**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-01 Корнієнко В.С.*



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**.

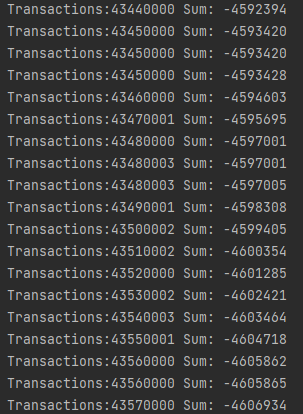
2. Реалізуйте приклад Producer-Consumer application (див. https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html ). Модифікуйте масив даних цієї програми, які читаються, у масив чисел заданого розміру (100, 1000 або 5000) та протестуйте програму. Зробіть висновок про правильність 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**.

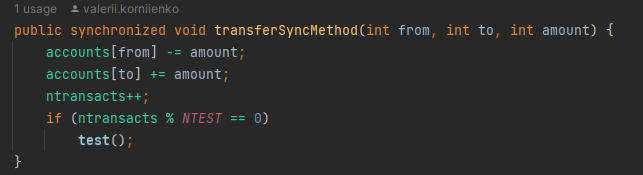
**Хід роботи:**

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

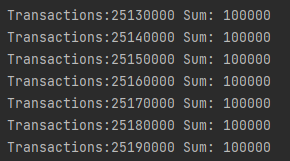


Можна побачити, що загальна сума постійно зменшується, що є некоректною роботою програми, оскільки сума має залишатися незмінною.

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

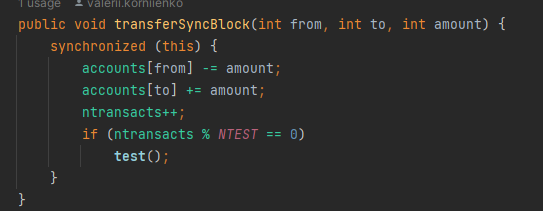


Тепер запустимо програму на виконання, але оновивши метод transfer:

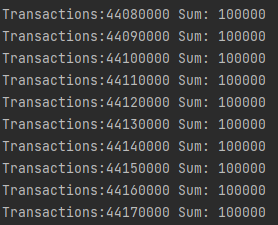


Бачимо що тепер сума незмінна навіть після великої кількості операцій. Можемо вважати таку поведінку коректною

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:

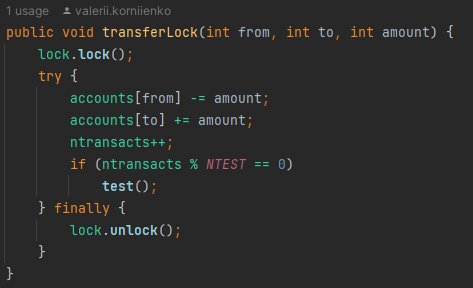


Тепер запустимо програму на виконання з оновленим методом transfer. Маємо наступні результати:

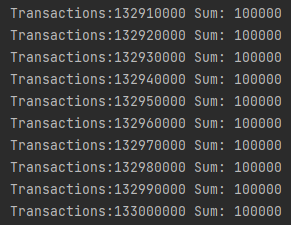


Бачимо що з цим методом синхронізації програма теж працює коректно

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

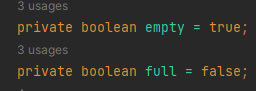


Знову запустимо програму на виконання щоб переконатися в правильності роботи оновленого методу:

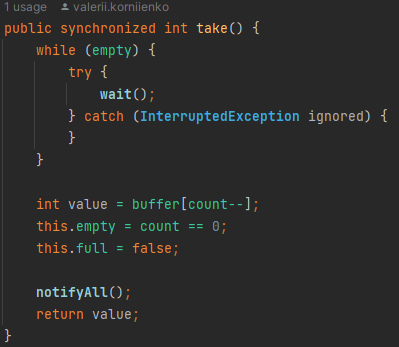
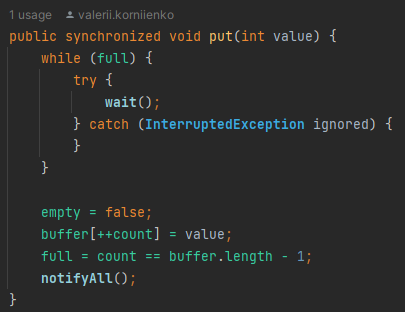


Програма працює правильно

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:



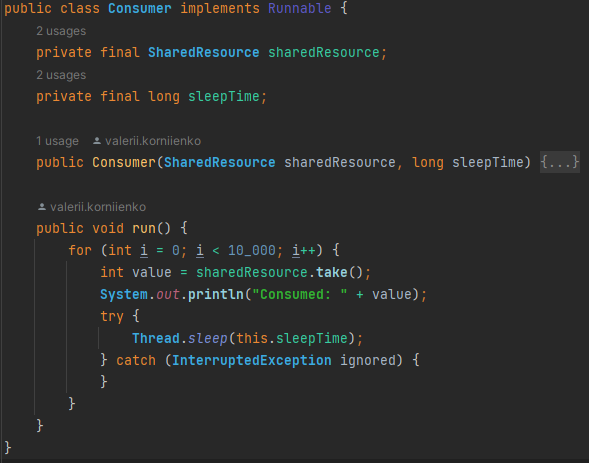
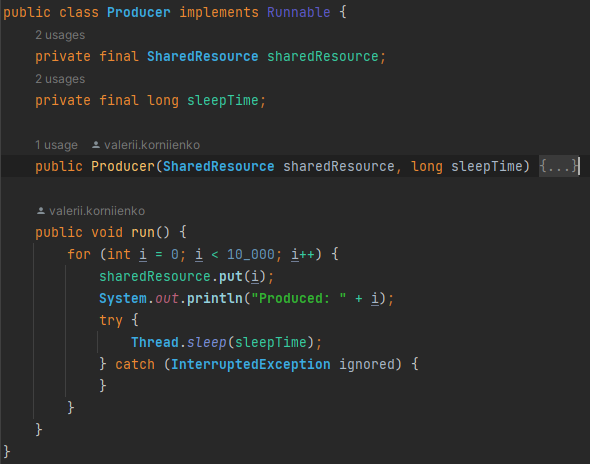
Тепер реалізуємо методи put i take в нашому буфері. Вони мають наступний вигляд:



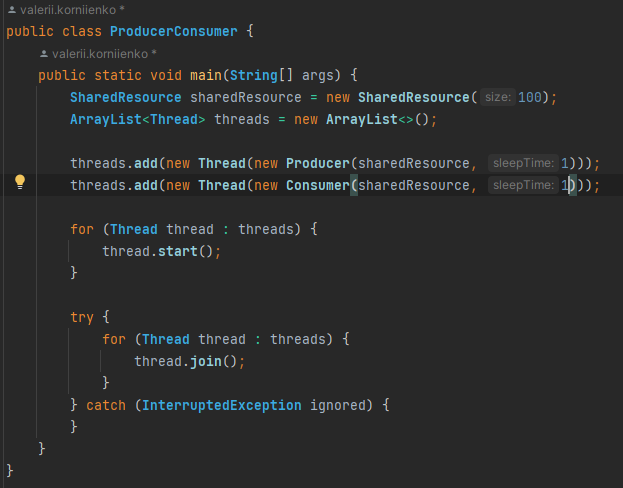
Як бачимо в методах put і take реалізовано механізм wait-notify в точності так як описувалося вище.

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.

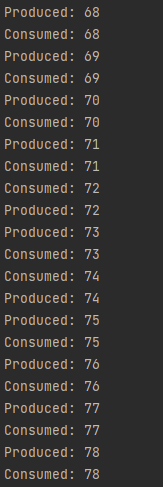
Класи Producer, Consumer мають наступний вигляд:



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



При запуску роботи програми маємо такі результати:



Даний результат є правильним

1. Для виконання 3 завдання я створив наступні класи:

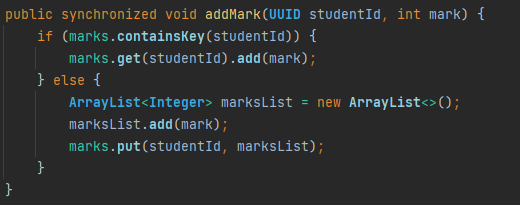
* Клас Student (містить інформацію про студента, його унікальний код)
* Клас Group (містить в собі список студентів які належать групі, назву групи)
* Клас Journal (містить список груп, список оцінок кожного окремого студента, метод для їх виставляння, тестування)
* Клас Teacher (implements Runnable, 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:

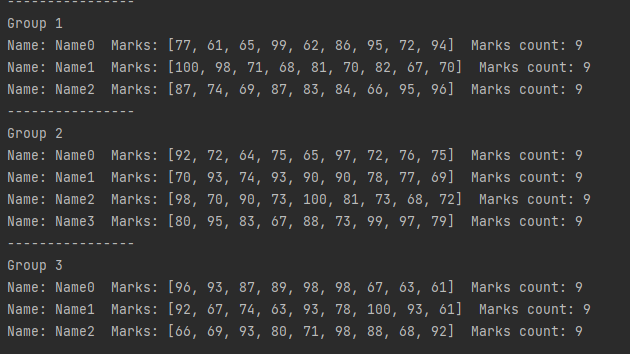


Як можемо зрозуміти, всі вчителі виставляють оцінки в журнал паралельно, тож операцію додавання оцінки треба синхронізувати. Для цього використаємо на методі addMark ключове слово synchronized. Тепер метод addMark виглядає наступним чином:



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:



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**();  
 }  
}

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:**

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;  
 }  
}

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**());  
 }  
 }  
 }  
}

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);  
 }  
 }  
 }  
}

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**();  
 }  
 }  
}