МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ

Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Лабораторная работа № 7

«Многопоточность»

Выполнила: Студентка группы БВТ2306

Максимова Мария

Москва 2024

Задание 1:

Реализация многопоточной программы для вычисления суммы элементов массива

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

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

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, которые они переносят, 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 5. Использование CompletableFuture: Используйте CompletableFuture для асинхронного выполнения задачи переноса товаров

Задание 1

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

import static java.util.concurrent.Executors.newFixedThreadPool;

public class SumArray {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
```

```
        int[] array = {5, 10, 15, 20, 25, 30, 200, 400, 250};
        int numOfThreads = 3;

        int partSize = (int) Math.ceil((double) array.length / numOfThreads);

        ExecutorService executor = newFixedThreadPool(numOfThreads);

        Future<Integer>[] futures = new Future[numOfThreads];
        for (int i = 0; i < numOfThreads; i++) {
            int start = i * partSize;
            int end = Math.min(start + partSize, array.length);

            int[] subArray = new int[end - start];
            System.arraycopy(array, start, subArray, 0, end - start);

            futures[i] = executor.submit(() -> calculateSum(subArray));
        }

        int totalSum = 0;
        for (Future<Integer> future : futures) {
            totalSum += future.get();  // Получаем результат выполнения задач
        }

        executor.shutdown();

        System.out.println("Общая сумма элементов массива: " + totalSum);
    }

    public static int calculateSum(int[] array) {
        int sum = 0;
        for (int num : array) {
            sum += num;
        }
        return sum;
    }
}
```

Код написан для многопоточной обработки массива с использованием пула потоков (ExecutorService). Он разделяет массив на несколько частей, каждая из которых обрабатывается в отдельном потоке, а затем результаты суммируются в главном потоке.

---

1. Импорты:

```
import java.util.concurrent.*;
import static java.util.concurrent.Executors.newFixedThreadPool;
```

- **import java.util.concurrent.*;** — импортирует классы из пакета java.util.concurrent, которые используются для работы с многопоточностью, включая ExecutorService и Future.
- **import static java.util.concurrent.Executors.newFixedThreadPool;** — статический импорт метода newFixedThreadPool() для создания пула потоков фиксированного размера.

---

2. Метод main:

```
public class ArraySumExecutor {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        int[] array = {5, 10, 15, 20, 25, 30, 200, 400,250};
        int numOfThreads = 3;
```

Исключения **InterruptedException** и **ExecutionException** связаны с обработкой результатов асинхронных задач в пуле потоков. В этом коде они используются для контроля ошибок, возникающих во время выполнения задач, и ожидания их завершения.

- Создание массива: Массив содержит числа {5, 10, 15, 20, 25, 30, 200, 400,250}.
- Количество потоков: задаём количество потоков (numOfThreads = 3).

---

3. Определение размера частей массива:

```
        int partSize = (int) Math.ceil((double) array.length / numOfThreads);
```

- partSize — размер каждой части массива, рассчитанный с помощью Math.ceil() для корректного деления массива на части, даже если его длина не делится на количество потоков без остатка.

---

4. Создание пула потоков:

```
ExecutorService executor = newFixedThreadPool(numOfThreads);
```

- Создаёт пул потоков фиксированного размера (3 потока). Каждый поток будет обрабатывать часть массива.

---

5. Разделение массива на подмассивы и отправка задач на выполнение:

```
Future<Integer>[] futures = new Future[numOfThreads];
for (int i = 0; i < numOfThreads; i++) {
   int start = i * partSize;
   int end = Math.min(start + partSize, array.length);

   int[] subArray = new int[end - start];
       System.arraycopy(array, start, subArray, 0, end – start futures[i] =
    executor.submit(() -> calculateSum(subArray));
}
```

Future<Integer>[] futures = new Future[numOfThreads];

Эта строка создает массив объектов Future, которые будут использоваться для хранения результатов асинхронных задач, выполняемых в пуле потоков

- **Цикл for:** делит массив на равные части и отправляет каждую часть на обработку.
  - **start и end:** определяют начало и конец текущей части массива.
  - **int i = 0**: Инициализация переменной i (счетчик цикла), начинаем с 0.
  - **i < numOfThreads**: Условие выполнения цикла — пока i меньше количества потоков (numOfThreads), цикл продолжается.
  - **i++**: Шаг цикла — увеличиваем i на 1 после каждой итерации.

Переменная start определяет начальный индекс подмассива в исходном массиве array.

- **int**: Объявление переменной типа int (целое число).
- **start**: Имя переменной, которая будет хранить начальный индекс подмассива.

- =: Оператор присваивания (присваиваем значение переменной start).
- **i * partSize**: Вычисление начального индекса:
  - i — текущая итерация цикла (0, 1, 2...).
  - partSize — размер каждой части (подмассива), вычисленный заранее как array.length / numOfThreads.
  - Умножение i * partSize сдвигает начало подмассива для каждой итерации.

Переменная end определяет конечный индекс подмассива. Она используется, чтобы не выйти за пределы массива.

- **int**: Объявление переменной типа int (целое число).
- **end**: Имя переменной, которая будет хранить конечный индекс подмассива.
- =: Оператор присваивания.
- **Math.min(a, b)**: Метод из класса Math, который возвращает меньшее из двух чисел a и b.

Аргументы для Math.min:

1. **start + partSize**: Предполагаемый конец подмассива:
   - start — начало подмассива.
   - partSize — размер подмассива.
2. **array.length**: Длина исходного массива, чтобы не выйти за его пределы.

Если последний подмассива меньше, чем partSize, метод Math.min корректирует его конец, чтобы он не вышел за границы массива.

Math.min(start + partSize, array.length:

1. **int[] array = {5, 10, 15, 20, 25, 30, 200, 400, 250};**
   Длина массива array.length = 9.
2. **int numOfThreads = 3;**
   Количество потоков numOfThreads = 3.
3. **Вычислим partSize:**

   int partSize = array.length / numOfThreads;

   - partSize = 9 / 3 = 3
     Таким образом, каждый поток будет обрабатывать **3 элемента**.

Цикл:

Цикл делит массив на части, определяя start (начало подмассива) и end (конец подмассива).

Итерация 1 (i = 0):

- start = i * partSize → start = 0 * 3 = 0**.**
- end = Math.min(start + partSize, array.length) → end = Math.min(0 + 3, 9) = Math.min(3, 9) = 3.
- Результат: Подмассив начинается с индекса 0 и заканчивается на 3 (не включая 3).
  Получаем: {5, 10, 15}.

Итерация 2 (i = 1):

- start = i * partSize → start = 1 * 3 = 3.
- end = Math.min(start + partSize, array.length) → end = Math.min(3 + 3, 9) = Math.min(6, 9) = 6.
- Результат: Подмассив начинается с индекса 3 и заканчивается на 6 (не включая 6).
  Получаем: {20, 25, 30}.

Итерация 3 (i = 2):

- start = i * partSize → start = 2 * 3 = 6.
- end = Math.min(start + partSize, array.length) → end = Math.min(6 + 3, 9)= Math.min(9, 9) = 9.
- Результат: Подмассив начинается с индекса 6 и заканчивается на 9 (не включая 9).
  Получаем: {200, 400, 250}.

Вывод:

С использованием **Math.min** гарантируется, что подмассивы не выйдут за границы массива:

- **Итерация 1:** {5, 10, 15}
- **Итерация 2:** {20, 25, 30}
- **Итерация 3:** {200, 400, 250}

Таким образом, массив разделен на 3 части, каждая из которых содержит 3 элемента.

- **Создание подмассива:** System.arraycopy() копирует элементы из оригинального массива в новый подмассив.

array: Исходный массив, из которого копируются элементы.
start: Индекс начала копирования в исходном массиве.
subArray: Целевой массив, в который копируются элементы.
0: Начальный индекс в целевом массиве (subArray), куда будут помещены элементы.
end - start: Количество элементов для копирования (размер подмассива).

- **executor.submit()** отправляет задачу на выполнение в пул потоков.
- Лямбда-выражение **() -> calculateSum(subArray)** выполняет метод calculateSum для подмассива.
- Результат сохраняется в массив futures, и его можно получить позже с помощью **future.get().**

6. Сбор результатов и вычисление общей суммы:

```
int totalSum = 0;
for (Future<Integer> future : futures) {
    totalSum += future.get();
}
```

- **Массив futures:** хранит объекты Future, которые представляют результаты выполнения задач.
- **int totalSum** — это переменная для накопления общей суммы значений, полученных из всех выполненных потоков.

В конце цикла переменная totalSum содержит общий результат, который можно вывести или использовать в дальнейших расчётах.

- **future.get():** Блокирующий вызов, который возвращает результат выполнения задачи (сумму подмассива).

---

7. Завершение пула потоков и вывод результата:

```
        executor.shutdown();
        System.out.println("Общая сумма элементов массива: " + totalSum);
    }
```

- **executor.shutdown():** Останавливает пул потоков после завершения всех задач.
- **Вывод общей суммы:** Итоговая сумма всех элементов массива выводится на консоль.

---

8. Метод calculateSum():

```
    public static int calculateSum(int[] array) {
        int sum = 0;
        for (int num : array) {
            sum += num;
        }
        return sum;
    }
}
```

- Этот метод принимает массив и вычисляет сумму его элементов.
- Используется в каждой задаче, отправленной на выполнение пулом потоков.

Результат выполнения программы:

При запуске программы она делит массив на три части, вычисляет сумму каждой части в отдельном потоке и затем складывает их результаты. Итоговая сумма выводится на экран.

Пример вывода:

Общая сумма элементов массива: 955

Основные концепции:

1. **Многопоточность:** Параллельное выполнение задач увеличивает производительность.
2. **Пул потоков (ExecutorService):** управляет фиксированным числом потоков, предотвращая создание лишних потоков.
3. **Future:** используется для получения результатов выполнения задач в многопоточной среде.

Задание 2

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

public class TheBiggestElement {

   public static void main(String[] args) throws InterruptedException,
ExecutionException {
      int[][] arr = {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {20, 510, 932, 392},
            {100, 200, 150, 35}
      };

      int numOfRows = arr.length;

      ExecutorService executor = Executors.newFixedThreadPool(numOfRows);

      Future<Integer>[] futures = new Future[numOfRows];

      for (int i = 0; i < numOfRows; i++) {
         final int row = i;
         futures[i] = executor.submit(() -> findMaxElement(arr[row]));
      }

      int ArrayMax = Integer.MIN_VALUE;
      for (Future<Integer> future : futures) {
         int rowMax = future.get();
```

```
            System.out.println("Максимум строки: " + rowMax);

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

        executor.shutdown();
        System.out.println("Наибольший элемент в матрице: " + ArrayMax);
    }

    private static int findMaxElement(int[] row) {
        int max = row[0];
        for (int value : row) {
            if (value > max) {
                max = value;
            }
        }
        return max;
    }
}
```

Этот код находит наибольший элемент в каждой строке двумерного массива (int[][] arr) и выводит эти максимальные значения в отдельный поток.

java.util.concurrent.Future — это интерфейс Future, который представляет будущий результат асинхронного вычисления — результат, который в конечном итоге появится в Future после завершения вычисления.

---

1. **Объявление класса и метода:**

public class TheBiggestElement throws InterruptedException, ExecutionException {

Создаётся класс TheBiggestElement, содержащий методы для поиска максимальных элементов и их вывода.

---

2. **Метод main(String[] args)**

Точка входа в программу. Здесь создаётся двумерный массив.

```
public static void main(String[] args) {
   int[][] arr = {
         {1, 2, 3, 4},
         {5, 6, 7, 8},
         {20, 510, 932, 392},
         {100, 200, 150, 35}
   };

int numOfRows = arr.length;
```

- Определяется количество строк в массиве arr и сохраняется в переменной numOfRows.

---

3. **Создание пула потоков:**

```
ExecutorService executor = newFixedThreadPool(numOfRows);
```

- Создаёт пул потоков фиксированного размера (количество строк). Каждый поток будет обрабатывать часть массива.

---

4. **Создание массива:**

```
Future<Integer> [] futures = new Future[numOfRows];
```

- Это массив для хранения будущих объектов

---

5. **Цикл:**

Цикл по строкам массива:

```
for (int i = 0; i < numOfRows; i++) {
          final int row = i;
         futures[i] = executor.submit(() -> findMaxElement(arr[row]));
      }
```

- **for (int i = 0; i < numOfRows; i++) { ... }**: Это обычный цикл for, который перебирает каждую строку двумерного массива arr. numOfRows содержит количество строк в массиве.

- **final int row = i;**: Эта строка создает финальную копию счетчика цикла i.

- **futures[i] = executor.submit(() -> findMaxElement(arr[row]));**: Это ядро распараллеливания.

    - **executor.submit(...)**: Отправляет задачу в ExecutorService (executor). ExecutorService управляет пулом потоков, позволяя параллельное выполнение задач.

    - **(() -> findMaxElement(arr[row]))**: это лямбда-выражение. Это краткий способ определения анонимной функции. В данном случае функция не принимает аргументов (()), а её тело — findMaxElement(arr[row]).

    - **futures[i] = ...**: Результат executor.submit(...) (который является объектом Future) сохраняется в i-ом элементе массива futures.

В итоге, этот код разделяет задачу поиска максимального элемента в каждой строке двумерного массива между несколькими потоками, управляемыми ExecutorService. Каждая строка обрабатывается параллельно, что потенциально ускоряет общее вычисление. Массив futures используется для отслеживания результатов каждого потока.

Цикл по результатам обработки каждой строки:

- **int ArrayMax = Integer.MIN_VALUE;**: Инициализируется переменная ArrayMax наименьшим возможным значением для типа int. Это гарантирует, что первый найденный максимум строки будет больше, чем начальное значение ArrayMax.

- **for (Future<Integer> future : futures) { ... }**: Цикл перебирает все объекты Future в массиве futures. Каждый Future содержит результат (максимальный элемент) для одной строки массива.

- **int rowMax = future.get();**: Метод future.get() блокирует выполнение до тех пор, пока соответствующий поток не завершит свою работу и не вернет результат (максимальный элемент строки). Этот результат записывается в переменную rowMax.

- **System.out.println("Максимум строки: " + rowMax);**: Выводит на консоль максимальный элемент, найденный в текущей строке.

- **if (rowMax > ArrayMax) { ArrayMax = rowMax; }**: Сравнивает текущий максимум строки (rowMax) с текущим глобальным максимумом (ArrayMax). Если максимум строки больше глобального максимума, глобальный максимум обновляется.

- **executor.shutdown();**: Завершает работу ExecutorService. Это важно для корректного освобождения ресурсов. После вызова shutdown() новые задачи не могут быть отправлены в executor, но уже запущенные задачи продолжат свою работу.

- **System.out.println("Наибольший элемент в матрице: " + ArrayMax);**: После обработки всех строк, выводит на консоль наибольший элемент, найденный во всем двумерном массиве.

В целом, этот код собирает результаты параллельных вычислений, ищет наибольшее значение среди всех найденных максимальных элементов строк и корректно завершает работу потоков. Он обеспечивает последовательное получение результатов и нахождение глобального максимума в массиве.

---

6. Метод findMaxElement(int[][] row)

Этот метод находит наибольший элемент в каждой строке двумерного массива

- **int max = row[0];**: Инициализируется переменная max первым элементом массива row. Это начальное предположение о максимальном значении.

- **for (int value: row) { ... }**: Это цикл for-each, который перебирает каждый элемент массива row. В каждой итерации переменная value принимает значение текущего элемента.

- **if (value > max) { max = value; }**: Внутри цикла происходит сравнение. Если текущий элемент value больше текущего максимального значения max, то max обновляется до значения value.

- **return max;**: После прохода по всем элементам массива, функция возвращает найденное максимальное значение.

Вывод программы:

4
8
932
200

Каждое число — это максимальный элемент соответствующей строки исходного массива:

1. Максимум в строке {1, 2, 3, 4} — **4**.
2. Максимум в строке {5, 6, 7, 8} — **8**.
3. Максимум в строке {20, 510, 932, 392} — **932**.
4. Максимум в строке {100, 200, 150, 35} — **200**.
5. Наибольший элемент в матрице: 932

Ключевые моменты:

1. **Инициализация максимума:** В каждой строке поиск начинается с первого элемента.
2. **Два вложенных цикла:** Внешний цикл проходит по строкам, внутренний — по элементам текущей строки.
3. **Сохранение результатов:** Максимум каждой строки сохраняется в массив result.

Этот код эффективно находит максимальные значения по строкам двумерного массива и демонстрирует базовые принципы обработки массивов в Java.

Задание 3

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.atomic.AtomicInteger;

public class WarehouseTransfer {

    private static final int MAX_WEIGHT = 150;
    private static void moveGoods(String worker, int[] weights, int start,
int end, AtomicInteger totalWeight) {
        int currentWeight = 0;
        List<Integer> goods = new ArrayList<>();

        for (int i = start; i < end; i++) {
            if (currentWeight + weights[i] <= MAX_WEIGHT) {
                currentWeight += weights[i];
                goods.add(weights[i]);
                weights[i] = 0;
            }
            if (currentWeight == MAX_WEIGHT || i == end - 1) {
                System.out.println(worker + " перенёс товары весом: " + goods
+ " (Общий вес: " + currentWeight + " кг)");
                totalWeight.addAndGet(currentWeight);
                currentWeight = 0;
                goods.clear();
            }
        }
    }

    public static void main(String[] args) {
        int[] weights = {50, 20, 30, 70, 40, 60, 80, 10, 20, 90, 50, 40, 60,
30};
        List<CompletableFuture<Void>> tasks = new ArrayList<>();

        AtomicInteger totalWeight = new AtomicInteger(0);
        int numOfWorkers = 3;
        int partSize = (int) Math.ceil((double) weights.length /
numOfWorkers);

        for (int i = 0; i < numOfWorkers; i++) {
            final int workerId = i + 1;
            final int start = i * partSize;
            final int end = Math.min(start + partSize, weights.length);
            tasks.add(CompletableFuture.runAsync(() -> moveGoods("Грузчик " +
workerId, weights, start, end, totalWeight)));
        }
        CompletableFuture.allOf(tasks.toArray(new
CompletableFuture[0])).join();
        System.out.println("Все товары перенесены на новый склад!");
    }
}
```

Данный код представляет собой реализацию асинхронной многопоточной программы**,** в которой грузчики переносят товары с одного склада на другой. Здесь используется класс **CompletableFuture**, который позволяет асинхронно выполнять задачи.

Данный класс представляет собой способ управления результатом выполнения асинхронных операций.

---

1. Основные параметры программы

- **MAX_WEIGHT = 150**:
  Максимальный вес товаров, который грузчик может перенести за раз.
- **Массив weights**:
  Массив, хранящий веса всех товаров, которые нужно перенести:
- int[] weights = {50, 20, 30, 70, 40, 60, 80, 10, 20, 90, 50, 40, 60, 30};
- **Количество грузчиков** (numOfWorkers):
  Грузчики разделяют работу между собой:
- int numOfWorkers = 2;

---

2. Разделение задач между грузчиками

Блок кода:

```
int partSize = (int) Math.ceil((double) weights.length / numOfWorkers);
for (int i = 0; i < numOfWorkers; i++) {
    final int workerId = i + 1;
    final int start = i * partSize;
    final int end = Math.min(start + partSize, weights.length);

    tasks.add(CompletableFuture.runAsync(() ->  moveGoods("Грузчик " +
workerId, weights, start, end, totalWeight)));
}
```

1. **Размер подмассива** (partSize):
   Массив делится на примерно равные части для каждого грузчика с помощью:
2. partSize = (int) Math.ceil((double) weights.length / numOfWorkers);

   Например, если длина массива 14 и грузчиков 2, то partSize = 7.

3. **Определение диапазонов**:
   - Грузчик №1 работает с диапазоном 0...6.
   - Грузчик №2 работает с диапазоном 7...13.

4. **Создание асинхронных задач**:

   С помощью CompletableFuture.runAsync() каждому грузчику назначается отдельная задача:

tasks.add(CompletableFuture.runAsync(() -> moveGoods("Грузчик " + workerId, weights, start, end, totalWeight)));

---

3. Метод moveGoods

Метод moveGoods выполняет перенос товаров конкретным грузчиком.

Блок кода:

```
private static void moveGoods(String worker, int[] weights, int start, int
end, AtomicInteger totalWeight) {
    int currentWeight = 0;
    List<Integer> goods = new ArrayList<>();

    for (int i = start; i < end; i++) {
        if (currentWeight + weights[i] <= MAX_WEIGHT) {
            currentWeight += weights[i];
            goods.add(weights[i]);
            weights[i] = 0;
        }
        if (currentWeight == MAX_WEIGHT || i == end - 1) {
            System.out.println(worker + " перенёс товары весом: " + goods + "
(Общий вес: " + currentWeight + " кг)");
            totalWeight.addAndGet(currentWeight);
            currentWeight = 0;
            goods.clear();
        }
    }
}
```

Инициализация переменных:

- currentWeight: текущий суммарный вес товаров, которые переносит грузчик.
- goods: список товаров, которые входят в текущую переноску.

for (int i = start; i < end; i++): Этот цикл перебирает часть массива weights, начиная с индекса start и до end. Это говорит о том, что массив weights представляет веса доступных товаров.

if (currentWeight + weights[i] <= MAX_WEIGHT): это условие проверяет, не превышает ли добавление веса текущего предмета (weights[i]) к текущему общему весу (currentWeight) максимальный допустимый вес (MAX_WEIGHT).

currentWeight += weights[i];: Если условие истинно (предмет можно добавить), его вес добавляется к currentWeight.

goods.add(weights[i]);: Вес добавленного предмета добавляется в список goods.

weights[i] = 0;: Эта строка устанавливает вес выбранного предмета равным 0. Это способ отметить предмет как уже выбранный и избежать его повторного выбора в будущих итерациях.

Перебор товаров:

Грузчик обрабатывает товары в своём диапазоне (start...end):

```
if (currentWeight + weights[i] <= MAX_WEIGHT) {
currentWeight += weights[i];
goods.add(weights[i]);
 weights[i] = 0;
}
```

- Если добавление товара не превышает **MAX_WEIGHT**, грузчик его "забирает".
- Товар помечается как перенесённый, обнуляя его вес.

Отправка товаров:

Когда суммарный вес товаров достигает **MAX_WEIGHT** или это последний товар в диапазоне, грузчик переносит их:

i == end - 1:
Это условие означает:
"Грузчик дошёл до последнего элемента в своём диапазоне."

```
System.out.println(worker + " перенёс товары весом: " + goods + "
(Общий вес: " + currentWeight + " кг)");
  totalWeight.addAndGet(currentWeight);
```

Обнуление для следующей партии:

Вес и список goods сбрасываются.

---

4. Ожидание завершения всех задач

Блок кода:

```
CompletableFuture.allOf(tasks.toArray(new CompletableFuture[0])).join();
System.out.println("Все товары перенесены на новый склад!");
```

- **CompletableFuture.allOf**:
  Объединяет все асинхронные задачи и ожидает их завершения.
- **join()**:
  Блокирует выполнение программы, пока все задачи не будут завершены.

---

5. Вывод программы

- Каждый грузчик переносит товары по своему диапазону.
- Перенос осуществляется партиями с учётом ограничения MAX_WEIGHT.
- В конце выводится сообщение:
- Все товары перенесены на новый склад!

Итог

- Код использует CompletableFuture для запуска асинхронных задач.
- Задачи распределяются между грузчиками, каждый из которых обрабатывает свою часть товаров.
- Грузчики переносят товары партиями с ограничением по максимальному весу (MAX_WEIGHT).