

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»

(МТУСИ)

Кафедра «Математическая Кибернетика и Информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №7

по дисциплине

«Информационные технологии и программирование»

на тему

“Многопоточность”

Выполнил:

студент группы БВТ2302

Миронов А. А.

Москва, 2024 г.

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.

```
public class Halfsum{
    Run | Debug
    public static void main(String[] args) {
        int[] arr1 = new int[]{1,2,3,4,5,6,7,8};
        Thread thread1 = new FirstThread(arr1);
        Thread thread2 = new SecondThread(arr1);
        thread1.start();
        thread2.start();
        try{
            thread1.join();
            thread2.join();
        } catch(InterruptedException i){
            i.printStackTrace();
        }
        System.out.println("Sum of 2 threads is: " + FirstThread.sum+SecondThread.sum);
    }
}

static class FirstThread extends Thread{
    private int[] halfarr;
    private static int sum;

    public FirstThread(int[] myarr){
        halfarr = myarr;
    }
}
```

Можно увидеть, что в классе мэйн мы создаём два разных потока, которые являются экземплярами наших классов потоков созданных ниже по коду. Каждый из таких классов для потока, чтобы быть потоком и иметь возможность класть в ссылку класса Thread должен наследовать класс Thread, который в свою очередь реализует интерфейс runnable. Благодаря этому интерфейсу у нас в нашем классе-потока описанном ниже и присутствует метод run, этот метод ничего не возвращает и является обязательным для каждого потока. Его мы получаем из класс thread или интерфейса runnable и переопределяем, прописываю внутри то, что должен выполнять поток при запуске. С помощью команды start поток запускается и начинает своё выполнение, автоматически начинает выполняться блок run. Можно обратить внимания на то, что при создании потоков мы передаём наш массив в каждый из них, чтобы как раз подсчитать суммы его половины в первом и сумму другой половины во втором.

Заметим, что в конце мы складываем результаты, полученные после выполнения первого и второго потока (и записанные промежуточно в статические поля их классов) в нашем основном потоке, чтобы наш главный поток подождал, пока вызванный потоки произведут вычисления и только потом складывал значения классов их полей, мы на каждом из двух потоков применяем метод `join`, который приостанавливает выполнение потока, вызывающего поток с методом джойн до момента, пока этот поток не закончит своё выполнение. Также места, где джойн, мы обёртываем в блок трай кэтч и ловим исключение `interrupted exception`, которое выбрасывается в том случае, если поток, который ожидает был прерван.

```
@Override
public void run(){
    for (int i=0;i<halfarr.length/2;i++){
        sum+=halfarr[i];
    }
}

static class SecondThread extends Thread{
    private int[] halfarr;
    private static int sum;

    public SecondThread(int[] myarr){
        halfarr = myarr;
    }

    @Override
    public void run(){
        for (int i=halfarr.length-1;i>=halfarr.length/2;i--){
            sum+=halfarr[i];
        }
    }
}
```

Задание 2.

Реализация многопоточной программы для поиска наибольшего элемента в матрице.

Вариант 1. 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.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
public class ThreadMatrix {

    static List<Integer> listmax = new ArrayList<>();
    Run | Debug
    public static void main(String[] args) {
        int[][] mat1 = new int[][]{
            {1,2,18,4,2},
            {1,2,7,4,35},
            {1,21,3,14,7},
            {10,2,10,4,49}
        };

        List<Thread> threadlinks = new ArrayList<>();
        for (int i=0; i<mat1.length;i++){
            Thread thread1 = new ThreadMax(mat1[i]);
            threadlinks.add(thread1);
            thread1.start();
        }
    }
}
```

Тут у нас почти всё то же самое, что в предыдущем лишь с тем основным отличием, что количество создаваемых потоков у нас варьируется за счёт того, что мы создаём их с помощью цикла. Мы создаём столько потоков, сколько у нас строк в матрице и при создании каждого потока, находящего максимальный элемент в строке и добавляющего его в специальный список максимальных элементов, запускаем его. Также, чтобы сохранить ссылку на поток, пока мы имеем к ней доступ по переменной, мы кладём ссылку в специальный эррэйлист. Мы собрали ссылки на потоки, чтобы после выхода из цикла с помощью ещё одного цикла применить к каждому из выполняющихся потоков метод `join()`, чтобы главный поток подождал конца выполнения всех потоков и только после этого выводил максимальный элемент из эррэйлиста, куда потоки складывали максимальные значения из их строк.

```

        for (Thread thr_link:threadlinks){
            try{
                thr_link.join();
            } catch (InterruptedException h){
                h.printStackTrace();
            }
        }

        System.out.println("Maximal number of matrix: " + Collections.max(listmax));
    }
}

```

```

static class ThreadMax extends Thread{
    private int[] arr;

    public ThreadMax(int[] myarr){
        arr = myarr;
    }

    @Override
    public void run(){
        int maxarr=0;
        for (int i=0;i<arr.length;i++){
            if (arr[i]>=maxarr){
                maxarr=arr[i];
            }
        }
        listmax.add(maxarr);
    }
}
}

```


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 1.

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

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

public class Load{
    static StorageA p_StorageA = new Load().new StorageA(amount:5);
    static StorageB p_StorageB = new Load().new StorageB();
    Run | Debug
    public static void main(String[] args) {
        Thread loader1 = new Load().new Loader();
        Thread loader2 = new Load().new Loader();
        Thread loader3 = new Load().new Loader();
        while(p_StorageA.getTotal() != 0){
            loader1.start();
            loader2.start();
            loader3.start();
            try {
                loader1.join();
                loader2.join();
                loader3.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(p_StorageB.getList());
    }
}
```

Для начала создадим объекты далее созданных классов складаА (откуда грузчики будут переносить товар) и складаБ (куда грузчики будут переносить товар). Так как мы создаём объекты внутренних классов, то сначала нужно создать объект нашего внешнего класса Load, а только затем объект внутреннего класса склада уже от объекта внешнего склада. А также, не забываем, что к объектам классов складов (к их переменным) мы обращаемся из статического метода мэйна, статический метод может ссылаться только на статические переменные, т.к. статические переменные создаются во время создания класса, которому они принадлежат, а нестатические переменные создаются только при создании объекта, которому они принадлежат, и то же самое относится к статическим методам, то статический метод просто не может обратиться к переменной (нестатической), которая ещё не создана. Поэтому мы делаем переменные складов статическими. Также мы не случайно создаём объекты складов за пределами метода мэйна, так как если бы мы создали их внутри мэйна, они были бы локальными переменными класса и мы не могли бы обращаться к ним за пределами метода мэйна внутри класса Load, а нам в дальнейшем понадобятся подобные обращения. Теперь создадим три переменные в которых будут лежать объекты типа thread и положим в каждый из них новый объект потока, также с помощью сначала создания объекта load, а только потом создания от него его внутреннего класса loader, это и будут наши грузчики. После создаём цикл while, который выполняется до того момента, пока общий вес товаров складаА (gettotal) не будет равняться нулю, т.е. пока грузчики не перенесут все товары на складБ. Этот цикл запускает наши три потока, а затем мы обёртываем в try catch блок, где мы при выполнении каждого из этих потоков заставляем поток, вызвавший его, ждать конца выполнения вызванного потока, а в catch мы ловим исключение interruptedexception, которое возникает, когда поток, спящий или ждущий своей очереди прерывается, при ловле этого исключения мы выводим на экран его текст. И После окончания цикла мы выводим на экран результат в виде массива объектов типа товар (товаров перенесённых грузчиками в склад Б из склада А), который мы получаем из объекта класса StorageB посредством нестатического метода getlist (все методы и классы включая последний в методе мэйна, значение которых неясно, расписаны ниже).

```

class Merchandise{
    private int weight;

    public Merchandise(int wei){
        weight=wei;
    }

    public int getWei(){
        return weight;
    }
}

class StorageA{
    private List<Merchandise> merchStack = new ArrayList<>();
    private int total = 0;

    public int getTotal(){
        return total;
    }

    public int particWei(){
        if(!merchStack.isEmpty()){
            return merchStack.get(index:0).getWei();
        }
        else{
            return 0;
        }
    }
}

```

```

    public StorageA(int amount){
        for(int i=0; i<amount;i++){
            Merchandise tmerch = new Merchandise(wei:11);
            merchStack.add(tmerch);
            total+=tmerch.getWei();
        }
    }

    public Merchandise takethelast(){
        Merchandise merch = merchStack.get(merchStack.size()-1);
        total-=merch.getWei();
        merchStack.remove(merchStack.size()-1);
        return merch;
    }
}

```


Для начала опишем наш класс `Merchandise`, который будет классом нашего товара, наших грузов, которые грузчики-потоки будут переносить из склада А в склада Б. Естественно, основное поле товара, у нас приват и оно отвечает за вес товара. Также прописываем конструктор класса, в котором при создании объекта мы будем прописывать вес нашего товара, и вдобавок создадим метод, который будем использовать для получения веса конкретного товара, т е конкретного экземпляра класса `Merchandise` – `getWei`.

Теперь создадим наш класс склада А, в котором одна из переменных является списком, содержащим переменные типа `Merchandise` т е это и есть хранилище для товаров, а другая переменная отвечает за сумму веса товаров. Далее мы создали метод `particWei`, при применении которого к экземпляру класса нашего склада мы будем получать вес одного товара со склада. Т е если наше хранилище (`merchstack`) не пусто, то мы при применении этого метода получим вес товара, хранящегося в эррэйлисте по нулевому индексу, в обратном же случае метод вернёт 0. Ну и геттер для общего веса всех товаров, хранящихся на складе. Прописываем конструктор, в который передаём один параметр, значащий количество товаров, создаваемых при инициализации нового склада А. Конструктор будет с помощью цикла создавать столько раз новый товар с указанным весом, добавлять его в хранилище и прибавлять его вес к общему весу товаров на складе, какое число будет указано в конструкторе. И наконец прописываем последний метод `takethelast`, который будет позволять брать последний товар (самый верхний) из нашего склада. Мы сначала записываем как переменную наш последний товар из эррэйлиста, затем удаляем его из списка, минусуем из общей массы его вес и возвращаем этот объект-товар в конце метода.

```
class Loader extends Thread {  
    private List<Merchandise> merchload = new ArrayList<>();  
    private int Ltotal = 0;  
    final private int limit = 50;
```

```

@Override
public void run(){
    while (p_StorageA.merchStack.size()!=0 && Lttotal<=limit){
        Merchandise tempM = p_StorageA.takethelast();
        merchload.add(tempM);
        Lttotal+=tempM.getWei();
    }
    try{
        Thread.sleep(millis:1500);
    }catch(InterruptedException r){
        r.printStackTrace();
    }

    while(!merchload.isEmpty()){
        int lastel = merchload.size()-1;
        p_StorageB.putin(merchload.get(lastel));
        Lttotal-=merchload.get(lastel).getWei();
        merchload.remove(lastel);
    }
}
}

```

Опишем класс, который будет представлять наших грузчиков. Полями у них будет эррэйлист, (merchload) который символизирует мешок, куда они складывают свои товары и общий вес товаров. Также сделаем final переменную, которая будет содержать в себе предел грузоподъёмности одного грузчика-потока. Не забываем, что мы наследовали класс Thread, что и наделяет наш класс особыми функциями потока. Переопределим метод run. Первый цикл while будет отвечать за собирание грузчиком товаров со склада А в наш merchload, извлекая их методом takethelast, а также за прибавление их веса, и работать этот цикл будет до того момента пока merchstack склада А не станет пустым или до того момента, как добавление следующего товара не перегрузит нашего грузчика (т е общий вес не превысит 50 кг). Второе условие указано на скриншоте иначе — неверно, я изменил, чтобы грузчик не набирал больше лимита. Также я сделал методы takethelast, particWei и putin synchronized, этот маркер не даёт обращаться к методу больше, чем одному потоку одновременно (на скриншотах это также не выполнено, я сделал такие маркеры, потому что иначе программа работала непредсказуемо и не всегда все товары были

перенесены в другой склад, потому что возникала гонка потоков и соответственно ошибки, например, когда один поток проверял, что следующий товар не перегрузит его и получал удовлетворительный ответ, готовясь взять последний товар, а другой поток раньше делал всё это и использовал метод `takethelast` тоже раньше, забирая последний товар, и в таком случае наш первый поток пытался извлечь товар из пустого эррэйлиста по индексу `-1`, что приводило к выбросу исключения). Теперь симитируем путь, который преодолевает грузчик и усыпим текущий поток (т е тот, в котором вызываем метод `sleep` сейчас) на 1.5 секунды. После спячки мы вынимаем по индексу из `merchload` последний товар и кладём его с помощью метода класса склада `Б putin` в склад `Б` (его экземпляр), затем уменьшаем на вес этого товара общий вес мешка грузчика (`Ltotal`) и удаляем по индексу этот элемент из `merchload`. Вся эта конструкция обернута в цикл `while`, который работает до тех пор, пока `merchload` не станет пустым.

```
class StorageB{
    private ArrayList<Merchandise> merchStackB = new ArrayList<>();
    private int total = 0;

    public int getTotal(){
        return total;
    }

    public void putin(Merchandise merchy){
        merchStackB.add(merchy);
        total+=merchy.getWei();
    }

    public List<Merchandise> getList(){
        return new ArrayList<>(merchStackB);
    }
}
```

Теперь осталось описать лишь склад`Б`, который тоже в качестве своего места для хранения товаров будет иметь эррэйлист, а также сумму веса всех товаров, хранящихся на складе. Делаем геттер для `total`. Нестатический метод `putin`, в который мы передаём товар, а метод этот ничего не возвращает и добавляет этот наш товар в хранилище склада`Б` и увеличивает индикатор веса всех товаров склада`Б` на вес этого товара. Ну и инальный метод `getlist`, который используется

для получения эррэйлиста со всеми товарам хранищимися на складеБ на данный момент.

Вывод:

Мы научились работать с многопоточностью в джава, а в частности поняли, как работать с главным интерфейсом и классом для создания потоков, с методами этого класса, тобеж как запускать потоки, заставляя их ожидать завершения работы своих собратьев или приостанавливать свою работу на время. Мы узнали, в каких случаях поток может выбрасывать исключение при прерывании, а также изучили способ разрешения состояния гонки потоков. В общем мы пришли к заключению, что задействование многопоточности может значительно сокращать время работы программы и оптимизировать ресурсы компьютера.