**Практичне завдання 1.2**

#### ***Приклад 1.*** *Одночасний доступ до ресурсу*

public class Account implements Runnable {

private int balance = 50;

public int getBalance() {

return balance;

}

public void withdraw(int amount) {

balance -= amount;

}

public void run() {

for (int x = 0; x < 5; x++) {

makeWithdrawal(10);

if (getBalance() < 0) {

System.out.println("account is overdrawn!");

}

}

}

private void makeWithdrawal(int amount) {

if (getBalance() >= amount) {

System.out.println(Thread.currentThread().getName()

+ " is going to withdraw");

try {

Thread.sleep(500);

} catch (InterruptedException ex) {

ex.printStackTrace();

}

withdraw(amount);

System.out.println(Thread.currentThread().getName()

+ " completes the withdrawal. The balance is "

+ getBalance());

} else {

System.out.println("Not enough in account for "

+ Thread.currentThread().getName()

+ " to withdraw " + getBalance());

}

}

}

public class AccountDanger{

public static void main(String[] args) {

Account account = new Account();

Thread one = new Thread(account);

Thread two = new Thread(account);

one.setName("Fred");

two.setName("Lucy");

one.start();

two.start();

}

}

#### 

#### ***Приклад 2.*** *Синхронізація доступу до ресурсу*

public class AccountDanger implements Runnable {

private Account account = new Account();

public static void main(String[] args) {

AccountDanger accountDanger = new AccountDanger();

Thread one = new Thread(accountDanger);

Thread two = new Thread(accountDanger);

one.setName("Fred");

two.setName("Lucy");

one.start();

two.start();

}

public void run() {

for (int x = 0; x < 5; x++) {

makeWithdrawal(10);

if (account.getBalance() < 0) {

System.out.println("account is overdrawn!");

}

}

}

private synchronized void makeWithdrawal(int amt) {

if (account.getBalance() >= amt) {

System.out.println(Thread.currentThread().getName()

+ " is going to withdraw");

try {

Thread.sleep(500);

} catch (InterruptedException ex) {

ex.printStackTrace();

}

account.withdraw(amt);

System.out.println(Thread.currentThread().getName()

+ " completes the withdrawal. The balance is "

+ account.getBalance());

} else {

System.out.println("Not enough in account for "

+ Thread.currentThread().getName()

+ " to withdraw " + account.getBalance());

}

}

}

#### 

#### ***Приклад 3.*** *Взаємне блокування*

public class DeadlockRisk implements Runnable {

private static class Resource {

}

private final Resource scissors = new Resource();

private final Resource paper = new Resource();

public void doSun() {

synchronized (scissors) { // May deadlock here

System.out.println(Thread.currentThread().getName()

+ " взяла ножиці для вирізання сонечка");

synchronized (paper) {

System.out.println(Thread.currentThread().getName()

+ " взяла папір для вирізання сонечка");

System.out.println(Thread.currentThread().getName()

+ " вирізає сонечко");

}

}

}

public void doCloud() {

synchronized (paper) { // May deadlock here

System.out.println(Thread.currentThread().getName()

+ " взяла папір для вирізання хмаринки");

synchronized (scissors) {

System.out.println(Thread.currentThread().getName()

+ " взяла ножиці для вирізання хмаринки");

System.out.println(Thread.currentThread().getName()

+ " вирізає хмаринку");

}

}

}

public void run() {

doSun();

doCloud();

}

public static void main(String[] args) {

DeadlockRisk job = new DeadlockRisk();

Thread masha = new Thread(job, "Маша");

Thread dasha = new Thread(job, "Даша");

masha.start();

dasha.start(); }}

#### 

#### ***Приклад 4.*** *Взаємодія потоків*

public class Producer implements Runnable {

private MyQueue myQueue;

public Producer(MyQueue myQueue) {

this.myQueue = myQueue;

}

@Override

public void run() {

for (int i = 0; i < 100; i++) {

myQueue.put(i);

}

}

}

public class Consumer implements Runnable {

private MyQueue myQueue;

public Consumer(MyQueue myQueue) {

this.myQueue = myQueue;

}

@Override

public void run() {

for (int i = 0; i < 100; i++) {

myQueue.get();

}

}

}

public class MyQueue {

private int n;

boolean valueSet = false;

public synchronized int get() {

while (!valueSet) {

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("Отримано: " + n);

valueSet = false;

notify();

return n;

}

public synchronized void put(int n) {

while (valueSet) {

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

valueSet = true;

this.n = n;

System.out.println("Відправлено: " + n);

notify();

}

}

public class ProducerDemo {

public static void main(String[] args) {

MyQueue myQueue = new MyQueue();

Consumer consumer = new Consumer(myQueue);

Producer producer = new Producer(myQueue);

Thread t1 = new Thread(consumer);

Thread t2 = new Thread(producer);

t1.start();

t2.start(); }

}

**Завдання**

1. Необхідно створити три потоки які змінюють один і той же об'єкт. Кожен потік повинен вивести на екран одну букву 100 раз, і потім збільшити значення символу на 1.

a) Створити клас, що розширює Thread.

б) Перевизначити метод run (), тут буде знаходиться синхронізований блок коду.

в) Для того щоб три об'єкти-потоку мали доступ до одного об'єкту, створюємо конструктор, що приймає на вхід StringBuilder об'єкт.

г) Синхронізований блок коду буде отримувати монітор на об'єкт StringBuilder з пункту в).

д) Усередині синхронізованого блоку коду виведіть StringBuilder на екран 100 разів, а потім збільште значення символу на 1.

е) У методі main () створіть один об'єкт класу StringBuilder, використовуючи символ 'a'. Потім створіть три екземпляри об'єкта нашого класу і запустіть потоки. (1 б.)

1. Змінити MyQueue:

a) Замість int n додати Queue <T> (MyQueue зробити узагальненим), яка буде містити об'єкти, що створюють Producer.

б) Додайте ще один об'єкт Consumer, який буде запускатися теж окремим потоком.

в) Виводьте на консоль який з об'єктів Consumer обробив об'єкт з черги.

г) Змініть цикл for на нескінченний цикл. (1 б.)

1. Обідають філософи. П'ять безмовних філософів сидять навколо круглого столу, перед кожним філософом стоїть тарілка спагеті. Виделки лежать на столі між кожною парою найближчих філософів. Кожен філософ може або їсти, або міркувати. Філософ може їсти тільки тоді, коли тримає дві виделки - взяту справа і зліва. Взяття кожної виделки і повернення її на стіл є роздільними діями, які повинні виконуватися одне за іншим.

<https://en.wikipedia.org/wiki/Dining_philosophers_problem>

Кожному філософу даємо окремий потік.

1) Спершу всім встановлюємо однакові пріоритети.

2) Потім даємо декільком філософам однаковий пріоритет, але вищий, ніж в інших.

3) Запустити з друком і аналізом моделі та продемонструвати що дає зміна пріоритетів. (1 б. )