrays.copyOfRange(array, middle, array.length);

        ArraySummator threadOne = new ArraySummator(firstPart);
        ArraySummator threadTwo = new ArraySummator(secondPart);

        threadOne.setName("Thread_1");
```

```
        threadTwo.setName("Thread_2");

        threadOne.start();
        threadTwo.start();

        threadOne.join();
        threadTwo.join();

        double resultOne = threadOne.getSum();
        double resultTwo = threadTwo.getSum();

        System.out.println(resultOne);
        System.out.println(resultTwo);
        System.out.println((resultOne + resultTwo));

        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Ошибка индексации массива: " + e.getMessage());
        } catch (InterruptedException e) {
            System.out.println("Поток был прерван: " + e.getMessage());
        }
    }
}
```

Будем решать задание через наследование класса Thread, это один из способов создания потоков (второй – реализация интерфейса Runnable). Преимущества нашего способа: проще начать работу, прямой доступ к методам класса Thread, не нужно создавать отдельный объект Thread.

Создаём поля класса Thread: part – часть массива, которую будет обрабатывать данный поток и result – результат вычислений (сумма элементов части массива).

Конструктор принимает массив чисел и сохраняет его в поле part. Каждый экземпляр ArraySummator будет работать со своей частью массива.

Метод run() – это то, что поток выполняет. Для корректной работы мы переопределяем метод run() класса Thread. Мы инициализируем переменную результата, циклом for-each проходимся по всем элементам массива, добавляем элемент к сумме, в конце выводим информацию о работе потока. Далее в коде мы не вызываем run() напрямую, его вызывает JVM после вызова start().

Так как поле result приватное, нам нужен геттер для получения результата вычислений.

В методе main создаём массив из 6 элементов. Вычисляем середину массива, с помощью целочисленного деления.

Разделяем массив с помощью Arrays.copeOfRange(array, 0, middle) – копирует элементы с индексами от 0 до 2, и Arrays.copeOfRange(array, 0, array.length). Для безопасности именно копируем, а не передаём ссылку.

Далее создаём два объекта-потока, передавая им разные части массива, устанавливаем имена. Запускаем потоки, вызывая метод start() (создаёт новый поток выполнения, помещает его в состояние RUNNABLE (готов к выполнению), JVM вызывает метод run() в новом потоке).

Метод join() – механизм синхронизации. Основной поток main() останавливается, ждёт завершения threadOne, после завершения threadOne ждёт завершения threadTwo, только потом продолжает выполнение.

Наконец получаем результаты из потоков и выводим их.

В конце также обрабатываем исключения: ArrayIndexOutOfBoundsException – ошибка при работе с индексами массива, InterruptedException – поток был прерван во время ожидания.

```
Поток Thread_1 посчитал сумму. Она равна: 38.9
Поток Thread_2 посчитал сумму. Она равна: 109.92
38.9
109.92
148.82
PS C:\Users\taale\OneDrive\Desktop\ITandP\LabWorkSeven>
```

Задание 2 (вариант 1)

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.

```java
public class MatrixMaxFinder extends Thread {
    private double[] row;
    private double rowMax;

    public MatrixMaxFinder(double[] row) {
        this.row = row;
    }

    @Override
    public void run() {
        rowMax = row[0];
        for (double elem : row) {
            if (elem > rowMax) {
                rowMax = elem;
            }
        }
        System.out.println("Поток " + getName() + " нащёл максимум. Он равен: " + rowMax);
```

```
    }

    public double getRowMax() {
        return rowMax;
    }

    public static void main(String[] args) {
        double[][] matrix = {
            {12.5, 8.3, 15.7, 4.2},
            {3.2, 25.1, 9.8, 18.4},
            {7.4, 18.6, 11.2, 30.5}
        };

        MatrixMaxFinder[] threads = new MatrixMaxFinder[matrix.length];

        for (int i = 0; i < matrix.length; i++) {
            threads[i] = new MatrixMaxFinder(matrix[i]);
            threads[i].setName("Thread_Row-" + (i + 1));
        }

        for (MatrixMaxFinder thread : threads) {
            thread.start();
        }

        try {
            for (MatrixMaxFinder thread : threads) {
                thread.join();
            }
        } catch (InterruptedException e) {
            System.out.println("Ошибка: " + e.getMessage());
            return;
        }

        double globalMax = threads[0].getRowMax();
        for (int i = 0; i < threads.length; i++) {
            double rowMax = threads[i].getRowMax();

            if (rowMax > globalMax) {
                globalMax = rowMax;
            }
        }

        System.out.println("Наибольший элемент во всей матрице: " + globalMax);
    }
}
```

Как и в предыдущем задании наследуем от Thread, это позволяет каждому экземпляру класса быть самостоятельным потоком. Создаём поля класс: row – одна строка матрицы (одномерный массив), rowMax – максимальный

элемент в этой строке. У каждого потока свой экземпляр этих полей, нет общих данных между потоками.

Конструктор принимает строку матрицы, каждому потоку передаётся своя строка для обработки.

В методе run() прописан алгоритм поиска максимума: инициализируем – предполагаем, что первый элемент максимальный, проходимся по всем элементам строки, если текущий элемент больше rowMax, обновляем значение. Информируем о результате.

Вызываем из главного потока геттер после завершения вычислений.

В методе main() прописываем матрицу (массив массивов). Создаём массив потоков, количество потоков = количеству строк матрицы.

Создаём потоки: new MatrixMaxFinder(matrix[i]) – передаём i-ю строку матрицы, matrix[i] – одномерный массив. Даём имена каждому потоку. Каждый поток получает ссылку на строку матрицы, а не копию.

Далее запускаем все потоки с помощью цикла for-each. Планировщик потоков ОС решает, когда и на каком ядре выполнять каждый поток.

Реализуем синхронизацию потоков с помощью join(): главный поток останавливается на каждом join(), ждёт завершения соответствующего потока.

В конце ищем глобальный максимум. Берём максимум из первой сроки как начальное значение. Далее проходимся по всем потокам: получаем максимум каждой строки через getRowMax(), сравниваем с текущим globalMax, обновляем, если нашли больше.

```
Поток Thread_Row-1 нащёл максимум. Он равен: 15.7
Поток Thread_Row-2 нащёл максимум. Он равен: 25.1
Поток Thread_Row-3 нащёл максимум. Он равен: 30.5
Наибольший элемент во всей матрице: 30.5
PS C:\Users\taale\OneDrive\Desktop\ITandP\LabWorkSeven>
```

Задание 3 (вариант 1)

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 одно временно, 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.
Использование Thread. Создайте классы Товар, Склад, и Грузчик. Каждый грузчик должен быть представлен в виде отдельного потока.

```
class Product {
    private String name;
    private double weight;

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

    public String getName() {
        return name;
    }

    public double getWeight() {
        return weight;
    }

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

Создаём класс Product, который представляет товар на кладе. Объявляем поля класса: name – название товара, weight – вес товара.

Конструкторе инициализирует поля при создании объекта.

Прописываем геттеры для доступа к приватным полям. Метод toString() возвращает строковое представление товара для вывода.

```
class Warehouse {
    private Product[] products;
    private int currentIndex = 0;

    public Warehouse(Product[] products) {
        this.products = products;
    }

    public synchronized Product getNextProduct() {
        if (currentIndex >= products.length) {
            return null;
        }

        Product product = products[currentIndex];
        currentIndex++;
        return product;
```

```
    }

    public boolean hasProducts() {
        return currentIndex < products.length;
    }

    public int getRemainingCount() {
        return Math.max(0, products.length - currentIndex);
    }
}
```

Класс Warehouse управляет товарами на кладе, обеспечивает потокобезопасный доступ.

Прописываем поля: products – массив всех товаров на складе, currentIndex – счётчик для отслеживания, какой товар выдавать следующим.

Метод getNextProduct(). synchronized гарантирует, что только один поток может выполнять этот метод в данный момент. currentIndex >= products.length — есть ли ещё товары. Берём товар по текущему индексу, увеличиваем индекс атомарно, возвращает товар или null, если товары закончились.

Прописываем вспомогательные методы: hasProducts() – проверяет, есть ли товары, getRemainingCount() – возвращает количество оставшихся товаров.

```
class Loader extends Thread {
    private String name;
    private Warehouse warehouse;
    private static double totalWeight = 0;
    private static final Object lock = new Object();

    public Loader(String name, Warehouse warehouse) {
        this.name = name;
        this.warehouse = warehouse;
    }

    @Override
    public void run() {
        System.out.println(name + " начал работу");

        while (warehouse.hasProducts()) {
            Product product = warehouse.getNextProduct();

            if (product == null) {
                break;
            }

            synchronized (lock) {
                if (totalWeight + product.getWeight() > 150) {
                    System.out.println(name + " сигнализирует, что набрано 150 кг. Грузчики едут
на другой склад.");
```

```
                    deliverToOtherWarehouse();
                    totalWeight = 0;
                }

                totalWeight += product.getWeight();
                System.out.println(name + " взял: " + product + " | Текущий общий вес: " +
String.format("%.2f", totalWeight) + " кг");
                if (Math.abs(totalWeight - 150.0) < 0.01) {
                    System.out.println(name + " сигнализирует: точно 150 кг. Грузчики едут на
другой склад.");
                    deliverToOtherWarehouse();
                    totalWeight = 0;
                }
            }

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        synchronized (lock) {
            if (totalWeight > 0 && !warehouse.hasProducts()) {
                System.out.println(name + " сигнализирует, что товары закончились. Последняя
партия " + String.format("%.2f", totalWeight) + " кг едет на другой склад.");
                deliverToOtherWarehouse();
                totalWeight = 0;
            }
        }

        System.out.println(name + " завершил работу");
    }

    private void deliverToOtherWarehouse() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Класс Loader представляет грузчика как поток выполнения.

Прописываем поля: name – имя грузчика, warehouse – ссылка на склад, totalWeight – статическое поле, то есть это поле одно на все экземпляры класса, все грузчики видят и изменяют одно и то же значение, именно здесь хранится общий вес партии, lock – объект для синхронизации, он один для всех экземпляров и используется в блоке synchronized.

Конструктор принимает имя и ссылку на клад.

Метод run() описывает, что делает каждый грузчик. Работает цикл while, пока на складе есть товары. Получаем товар warehouse.getNextProduct(), если вернул null, выходим из цикла.

Ключевая секция synchronized. Происходит захват монитора – только один грузчик может выполнять код в этом блоке: идёт проверка лимита, если добавление нового товара превысит 150 кг, то грузчик едет на другой склад, значение totalWeight сбрасывается до 0.

Если предел не набран, то добавляем вес взятого товара к текущему totalWeight, выводится информация, какой товар взяли и каков его вес. Проверяем точность совпадения: если вес ровно 150 кг, грузчики едут на другой склад, totalWeight обнуляется.

Завершение цикла – имитация времени, которое грузчики тратят на взятие товара.

Завершение работы грузчиков: проверяем остались ли товары, даже если вес меньше 150 кг, грузчики едут на другой склад, счётчик сбрасывается.

Метод deliverToOtherWarehouse() – имитация поездки на другой клад (задержка 500 мс).

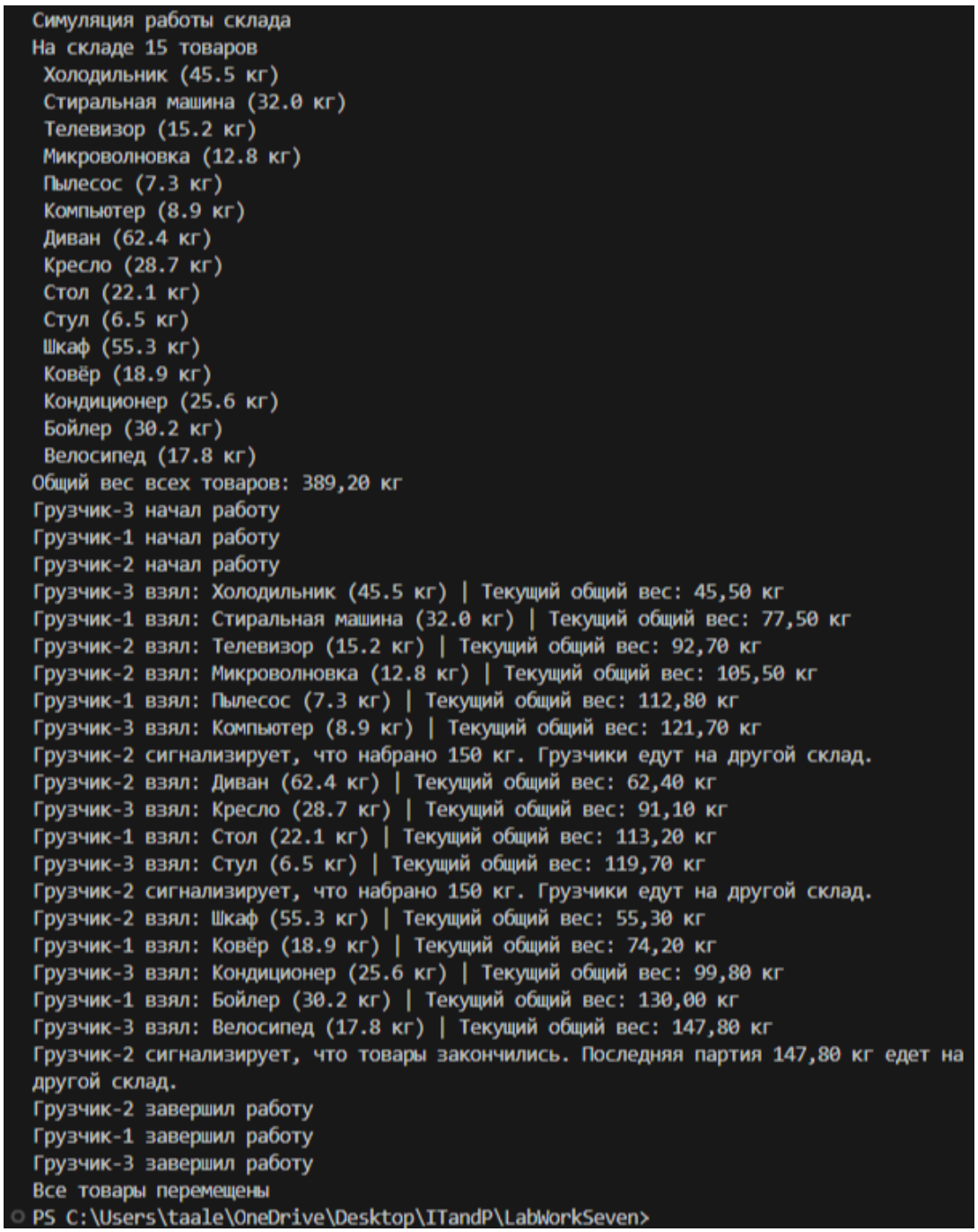
```
public class Main {
    public static void main(String[] args) {
        Product[] products = {
            new Product("Холодильник", 45.5),
            new Product("Стиральная машина", 32.0),
            new Product("Телевизор", 15.2),
            new Product("Микроволновка", 12.8),
            new Product("Пылесос", 7.3),
            new Product("Компьютер", 8.9),
            new Product("Диван", 62.4),
            new Product("Кресло", 28.7),
            new Product("Стол", 22.1),
            new Product("Стул", 6.5),
            new Product("Шкаф", 55.3),
            new Product("Ковёр", 18.9),
            new Product("Кондиционер", 25.6),
            new Product("Бойлер", 30.2),
            new Product("Велосипед", 17.8)
        };

        System.out.println("Симуляция работы склада");
        System.out.println("На складе " + products.length + " товаров");
        double totalWeight = 0;
        for (Product p : products) {
            System.out.println(" " + p);
```

```
            totalWeight += p.getWeight();
        }
        System.out.println("Общий вес всех товаров: " + String.format("%.2f",
totalWeight) + " кг");

        Warehouse warehouse = new Warehouse(products);

        Loader loader1 = new Loader("Грузчик-1", warehouse);
        Loader loader2 = new Loader("Грузчик-2", warehouse);
        Loader loader3 = new Loader("Грузчик-3", warehouse);

        loader1.start();
        loader2.start();
        loader3.start();

        try{
            loader1.join();
            loader2.join();
            loader3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Все товары перемещены");
    }
}
```

В классе main() создаём массив товаров: 15 товаров с разными весами. Далее показываем все товары и их общий вес.

Создаём объекты склада товаров (передаём массив товаров) и грузчиков (каждому передаём имя и ссылку на клад).

Метод start() создаёт новый потоки выполнения. Метод join() заставляет главный поток ждать завершения дочерних потоков.



Вывод

В ходе лабораторной работы были успешно освоены практические навыки многопоточного программирования в Java: изучены способы создания потоков через наследование от класса Thread, механизмы синхронизации доступа к общим ресурсам с использованием synchronized и AtomicInteger, а также разработаны параллельные алгоритмы для вычисления суммы массива, поиска максимума в матрице и симуляции работы склада с несколькими грузчиками. Работа продемонстрировала эффективность многопоточного подхода для распараллеливания вычислений и решения задач с взаимодействующими потоками, что является важным навыком для создания

высокопроизводительных приложений, оптимально использующих многоядерные процессоры.

Ответы на контрольные вопросы

1. Как реализуется многопоточность в Java?
   Ответ: многопоточность в Java реализуется через потоки (thread), которые представляют собой независимые последовательности выполнения внутри одного процесса. Java представляет два основных способа создания потоков: наследование от класса Thread и реализация интерфейса Runnable. Библиотеки ExecutorService и Executors для управления пулами потоков.
2. Что такое поток?
   Ответ: поток (Thread) – это наименьшая единица выполнения внутри процесса, представляющая собой независимую последовательность инструкций, которая может выполнятся параллельно с другими потоками.
3. Для чего нужно ключевое слово synchronized?
   Ответ: это ключевое слово в Java, которое обеспечивает синхронизацию доступа к общим ресурсам несколькими потоками. Оно предотвращает состояние гонки (race condition), когда несколько потоков одновременно изменяют одни и те же данные.
4. Для чего нужно ключевое слово volatile?
   Ответ: это ключевое слово в Java, которое обеспечивает видимость изменений переменной для всех потоков и предотвращает кэширование значения в регистрах процессора. Оно решает проблему несогласованного чтения переменных разными потоками.
5. Зачем нужно синхронизировать потоки?
   Ответ: синхронизация потоков нужна для предотвращения проблем при одновременном доступе нескольких потоков к общим ресурсам: race condition (гонка данных) — когда потоки одновременно читают/изменяют одни данные, результат становится непредсказуемым, повреждение данных (data corruption) — данные могут оказаться в несогласованном состоянии.
6. Какие есть способы синхронизации потоков?
   Ответ: synchronized (полная атомарность) — ключевое слово для методов или блоков кода, volatile (для флагов и простых перемменных) — для видимости изменений переменных между потоками, атомарные классы (для счётчиков без блокировок) — AtomicInteger, AtomicBoolean для атомарных операций, синхронизированные коллекции.
7. В чем разница между Thread и Runnable?
   Ответ: Thread — это класс, который сам является потоком выполнения и совмещает логику задачи с механизмом её запуска, тогда как Runnable — это интерфейс, описывающий только задачу (метод run()), которую

можно выполнить различными способами: через Thread, ExecutorService или даже в текущем потоке, что обеспечивает большую гибкость и соблюдение принципа разделения ответственности, поскольку отделяет логику задачи от механизма её многопоточного выполнения.

8. Какие состояния может иметь поток? Опишите жизненный цикл потока.
   Ответ: жизненный цикл потока включает 6 состояний: NEW – поток создан, но ещё не запущен, RUNNABLE – поток запущен, может выполнятся, BLOCKED – поток ждёт монитор для входа в synchronized блок, WAITING – поток ждёт сигнала, TIMED_WAITING – поток ждёт с ограничение времени, TERMINATED – поток завершил выполнение метода run().
9. Что такое daemon-поток? Как его создать?
   Ответ: это поток, который работает в фоновом режиме и автоматически завершается, когда все обычные потоки завершили работу. Он не препятствует завершению программы.
10. Как принудительно остановить поток?
    Ответ: в Java нельзя безопасно принудительно остановить поток, рекомендуемый способ – использовать флаг. Thread.stop() – устаревший, небезопасный, может повредить данные, Thread.interrupt() — устанавливает флаг прерывания, но поток должен сам проверять его, Флаг + interrupt() — рекомендуемый способ.
11. Как работает метод join()? Для чего он используется?
    Ответ: метод join() заставляет текущий поток ожидать завершения другого потока, на котором он вызывается. Это используется для синхронизации потоков, когда необходимо дождаться результатов выполнения одного потока, прежде чем продолжить работу в другом, особенно когда несколько потоков обрабатывают части задачи, результаты которой нужно объединить в главном потоке.
12. Что такое «гонка данных» (race condition)?
    Ответ: гонка данных (race condition) — это ошибка многопоточного программирования, возникающая, когда несколько потоков одновременно обращаются к общим данным и хотя бы один из потоков изменяет эти данные, что может привести к непредсказуемому и некорректному результату, поскольку конечное состояние данных зависит от порядка и времени выполнения потоков.
13. Что такое deadlock? Как его избежать?
    Ответ: ситуация, когда два или более потока бесконечно ждут друг друга, удерживая нужные друг другу ресурсы (блокировки), и не могут продолжить выполнение.
14. Что такое wait(), notify() и notifyAll()? В каком классе они объявлены?
    Ответ: это методы для координации потоков, они объявлены в классе Object: wait() — освобождает монитор и переводит поток в WAITING,

пока другой поток не вызовет notify(), notify() — будит один случайный поток, ожидающий на этом мониторе, notifyAll() — будит все потоки, ожидающие на этом мониторе.

15. Что такое ThreadPool? Какие реализации ExecutorService есть в Java?
    Ответ: ThreadPool (пул потоков) — это набор заранее созданных потоков, готовых выполнять задачи, что эффективнее создания нового потока для каждой задачи.
    Реализации ExecutorService в Java:
    - Executors.newFixedThreadPool(n) — фиксированное количество потоков
    - Executors.newCachedThreadPool() — создаёт новые по мере необходимости, переиспользует свободные
    - Executors.newSingleThreadExecutor() — один поток (очередь задач)
    - Executors.newScheduledThreadPool(n) — для отложенного/периодического выполнения
    - Executors.newWorkStealingPool() — пул с work-stealing алгоритмом (Java 8+)