Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

Комп‘ютерного практикуму № 3 з дисципліни
«Технології паралельних та розподілених обчислень»

«Розробка паралельних програм з використанням механізмів синхронізації: синхронізовані методи, локери, спеціальні типи»

Виконав(ла) *ІП-01 Корнієнко В.С.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив(ла) *Стеценко І. В.* _____
(прізвище, ім'я, по батькові)

Київ 2022

Завдання:

1. Реалізуйте програмний код, даний у лістингу, та протестуйте його при різних значеннях параметрів. Модифікуйте програму, використовуючи методи управління потоками, так, щоб її робота була завжди коректною. Запропонуйте три різних варіанти управління. **30 балів**.
2. Реалізуйте приклад Producer-Consumer application (див. https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html). Модифікуйте масив даних цієї програми, які читаються, у масив чисел заданого розміру (100, 1000 або 5000) та протестуйте програму. Зробіть висновок про правильність роботи програми. **20 балів**.
3. Реалізуйте роботу електронного журналу групи, в якому зберігаються оцінки з однієї дисципліни трьох груп студентів. Кожного тижня лектор і його 3 асистенти виставляють оцінки з дисципліни за 100-бальною шкалою. **40 балів**.
4. Зробіть висновки про використання методів управління потоками в java. **10 балів**.

Хід роботи:

Після реалізації коду з лістингу запускаємо програму і маємо наступні результати:

```
Transactions:43440000 Sum: -4592394
Transactions:43450000 Sum: -4593420
Transactions:43450000 Sum: -4593420
Transactions:43450000 Sum: -4593428
Transactions:43460000 Sum: -4594603
Transactions:43470001 Sum: -4595695
Transactions:43480000 Sum: -4597001
Transactions:43480003 Sum: -4597001
Transactions:43480003 Sum: -4597005
Transactions:43490001 Sum: -4598308
Transactions:43500002 Sum: -4599405
Transactions:43510002 Sum: -4600354
Transactions:43520000 Sum: -4601285
Transactions:43530002 Sum: -4602421
Transactions:43540003 Sum: -4603464
Transactions:43550001 Sum: -4604718
Transactions:43560000 Sum: -4605862
Transactions:43560000 Sum: -4605865
Transactions:43570000 Sum: -4606934
```

Можна побачити, що загальна сума постійно зменшується, що є некоректною роботою програми, оскільки сума має залишатися незмінною.

Першим способом вирішення нашої проблеми є використання ключового слова synchronized в сигнатурі методу transfer. Можемо представити оновлений метод наступним чином:

```
1 usage   valerii.korniienko
public synchronized void transferSyncMethod(int from, int to, int amount) {
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    if (ntransacts % NTEST == 0)
        test();
}
```

Тепер запустимо програму на виконання, але оновивши метод transfer:

```
Transactions:25130000 Sum: 100000
Transactions:25140000 Sum: 100000
Transactions:25150000 Sum: 100000
Transactions:25160000 Sum: 100000
Transactions:25170000 Sum: 100000
Transactions:25180000 Sum: 100000
Transactions:25190000 Sum: 100000
```

Бачимо що тепер сума незмінна навіть після великої кількості операцій. Можемо вважати таку поведінку коректною

Другий спосіб, завдяки якому можна досягти синхронізації операцій в цій задачі є використання блоку synchronized навколо частину, яку нам необхідно синхронізувати. Представимо оновлений метод transfer наступним чином:

```
1 usage   Valeni.Komilenko
public void transferSyncBlock(int from, int to, int amount) {
    synchronized (this) {
        accounts[from] -= amount;
        accounts[to] += amount;
        ntransacts++;
        if (ntransacts % NTEST == 0)
            test();
    }
}
```

Тепер запустимо програму на виконання з оновленим методом transfer. Маємо наступні результати:

```
Transactions:44080000 Sum: 100000
Transactions:44090000 Sum: 100000
Transactions:44100000 Sum: 100000
Transactions:44110000 Sum: 100000
Transactions:44120000 Sum: 100000
Transactions:44130000 Sum: 100000
Transactions:44140000 Sum: 100000
Transactions:44150000 Sum: 100000
Transactions:44160000 Sum: 100000
Transactions:44170000 Sum: 100000
```

Бачимо що з цим методом синхронізації програма теж працює коректно

Третім способом досягнення синхронізації даних є використання локерів, в даному випадку використаємо ReentrantLock. Оновлений метод буде виглядати наступним чином:

```
1 usage   valerii.korniienko
public void transferLock(int from, int to, int amount) {
    lock.lock();
    try {
        accounts[from] -= amount;
        accounts[to] += amount;
        ntransacts++;
        if (ntransacts % NTEST == 0)
            test();
    } finally {
        lock.unlock();
    }
}
```

Знову запустимо програму на виконання щоб переконатися в правильності роботи оновленого методу:

```
Transactions:132910000 Sum: 100000
Transactions:132920000 Sum: 100000
Transactions:132930000 Sum: 100000
Transactions:132940000 Sum: 100000
Transactions:132950000 Sum: 100000
Transactions:132960000 Sum: 100000
Transactions:132970000 Sum: 100000
Transactions:132980000 Sum: 100000
Transactions:132990000 Sum: 100000
Transactions:133000000 Sum: 100000
```

Програма працює правильно

2. В своїй реалізації задачі Producer-Consumer я використовував механізм wait-notify для синхронізації задач put та take в буфер таким чином, що операція put не може класти в переповнений буфер, а операція take не може брати з пустого. Для досягнення цього були створені 2 змінні:

```
3 usages
private boolean empty = true;
3 usages
private boolean full = false;
```

Тепер реалізуємо методи put і take в нашому буфері. Вони мають наступний вигляд:

```
1 usage  valerii.korniienko
public synchronized void put(int value) {
    while (full) {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }

    empty = false;
    buffer[++count] = value;
    full = count == buffer.length - 1;
    notifyAll();
}
```

```
1 usage  valerii.korniienko
public synchronized int take() {
    while (empty) {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }

    int value = buffer[count--];
    this.empty = count == 0;
    this.full = false;

    notifyAll();
    return value;
}
```

Як бачимо в методах put і take реалізовано механізм wait-notify в точності так як описувалося вище.

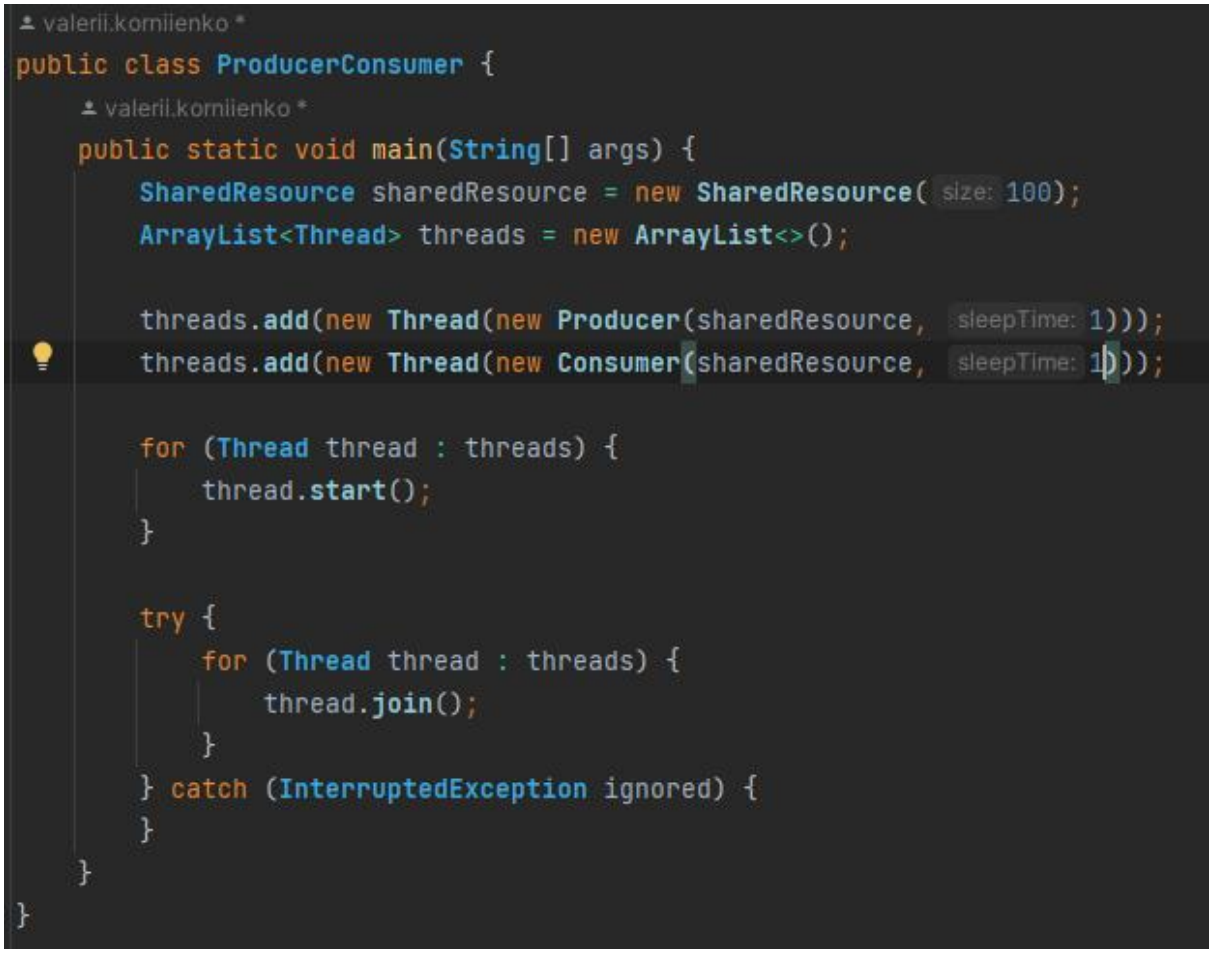
Тепер перевіримо правильність роботи програми. Для цього я створив 2 класи: Producer, Consumer, які імплементують інтерфейс Runnable та виконують функції кладіння та діставання з буферу відповідно.

Класи Producer, Consumer мають наступний вигляд:

```java
public class Producer implements Runnable {
    2 usages
    private final SharedResource sharedResource;
    2 usages
    private final long sleepTime;

    1 usage  valerii.korniienko
    public Producer(SharedResource sharedResource, long sleepTime) {...}

    valerii.korniienko
    public void run() {
        for (int i = 0; i < 10_000; i++) {
            sharedResource.put(i);
            System.out.println("Produced: " + i);
            try {
                Thread.sleep(sleepTime);
            } catch (InterruptedException ignored) {
            }
        }
    }
}
```

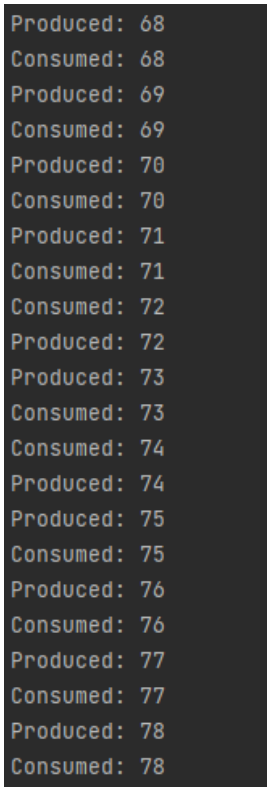
```java
public class Consumer implements Runnable {
    2 usages
    private final SharedResource sharedResource;
    2 usages
    private final long sleepTime;

    1 usage  valerii.korniienko
    public Consumer(SharedResource sharedResource, long sleepTime) {...}

    valerii.korniienko
    public void run() {
        for (int i = 0; i < 10_000; i++) {
            int value = sharedResource.take();
            System.out.println("Consumed: " + value);
            try {
                Thread.sleep(this.sleepTime);
            } catch (InterruptedException ignored) {
            }
        }
    }
}
```

Для перевірки правильності роботи програми створимо буфер розміром 100 та запустимо програму на виконання:



```
valerii.korniienko *
public class ProducerConsumer {
    valerii.korniienko *
    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource( size: 100);
        ArrayList<Thread> threads = new ArrayList<>();

        threads.add(new Thread(new Producer(sharedResource,  sleepTime: 1)));
        threads.add(new Thread(new Consumer(sharedResource,  sleepTime: 1)));

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

        try {
            for (Thread thread : threads) {
                thread.join();
            }
        } catch (InterruptedException ignored) {
        }
    }
}
```

При запуску роботи програми маємо такі результати:



Даний результат є правильним

3. Для виконання 3 завдання я створив наступні класи:
4.
    - Клас Student (містить інформацію про студента, його унікальний код)
    - Клас Group (містить в собі список студентів які належать групі, назву групи)
    - Клас Journal (містить список груп, список оцінок кожного окремого студента, метод для їх виставляння, тестування)
    - Клас Teacher (implements Runnable, виставляє оцінки групам)

Програма працює наступним чином: кожен тиждень вчителі виставляють оцінку в спільний журнал і в кінці кожного тижня ми виводимо вміст журналу для доведення правильності роботи програми.

Для додавання оцінок використовується клас Teacher, що виглядає наступним чином:

```
public class Teacher implements Runnable {
    2 usages
    private final Journal journal;
    2 usages
    private final ArrayList<Group> availableGroups;

    4 usages  valerii.korniienko
    public Teacher(Journal journal, ArrayList<Group> availableGroups) {...}

    valerii.korniienko
    public void run() {
        Random random = new Random();
        for (var group : availableGroups) {
            for (var student : group.getStudents()) {
                    journal.addMark(student.getId(),  mark: random.nextInt( bound: 41) + 60);
            }
        }
    }
}
```

Як можемо зрозуміти, всі вчителі виставляють оцінки в журнал паралельно, тож операцію додавання оцінки треба синхронізувати. Для цього використаємо на методі addMark ключове слово synchronized. Тепер метод addMark виглядає наступним чином:

```java
public synchronized void addMark(UUID studentId, int mark) {
    if (marks.containsKey(studentId)) {
        marks.get(studentId).add(mark);
    } else {
        ArrayList<Integer> marksList = new ArrayList<>();
        marksList.add(mark);
        marks.put(studentId, marksList);
    }
}
```

Тепер перевіримо правильність роботи програми. Для цього запустимо програму на 3 тижні. Кожного тижня кожному студенту мають виставляти по 3 оцінки.

Після запуску програми маємо наступні результати:

```
----------------
Group 1
Name: Name0  Marks: [77, 61, 65, 99, 62, 86, 95, 72, 94]  Marks count: 9
Name: Name1  Marks: [100, 98, 71, 68, 81, 70, 82, 67, 70]  Marks count: 9
Name: Name2  Marks: [87, 74, 69, 87, 83, 84, 66, 95, 96]  Marks count: 9
----------------
Group 2
Name: Name0  Marks: [92, 72, 64, 75, 65, 97, 72, 76, 75]  Marks count: 9
Name: Name1  Marks: [70, 93, 74, 93, 90, 90, 78, 77, 69]  Marks count: 9
Name: Name2  Marks: [98, 70, 90, 73, 100, 81, 73, 68, 72]  Marks count: 9
Name: Name3  Marks: [80, 95, 83, 67, 88, 73, 99, 97, 79]  Marks count: 9
----------------
Group 3
Name: Name0  Marks: [96, 93, 87, 89, 98, 98, 67, 63, 61]  Marks count: 9
Name: Name1  Marks: [92, 67, 74, 63, 93, 78, 100, 93, 61]  Marks count: 9
Name: Name2  Marks: [66, 69, 93, 80, 71, 98, 88, 68, 92]  Marks count: 9
```

Як бачимо, кожен студент має по 9 оцінок, можемо дійти висновку що програма працює коректно

Висновки

У Java існує декілька методів для забезпечення синхронізації потоків. Наприклад, можна використовувати ключове слово `synchronized`, синхронізовані блоки, локери або синхронізовані типи даних.

1. synchronized: Ключове слово synchronized використовується для забезпечення синхронізації доступу до спільних ресурсів. Це означає, що тільки один потік може отримати доступ до блоку коду, огорнутого в synchronized. Це допомагає уникнути проблем, таких як гонка за ресурсами (race condition), коли кілька потоків намагаються одночасно змінити спільний ресурс.

2. synchronized(obj) та locker використовується для синхронізації доступу до конкретного об'єкта. Це дозволяє забезпечити синхронізований доступ до об'єкта лише одним потоком одночасно.

3. Варто зазначити що locker дає більш гнучкий та високорівневий інтерфейс для блокування об’єкта(наприклад Conditions, метод tryLock і тд)

Також в даній роботі був використаний механізм wait-notify. Метод wait() використовується для призупинки потоку і звільнення блокувального об'єкта, щоб інші потоки могли виконати свої завдання. Метод notify() використовується для повідомлення призупиненому потоку, що він може продовжити виконання. Цей механізм дозволяє потокам взаємодіяти та координувати свою роботу.

Використання цих методів допомагає забезпечити безпечне та ефективне паралельне виконання коду в Java, зменшує ймовірність проблем з одночасним доступом до спільних ресурсів і дозволяє потокам спілкуватися та синхронізувати свою роботу.

Код програми:

Завдання 1:

```
import java.util.concurrent.locks.ReentrantLock;

public class AsynchBankTest {
    public static final int NACCOUNTS = 10;
    public static final int INITIAL_BALANCE = 10000;
    public static void main(String[] args) {
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
        int i;
        for (i = 0; i < NACCOUNTS; i++){
            TransferThread t = new TransferThread(b, i,
                    INITIAL_BALANCE);
            t.setPriority(Thread.NORM_PRIORITY + i % 2);
            t.start () ;
        }
    }
}
class Bank {
    public static final int NTEST = 10000;
    private final int[] accounts;
    private long ntransacts = 0;
    private final ReentrantLock lock;
    public Bank(int n, int initialBalance){
        accounts = new int[n];
        int i;
        for (i = 0; i < accounts.length; i++)
            accounts[i] = initialBalance;
        ntransacts = 0;
        lock = new ReentrantLock();
    }

    public void transfer(int from, int to, int amount) {
        accounts[from] -= amount;
        accounts[to] += amount;
        ntransacts++;
        if (ntransacts % NTEST == 0)
            test();
    }

    public synchronized void transferSyncMethod(int from, int
to, int amount) {
        accounts[from] -= amount;
        accounts[to] += amount;
        ntransacts++;
        if (ntransacts % NTEST == 0)
            test();
    }
    public void transferSyncBlock(int from, int to, int
```

```
amount) {
        synchronized (this) {
            accounts[from] -= amount;
            accounts[to] += amount;
            ntransacts++;
            if (ntransacts % NTEST == 0)
                test();
        }
    }
    public void transferLock(int from, int to, int amount) {
        lock.lock();
        try {
            accounts[from] -= amount;
            accounts[to] += amount;
            ntransacts++;
            if (ntransacts % NTEST == 0)
                test();
        } finally {
            lock.unlock();
        }
    }

    public void test(){
        int sum = 0;
        for (int i = 0; i < accounts.length; i++)
            sum += accounts[i] ;
        System.out.println("Transactions:" + ntransacts
                + " Sum: " + sum);
    }
    public int size(){
        return accounts.length;
    }
}
class TransferThread extends Thread {
    private Bank bank;
    private int fromAccount;
    private int maxAmount;
    private static final int REPS = 1000;
    public TransferThread(Bank b, int from, int max){
        bank = b;
        fromAccount = from;
        maxAmount = max;
    }
    @Override
    public void run(){
        while (true) {
            for (int i = 0; i < REPS; i++) {
                int toAccount = (int) (bank.size() *
Math.random());
                int amount = (int) (maxAmount *
Math.random()/REPS);
```

```
//                  bank.transfer(fromAccount, toAccount,
amount);
//                  bank.transferSyncMethod(fromAccount,
toAccount, amount);
//                  bank.transferSyncBlock(fromAccount,
toAccount, amount);
                bank.transferLock(fromAccount, toAccount,
amount);
            }
        }
    }
}
```

Завдання 2:

```
public class Consumer implements Runnable {
    private final SharedResource sharedResource;
    private final long sleepTime;

    public Consumer(SharedResource sharedResource, long
sleepTime) {
        this.sharedResource = sharedResource;
        this.sleepTime = sleepTime;
    }

    public void run() {
        for (int i = 0; i < 10_000; i++) {
            int value = sharedResource.take();
            System.out.println("Consumed: " + value);
            try {
                Thread.sleep(this.sleepTime);
            } catch (InterruptedException ignored) {
            }
        }
    }
}

public class Producer implements Runnable {
    private final SharedResource sharedResource;
    private final long sleepTime;

    public Producer(SharedResource sharedResource, long
sleepTime) {
        this.sharedResource = sharedResource;
        this.sleepTime = sleepTime;
    }

    public void run() {
        for (int i = 0; i < 10_000; i++) {
            sharedResource.put(i);
```

```
            System.out.println("Produced: " + i);
            try {
                Thread.sleep(sleepTime);
            } catch (InterruptedException ignored) {
            }
        }
    }
}


public class SharedResource {
    private boolean empty = true;
    private boolean full = false;
    private int[] buffer;
    private int count = 0;
    public SharedResource(int size){
        buffer = new int[size];
    }

    public synchronized int take() {
        while (empty) {
            try {
                wait();
            } catch (InterruptedException ignored) {
            }
        }

        int value = buffer[count--];
        this.empty = count == 0;
        this.full = false;

        notifyAll();
        return value;
    }

    public synchronized void put(int value) {
        while (full) {
            try {
                wait();
            } catch (InterruptedException ignored) {
            }
        }

        empty = false;
        buffer[++count] = value;
        full = count == buffer.length - 1;
        notifyAll();
    }
}
```

```java
import java.util.ArrayList;

public class ProducerConsumer {
    public static void main(String[] args) {
        SharedResource sharedResource = new
SharedResource(100);
        ArrayList<Thread> threads = new ArrayList<>();

        threads.add(new Thread(new Producer(sharedResource,
1)));
        threads.add(new Thread(new Consumer(sharedResource,
1)));

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

        try {
            for (Thread thread : threads) {
                thread.join();
            }
        } catch (InterruptedException ignored) {
        }
    }
}
```

Завдання 3:

```java
import java.util.UUID;

public class Student {
    private final String name;
    private final String surname;
    private final UUID id;

    public Student(String name, String surname) {
        this.name = name;
        this.surname = surname;
        this.id = UUID.randomUUID();
    }

    public UUID getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

```java
import java.util.ArrayList;

public class Group {
    private ArrayList<Student> students = new ArrayList<>();
    private String name;

    public Group(ArrayList<Student> students, String name) {
        this.students = students;
        this.name = name;
    }

    public Group(String name) {
        this.name = name;
    }

    public void addStudent(Student student) {
        students.add(student);
    }

    public void removeStudent(Student student) {
        students.remove(student);
    }

    public ArrayList<Student> getStudents() {
        return students;
    }

    public String getName() {
        return name;
    }
    public void randomFill(int count) {
        for (int i = 0; i < count; i++) {
            students.add(new Student("Name" + i, "Surname" +
i));
        }
    }
}

import java.util.ArrayList;
import java.util.HashMap;
import java.util.UUID;

public class Journal {
    private ArrayList<Group> groups = new ArrayList<>();
    private final HashMap<UUID, ArrayList<Integer>> marks =
new HashMap<>();
    private String name;

    public Journal(ArrayList<Group> groups, String name) {
        this.groups = groups;
        this.name = name;
```

```
    }

    public Journal(String name) {
        this.name = name;
    }

    public void addGroup(Group group) {
        groups.add(group);
    }

    public void removeGroup(Group group) {
        groups.remove(group);
    }

    public ArrayList<Group> getGroups() {
        return groups;
    }

    public String getName() {
        return name;
    }

    public synchronized void addMark(UUID studentId, int mark)
{
        if (marks.containsKey(studentId)) {
            marks.get(studentId).add(mark);
        } else {
            ArrayList<Integer> marksList = new ArrayList<>();
            marksList.add(mark);
            marks.put(studentId, marksList);
        }
    }

    public void test() {
        for (var group : groups) {
            System.out.println("-----------------");
            System.out.println(group.getName());
            for (var student : group.getStudents()) {
                System.out.print("Name: " +
student.getName());
                System.out.print("  Marks: " +
marks.get(student.getId()));
                System.out.println("  Marks count:
"+marks.get(student.getId()).size());
            }
        }
    }
}
```

```java
import java.util.ArrayList;
import java.util.Random;

public class Teacher implements Runnable {
    private final Journal journal;
    private final ArrayList<Group> availableGroups;

    public Teacher(Journal journal, ArrayList<Group>
availableGroups) {
        this.journal = journal;
        this.availableGroups = availableGroups;
    }

    public void run() {
        Random random = new Random();
        for (var group : availableGroups) {
            for (var student : group.getStudents()) {
                    journal.addMark(student.getId(),
random.nextInt(41) + 60);
            }
        }
    }
}
```

```java
import java.util.ArrayList;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) throws
InterruptedException {
        Group group = new Group("Group 1");
        Group group2 = new Group("Group 2");
        Group group3 = new Group("Group 3");
        group.randomFill(3);
        group2.randomFill(4);
        group3.randomFill(3);
        Journal journal = new Journal(new
ArrayList<>(Arrays.asList(group, group2, group3)), "Journal");

        var lecturer = new Teacher(journal, new
ArrayList<>(Arrays.asList(group, group2, group3)));
        var teacher1 = new Teacher(journal, new
ArrayList<>(Arrays.asList(group, group2)));
        var teacher2 = new Teacher(journal, new
ArrayList<>(Arrays.asList(group2, group3)));
        var teacher3 = new Teacher(journal, new
ArrayList<>(Arrays.asList(group, group3)));

        int weeksCount = 3;
        ArrayList<Thread> threads = new ArrayList<>();
```

```
        for (int i = 0; i < weeksCount; i++) {
            threads.add(new Thread(lecturer));
            threads.add(new Thread(teacher1));
            threads.add(new Thread(teacher2));
            threads.add(new Thread(teacher3));

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

            try {
                for (Thread thread : threads) {
                    thread.join();
                }
            } catch (InterruptedException ignored) {
            }

            System.out.println("Week " + (i + 1));
            journal.test();
            Thread.sleep(100);
            threads.clear();
        }
    }
}
```