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

з лабораторної роботи №3 з дисципліни
«Технології паралельних обчислень»

Виконав:
Гончаренко Дмитро ІТ-01

Захищено з оцінкою _____
Дата захисту _____

Перевірила: проф.
Стеценко І. В.

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

1. Початковий лістинг представляє простий банківський сценарій з використанням багатопоточності. Основна мета програми - виконання переказу коштів між рахунками у банку. Програма створює NACCOUNTS рахунків з початковим балансом INITIAL_BALANCE. Клас Bank містить метод transfer(), який відповідає за переказ коштів між рахунками. Кожен раз, коли виконується переказ, збільшується лічильник ntransacts, який відстежує кількість здійснених транзакцій. При досягненні кратного значення NTEST, викликається метод test(), який виводить повідомлення про кількість транзакцій та суму на рахунках. Клас TransferThread є потоком, який виконує випадкові перекази між рахунками. У циклі run() виконується REPS ітерацій, під час кожної ітерації генеруються випадкові номери рахунків та сума переказу, і викликається метод transfer() класу Bank для здійснення переказу коштів.

Реалізація програмного коду з лістингу

```
Transactions:80043 Sum: 98626
Transactions:70006 Sum: 98745
Transactions:60004 Sum: 98828
Transactions:20004 Sum: 99433
Transactions:80003 Sum: 98626
Transactions:90000 Sum: 98626
Transactions:40006 Sum: 98943
Transactions:100011 Sum: 98345
Transactions:110007 Sum: 98025
Transactions:120006 Sum: 97727
Transactions:130009 Sum: 97226
Transactions:140002 Sum: 96960
Transactions:30081 Sum: 99267
Transactions:150008 Sum: 96580
Transactions:160012 Sum: 96226
Transactions:50008 Sum: 98853
```

Однак, у початковому лістингу відсутні механізми синхронізації, тому програма працює некоректно в умовах багатопоточності. Виникають проблеми зі зміною балансів на рахунках, дублювання транзакцій та неочікуване поведінка програми.

Підсумовуючи, ось проблеми, які треба вирішити:

- Транзакції відбуваються в неправильному порядку та не на однакових проміжках
- Загальна сума поступово зменшується
- Нескінчений цикл ітерацій

Почнемо з останії(нескінчений цикл ітерацій). Проблема з нескінченим циклом ітерацій виникає через відсутність умови зупинки циклу в потоці TransferThread. Щоб вирішити цю проблему, я додав умову зупинки циклу в потоці TransferThread. Ця умова пов'язана з досягненням ліміту кількості транзакцій або зі станом переривання потоку.

Оновлений код

```java
private int transactionLimit;

1 usage
public TransferThread(Bank b, int from, int max, int limit) {
    bank = b;
    fromAccount = from;
    maxAmount = max;
    transactionLimit = limit;
}

@Override
public void run() {
    int transactionCount = 0;
    while (transactionCount < transactionLimit && !isInterrupted()) {
        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);
            transactionCount++;

            if (transactionCount >= transactionLimit) {
                break;
            }
```

У цьому оновленому коді я додав змінну transactionLimit, яка встановлює ліміт кількості транзакцій для кожного потоку. В циклі run() перевіряється, чи не досягнуто ліміту кількості транзакцій або чи не був потік перерваний. Якщо будь-яка з цих умов виконується, цикл завершується, а потік зупиняється.
За допомогою цих змін, цикл в потоці TransferThread буде зупинятися після досягнення ліміту кількості транзакцій або в разі переривання потоку. Це допоможе уникнути нескінченого виконання транзакцій та забезпечити правильну роботу програми.

Результат

```
Transactions:30010 Sum: 99827
Transactions:40006 Sum: 99713
Transactions:50002 Sum: 99698
Transactions:20012 Sum: 99962
Transactions:61016 Sum: 99659
Transactions:10026 Sum: 99987
Transactions:70090 Sum: 99379
Transactions:80010 Sum: 99115
Transactions:90007 Sum: 98578
```

Далі потрібно вирішити дві інші проблеми.

1 спосіб - використання ключового слова synchronized

Ключове слово synchronized в Java використовується для створення синхронізованих блоків коду або синхронізованих методів.
Коли метод оголошується як synchronized, тільки один потік може виконувати цей метод в будь-який момент часу. Інші потоки, які намагаються виконати цей метод, повинні зачекати, поки перший потік завершить виконання. Цей підхід дозволяє забезпечити взаємовиключення потоків та правильний порядок виконання коду. Коли потік виконує синхронізований метод, інші потоки мають зачекати, поки виконання поточного методу не завершиться.
У випадку, коли кілька потоків намагаються виконати різні синхронізовані методи на одному об'єкті, вони також будуть блоковані один від одного. Це допомагає уникнути проблем конкурентного доступу до спільних ресурсів та забезпечує коректну роботу програми у багатопотоковому середовищі.

Використання ключового слова synchronized для методів transfer() та test() допоможе забезпечити взаємовиключення потоків та правильний порядок виконання.

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

1 usage
public synchronized void test() {
    int sum = 0;
    for (int i = 0; i < accounts.length; i++)
        sum += accounts[i];
    System.out.println("Transactions: " + ntransacts + " Sum: " + sum);
}
```

Результат

```
Transactions: 10000 Sum: 100000
Transactions: 20000 Sum: 100000
Transactions: 30000 Sum: 100000
Transactions: 40000 Sum: 100000
Transactions: 50000 Sum: 100000
Transactions: 60000 Sum: 100000
Transactions: 70000 Sum: 100000
Transactions: 80000 Sum: 100000
Transactions: 90000 Sum: 100000
Transactions: 100000 Sum: 100000
```

2 спосіб – Lock/ReentrantLock

Використовується пакет java.util.concurrent.locks, який надає можливості для синхронізації доступу до ресурсів між потоками. Зокрема, використовується клас ReentrantLock, який є реалізацією інтерфейсу Lock. Lock використовується для створення блокувань, щоб гарантувати, що тільки один потік може отримати доступ до коду, захищеного блокуванням, в один момент часу. Це дозволяє забезпечити правильну синхронізацію між потоками та запобігти race conditions

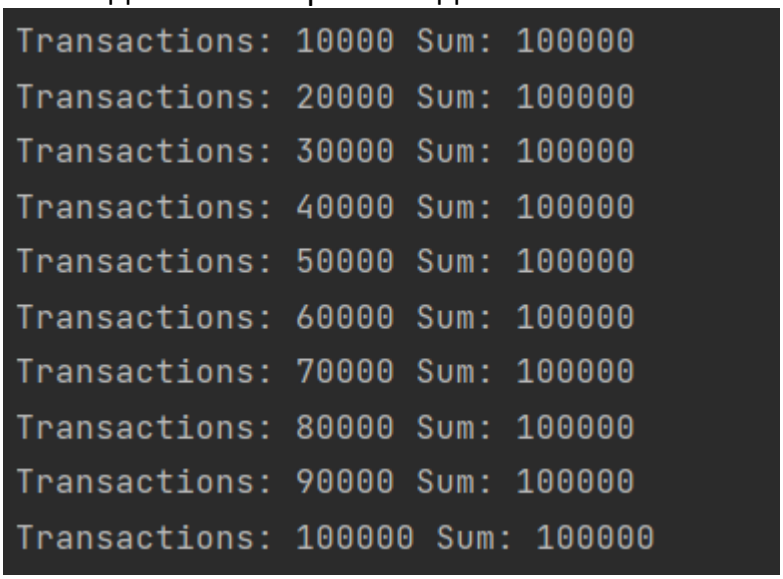
і помилкам у багатопотоковому середовищі. Блокування дозволяє потокам отримувати доступ до критичної секції коду послідовно, уникнувши некоректних конкурентних ситуацій та забезпечивши безпеку та правильність виконання операцій.

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

Також в main треба додати метод join(). Метод join() блокує виконання головного потоку до тих пір, поки кожен потік зі списку threads не завершить своє виконання. Це забезпечує, що всі потоки будуть виконані перед продовженням виконання головного потоку.

```java
public static void main(String[] args) throws InterruptedException {
    Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
    int i;
    int transactionLimit = 10000;
    List<TransferThread> threads = new ArrayList<>();

    for (i = 0; i < NACCOUNTS; i++) {
        TransferThread t = new TransferThread(b, i, INITIAL_BALANCE, transactionLimit);

        t.setPriority(Thread.NORM_PRIORITY + i % 2);
        t.start();
        threads.add(t);
    }

    for (TransferThread t : threads) {
        t.join();
    }
```

Таким чином, використання методу join() дозволяє дочекатись завершення всіх потоків перед тим, як програма завершиться повністю. Це важливо, щоб переконатись, що всі транзакції були оброблені перед виведенням результатів або подальшою обробкою даних.




3 спосіб – синхронізований блок

Різниця між синхронізованим методом і синхронізованим блоком в тому, яка частина коду забезпечується синхронізацією.
Синхронізований метод: У цьому випадку весь метод помічений ключовим словом synchronized. Це означає, що доступ до цього методу буде синхронізованим для всіх потоків, які викликають цей метод. Коли один потік викликає синхронізований метод, інші потоки повинні очікувати, доки перший потік не завершить виконання методу. Це забезпечує синхронізований доступ до ресурсів, які контролюються методом.
Синхронізований блок: У цьому випадку ви використовуєте ключове слово synchronized разом з об'єктом або ресурсом, який ви хочете синхронізувати. Всередині синхронізованого блоку ви вказуєте об'єкт або ресурс, який буде використовуватись для блокування доступу до коду всередині блоку. Тільки один потік може виконувати код всередині синхронізованого блоку одночасно, інші потоки повинні чекати, доки перший потік не вийде з блоку.
Отже, основна різниця полягає в тому, яка частина коду забезпечується синхронізацією. Синхронізований метод синхронізує весь метод, тоді як синхронізований блок дозволяє синхронізувати лише певну частину коду. Вибір між цими двома підходами залежить від потреб вашої програми та специфічності ситуації, яку ви намагаєтесь вирішити.

Оскільки метод transfer маніпулює даними банку (зменшує баланс одного рахунку і збільшує баланс іншого), важливо забезпечити атомарність операцій і унікальний доступ до ресурсів для кожного потоку, який викликає цей метод.

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

Результат

```
Transactions:10000 Sum: 100000
Transactions:20000 Sum: 100000
Transactions:30000 Sum: 100000
Transactions:40000 Sum: 100000
Transactions:50000 Sum: 100000
Transactions:60000 Sum: 100000
Transactions:70000 Sum: 100000
Transactions:80000 Sum: 100000
Transactions:90000 Sum: 100000
Transactions:100000 Sum: 100000
```

**Завдання 2.**

Програма Producer-Consumer є одним з найпоширеніших способів координації роботи потоків. В ній є два типи потоків: виробник (Producer) і споживач

(Consumer). Виробник виробляє дані (повідомлення), а споживач споживає ці дані. Обмін даними між потоками відбувається через спільний об'єкт (Drop).
Основна ідея полягає в тому, що виробник не може виробляти нові дані, якщо старі дані не були спожиті споживачем, і навпаки, споживач не може споживати нові дані, якщо виробник не виробив їх. Ця синхронізація забезпечується за допомогою використання "охоронних блоків" (guarded blocks) і методів wait() та notify(). В даному прикладі Producer-Consumer використовуються методи wait() та notify() для реалізації guarded blocks і забезпечення координації між потоками. Потоки використовують ці методи для чекання на вільний буфер або наявність даних у буфері.
В методі take() класу Drop потік-споживач викликає wait(), якщо буфер порожній, і чекає, доки потік-виробник не помістить дані в буфер і викличе notifyAll(), щоб прокинути усі чекаючі потоки-споживачі.
Аналогічно, в методі put() класу Drop потік-виробник викликає wait(), якщо буфер повний, і чекає, доки потік-споживач не забере дані з буфера і викличе notifyAll(), щоб прокинути усі чекаючі потоки-виробники.
Таким чином, за допомогою методів wait(), notify(), і notifyAll(), реалізовано механізм guarded blocks, який дозволяє потокам ефективно координувати свою роботу і чекати на відповідні умови без зайвого споживання процесорного часу.

Клас Drop - спільний об'єкт, що забезпечує комунікацію між класами Producer і Consumer.
Клас Consumer - отримує повідомлення з об'єкта Drop і виводить їх на екран.
Клас Producer - створює повідомлення і передає їх у об'єкт Drop для споживання.

Спочатку реалізуємо початковий приклад з лістингу

```
MESSAGE RECEIVED: Mares eat oats
MESSAGE RECEIVED: Does eat oats
MESSAGE RECEIVED: Little lambs eat ivy
MESSAGE RECEIVED: A kid will eat ivy too
```

Для модифікації початкового варіанту коду були внесені наступні зміни:

В класі Consumer було додано поле int messageCount, яке використовується для відстеження порядкового номера повідомлення.
В циклі while у методі run() класу Consumer, замість порівняння зі стрічкою "DONE", використовується порівняння зі значенням -1. Тобто, цикл

продовжується доки не буде отримано значення -1, яке вказує на завершення виробництва.
В класі Producer було додано поля int iterations і int maxNumber, які визначають кількість ітерацій виробництва і максимальне значення, яке може бути згенеровано.
В циклі for у методі run() класу Producer, замість статичного масиву String importantInfo[], використовується генерація випадкового числа методом random.nextInt(maxNumber).
В класі Drop змінено тип поля private String message на private int number, оскільки тепер використовується числове значення повідомлення.
В класі Drop методи take() і put() було змінено так, щоб працювати з числовим значенням.
У класі ProducerConsumerExample створено об'єкти Drop, Producer і Consumer з відповідними аргументами. Додано два окремих потоки для виробника і споживача, і за допомогою методу join() головний потік очікує їх завершення.
Після внесення змін, програма буде працювати наступним чином:

Виробник генерує випадкові числа в межах від 0 до maxNumber і передає їх споживачу через об'єкт Drop.
Споживач отримує ці числа і виводить на консоль у форматі "Received: Message#num", де num - порядковий номер повідомлення.
Процес повторюється iterations разів.
Після завершення виробництва, споживач отримує спеціальне значення -1, яке вказує на кінець виробництва.
В кінці програми потоки виробника і споживача об'єднуються з головним потоком за допомогою методу join(), щоб дочекатися їх завершення перед закінченням програми.

Результат

```
Received: Message#1 - 88
Received: Message#2 - 834
Received: Message#3 - 506
Received: Message#4 - 75
Received: Message#5 - 976
Received: Message#6 - 691
Received: Message#7 - 694
Received: Message#8 - 641
Received: Message#9 - 518
Received: Message#10 - 511
Received: Message#11 - 401
Received: Message#12 - 150
Received: Message#13 - 838
```

**Завдання 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.

Нехай кожен робочий день(понеділок-п’ятниця) лектор і його 3 асистенти виставляють кожному студенту по одній оцінці. Було створено такі класи:

- Teacher (Вчитель): Цей клас реалізує інтерфейс Runnable і представляє вчителя (лектора або асистента). Кожен вчитель відповідає за виставлення оцінок студентам. Він отримує список студентів, журнал оцінок та своє ім'я (наприклад, "Lecturer" або "Assistant 1"). У методі run() вчитель ітерується по списку студентів та виставляє кожному студенту оцінку за кожний день (понеділок-п'ятниця).

- Student (Студент): Цей клас представляє студента. Він має поле для зберігання імені студента. Клас Student використовується для створення об'єктів студентів.

- GradingJournal (Журнал оцінок): Цей клас відповідає за збереження оцінок студентів. Він має мапу, де ключами є студенти, а значеннями є список оцінок для кожного студента. Клас GradingJournal також має синхронізований метод addGrade(), який виставляє оцінку конкретному студенту і додає її до списку оцінок студента. Це забезпечує безпечний доступ до журналу оцінок з боку різних вчителів, які виконують свої дії паралельно.

- Main (Головний клас): Цей клас містить метод main(). В ньому створюються об'єкти студентів, журнал оцінок та вчителів. Також створюються потоки для виконання вчителів. Головний клас запускає потоки вчителів, дочекається їх виконання та виводить всі оцінки для кожного студента.

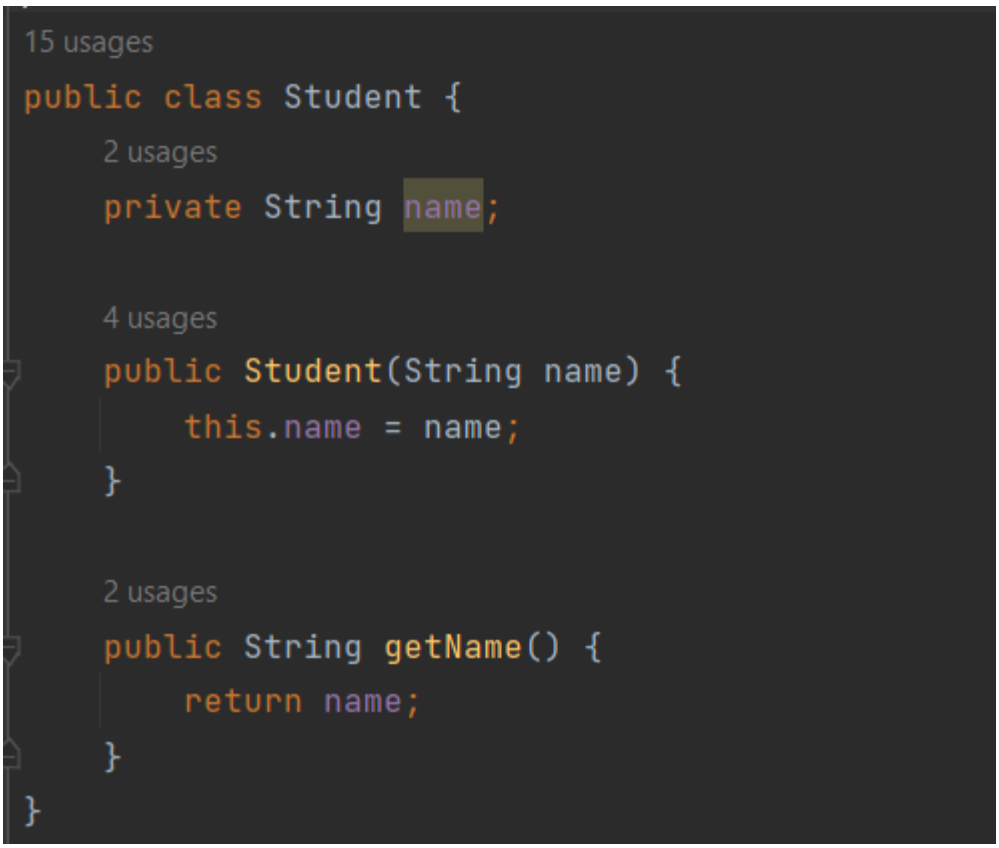
Клас GradingJournal

```
class GradingJournal {
    4 usages
    private Map<Student, Map<Integer, List<Integer>>> grades;
    3 usages
    private Lock lock;

    1 usage
    public GradingJournal(List<Student> students) {
        this.grades = new HashMap<>();
        for (Student student : students) {
            grades.put(student, new HashMap<>());
        }
        this.lock = new ReentrantLock();
    }

    1 usage
    public void addGrade(Student student, int grade, String teacherName, int week) {
        lock.lock();
        try {
            Map<Integer, List<Integer>> studentGrades = grades.get(student);
            if (!studentGrades.containsKey(week)) {
                studentGrades.put(week, new ArrayList<>());
            }
            List<Integer> weekGrades = studentGrades.get(week);
            weekGrades.add(grade);
            System.out.println("Grade added for student " + student.getName() + " by " + teacherName + ": " + grade + " (Week " +
        } finally {
            lock.unlock();
        }
    }
```

Конструктор GradingJournal приймає список студентів і ініціалізує поле grades, створюючи порожні списки оцінок для кожного студента. Крім того, створюється об'єкт ReentrantLock для lock. Метод addGrade додає оцінку для певного студента, вказуючи вчителя, який її виставив, та номер тижня. Метод

використовує ReentrantLock для забезпечення безпечної модифікації оцінок. Спочатку метод перевіряє, чи існує мапування оцінок для даного тижня, якщо ні, то створює його. Потім додає оцінку до списку оцінок для даного тижня студента. На виводі вказується інформація про студента, вчителя, оцінку та номер тижня. Метод getGrades повертає структуру оцінок журналу, яка містить оцінки кожного студента за кожний тиждень.У кінці, за допомогою unlock() відпускає lock. Метод getGrades повертає всі оцінки у формі Map<Student, List<Integer>>.

Student – містить інформацію про студента




Клас Teacher

```java
class Teacher implements Runnable {
    2 usages
    private List<Student> students;
    2 usages
    private GradingJournal gradingJournal;
    2 usages
    private String teacherName;

    4 usages
    public Teacher(List<Student> students, GradingJournal gradingJournal, String teacherName) {
        this.students = students;
        this.gradingJournal = gradingJournal;
        this.teacherName = teacherName;
    }

    @Override
    public void run() {
        Random random = new Random();
        int weeks = 3; // Кількість тижнів

        for (int week = 1; week <= weeks; week++) {
            for (Student student : students) {
                for (int i = 0; i < 5; i++) {
                    int grade = random.nextInt( bound: 101); // Випадкова оцінка від 0 до 100
                    gradingJournal.addGrade(student, grade, teacherName, week);
                    try {
                        Thread.sleep( millis: 100); // Затримка між виставленням оцінок
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
```

Було створено додатковий параметр numOfWeeks, який вказує кількість тижнів, протягом яких виконується оцінювання студентів. Всередині методу run виконується цикл для кожного тижня, а всередині цього циклу виконується цикл для кожного студента. Для кожного студента виставляється випадкова оцінка, і за допомогою методу addGrade класу GradingJournal додається оцінка до журналу. Після виставлення оцінки виконується затримка перед наступним виставленням оцінки.

Main

```java
public class Main {
    no usages
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
                new Student( name: "Dmytro Honcharenko IT-01"),
                new Student( name: "Clay Thompson IT-02"),
                new Student( name: "Andriy Voytenko IT-03"),
                new Student( name: "Kostya Peshkov IT-04")
        );

        GradingJournal gradingJournal = new GradingJournal(students);

        Teacher lecturer = new Teacher(students, gradingJournal, teacherName: "Lecturer");
        Teacher assistant1 = new Teacher(students, gradingJournal, teacherName: "Assistant 1");
        Teacher assistant2 = new Teacher(students, gradingJournal, teacherName: "Assistant 2");
        Teacher assistant3 = new Teacher(students, gradingJournal, teacherName: "Assistant 3");

        Thread lecturerThread = new Thread(lecturer);
        Thread assistant1Thread = new Thread(assistant1);
        Thread assistant2Thread = new Thread(assistant2);
        Thread assistant3Thread = new Thread(assistant3);

        lecturerThread.start();
        assistant1Thread.start();
        assistant2Thread.start();
        assistant3Thread.start();

        try {
            lecturerThread.join();
            assistant1Thread.join();
            assistant2Thread.join();
            assistant3Thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Map<Student, Map<Integer, List<Integer>>> grades = gradingJournal.getGrades();

        for (Student student : students) {
            System.out.println(student.getName() + ":");
            Map<Integer, List<Integer>> studentGrades = grades.get(student);
            for (int week = 1; week <= 3; week++) {
                List<Integer> weekGrades = studentGrades.get(week);
                System.out.println("Week " + week + ": " + weekGrades);
            }
            System.out.println();
        }
    }
}
```

В класі Main створюється список студентів і об'єкт GradingJournal для збереження оцінок. Далі створюються об'єкти вчителів (лектор і 3 асистенти) і потоки для кожного з них. Кожен вчитель працює протягом 3 тижнів, виконуючи виставлення оцінок студентам. Потім виконується очікування завершення всіх потоків. На завершення виводяться оцінки кожного студента по тижнях.

Результат

```
Grade added for student Kostya Peshkov IT-04 by Assistant 2: 70 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Assistant 2: 54 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Assistant 1: 38 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Lecturer: 26 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Assistant 3: 1 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Assistant 3: 73 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Lecturer: 28 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Assistant 2: 70 (Week 3)
Grade added for student Kostya Peshkov IT-04 by Assistant 1: 28 (Week 3)
Dmytro Honcharenko IT-01:
Week 1: [74, 69, 44, 63, 24, 7, 66, 35, 96, 97, 3, 87, 82, 55, 19, 17, 91, 58, 43, 16]
Week 2: [57, 79, 33, 96, 45, 91, 30, 41, 49, 87, 15, 5, 79, 64, 53, 84, 86, 2, 69, 53]
Week 3: [81, 25, 78, 72, 46, 50, 55, 72, 81, 59, 37, 100, 9, 71, 27, 32, 72, 39, 75, 61]

Clay Thompson IT-02:
Week 1: [6, 51, 48, 56, 48, 75, 72, 18, 19, 17, 94, 63, 1, 92, 55, 71, 56, 25, 89, 16]
Week 2: [39, 94, 12, 92, 98, 68, 55, 19, 35, 92, 38, 100, 12, 61, 8, 43, 90, 62, 53, 96]
Week 3: [54, 44, 50, 27, 84, 51, 94, 82, 93, 81, 100, 85, 66, 30, 14, 23, 76, 45, 64, 54]

Andriy Voytenko IT-03:
Week 1: [19, 3, 63, 63, 15, 55, 48, 90, 55, 72, 34, 93, 37, 87, 83, 16, 70, 16, 4, 21]
Week 2: [75, 21, 20, 88, 43, 81, 17, 18, 47, 51, 65, 78, 4, 40, 51, 31, 96, 14, 80, 65]
Week 3: [4, 23, 14, 49, 17, 31, 85, 31, 44, 82, 53, 48, 60, 84, 6, 4, 90, 85, 55, 20]

Kostya Peshkov IT-04:
Week 1: [50, 74, 87, 15, 83, 0, 12, 91, 21, 57, 72, 28, 39, 10, 100, 9, 59, 41, 12, 55]
Week 2: [58, 69, 24, 90, 100, 37, 63, 71, 68, 50, 58, 79, 98, 38, 43, 73, 57, 37, 54, 38]
Week 3: [13, 33, 93, 20, 85, 7, 91, 31, 38, 74, 90, 70, 54, 38, 26, 1, 73, 28, 70, 28]
```

Висновки

У цій лабораторній роботі було застосовано різні способи управління потоками, які є основними для забезпечення цілісності даних. Були використані синхронізовані методи та блоки для забезпечення взаємовиключності доступу до ресурсів. Також було використано джоїни для забезпечення відповідного

завершення роботи потоків перед продовженням виконання основного потоку. Крім того, використано повідомлення та Lock/ReentrantLock для ефективної координації та синхронізації роботи потоків. Також в ході лабораторної роботи було дотримано принципу цілісності даних та безпечного виконання операцій за допомогою використання синхронізації та взаємовиключності.

У ході виконання першого завдання було реалізовано програмний код, який був дан у лістингу. Перед тестуванням програми було проведено модифікацію, використовуючи різні методи управління потоками, щоб забезпечити коректну роботу програми.

Друге завдання вимагало модифікації програми Producer-Consumer, щоб замість рядків передавати масив чисел заданого розміру. Це було зроблено шляхом зміни типу даних, що передаються через Drop, і проведення відповідних змін в коді. Програма була протестована з різними розмірами масиву і продемонструвала коректну роботу.

Третє завдання передбачало реалізацію роботи електронного журналу для трьох груп студентів, де кожного тижня лектор і його три асистенти виставляють оцінки. Було створено класи Student, GradingJournal, Teacher та Main, які взаємодіють між собою для виставлення оцінок і зберігання результатів.

В результаті виконання лабораторної роботи було успішно реалізовано всі три завдання з використанням многопотокового програмування.