In acest laborator vom lucra cu Threaduri si sincronizare.

Va reamintesc modul in care putem porni un thread.

Mai intai, definim o clasa care implementeaza Runnable, apoi instantiem o noua clasa Thread cu o instanta a acestei clase Runnable si ii dam start. Pe scurt, folosind o clasa anonima:

new Thread(new Runnable() {

public void run()

{

//code

}

}).start();

In momentul in care lucram cu mai multe threaduri pot sa apara probleme de sincronizare. Problemele de sincronizare sunt extrem de greu de depistat si debugat DUPA ce codul a fost scris, pentru ca se intampla aleator (cateodata codul va merge bine, cateodata va merge prost si cateodata poate sa mearga extrem de prost, aleator). Din acest motiv este vital ca fiecare programator sa fie constient de aspectul multi threading al aplicatiei la care lucreaza si sa prevada aceste probleme inca de la aparitia lor.

Intr-o aplicatie java, Threadurile au un singur rol: ele EXECUTA cod. Threadurile nu “au” obiecte. Adica nu are sens sa spunem ca obiectul X a fost creeat de Threadul Y deci e al lui si alte Threaduri vor bloca Threadul Y daca incearca sa acceseze X.

Pentru a executa cod, threadurile au acces la zona de memorie heap, acolo unde sunt stocate toate datele aplicatiei. De aici rezulta ca TOATE obiectele sunt accesibile din orice thread, cu conditia ca o referinta sa existe in cod. (De exemplu o clasa anonima poate sa vada membrul din clasa exterioara)

In aceste conditii este posibil ca doua (sau mai multe) Threaduri sa opereze modificari asupra unei singure zone de memorie “in paralel”. Sa luam un exemplu simplu:

O clasa are un membru “numar” de tip int si o metoda “incrementeaza” care arata asa:

public void incrementeaza()

{  
numar++;

}

Desi voi vedeti o singura instructiune acolo in realitate la numar++ sunt foarte multe. Spre exemplu, o scriere echivalenta ar fi asa:

public void incrementeaza()

{  
numar = numar+1;

}

Deja observati ca sunt doua instructiuni. In realitate ele sunt chiar mai multe dar haideti sa vedem ce se intampla in cazul acesta:

1. Se evalueaza expresia numar+1
2. Se salveaza valoarea expresiei in numar

Doar doua instructiuni. Acum sa luam in calcul cazul in care doua Threaduri executa codul in paralel. Instructiunile (din fericire) sunt atomice, adica nu pot fi executate pe jumatate chiar intr-un mediu in care se lucreaza in paralel. O sa consideram, pentru simplitate, instructiunile de mai sus ca fiind, fiecare din ele, atomice.

Threadul A executa codul. Aduce din memorie pe numar si evalueaza expresia numar+1. Deoarece numar are valoarea 10 (sa zicem….) expresia o sa aiba valoarea 11.

Intre timp Threadul B executa si el codul. Aduce din memorie pe numar, care NU a apucat sa fie modificat, si ecalueaza expresia numar+1 obtinand acelasi rezultat.

Threadul A executa numar = 11 (atat era rezultatul expresiei)

Threadul B executa si el numar = 11.

Observati ca in urma executiei codului numar ar fi trebuit sa fie 12, cu toate acestea numar a ramas doar 11.

De asemenea, sa remarcam ca daca A apuca sa execute ambele operatii inainte ca B sa inceapa executia nu s-ar fi manifestat problema. Din acest motiv am spus ca erorile pot sa apara “aleator”. Motivul pentru care este greu de debugat este ca problema apare doar cand mai multe fire ruleaza in paralel (daca ar fi existat doar A...nu erau probleme) iar computerul poate sa execute foarte repede codul si e capabil sa nu incurce borcanele spre deosebire de un om care de multe ori se pierde cand este vorba de lucrul in paralel (prea multe variabile).

Pentru rezolvarea problemei, aproape intotdeauna, solutia este sa marim “granularitatea” instructiunilor. Observati ca daca am fi considerat numar=numar+1; o singura instructiune care nu poate fi executata pe jumatate nu am fi avut probleme. In acest scop a fost creeat “synchronized”.

Blocul synchronized are urmatoarea forma:

synchronized (obiect)

{

// cod

}

Codul de mai sus trebuie citit asa: obtine “lockul” pentru obiectul “obiect” si abia dupa aceea incepi sa executi codul “//cod”. Dupa ce ai executat integral codul “//cod” elibereaza “lockul” pentru “obiect”.

Sa ne intoarcem la problema de mai sus si sa rescriem codul astfel:

public void incrementeaza()

{

synchronized(this)

{  
numar = numar +1;

}

}

Threadul A incepe executia. Incearca sa obtina “lockul” pentru obiectul this (ce inseamna this???), reuseste, deoarece nimeni nu are “lockul”. Apoi calculeaza expresia numar+1 si obtine 11 (cu datele din exemplul de mai sus).

Threadul B incepe executia. Incearca sa obtina “lockul” pe acelasi obiect this, dar nu reuseste, A avand “lockul”. Deci B se blockeaza si asteapta (intra in state BLOCKED). Asteapta pana cand “lockul” pe this este eliberat.

Threadul A continua si scrie 11 in numar.

Threadul B inca asteapta.

Threadul A elibereaza lockul si isi vede de treaba in continuare.

Threadul B poate face lock, face numar+1 si obtine 12. Apoi scrie numar = 12.

Observati ca nu mai este posibil sa avem probleme.

Pentru cazul in care vrem sa facem syncrhonized pe this, si tot codul dintr-o metoda sa intre in blocul synchronized exista o varianta mai scurta (si perfect echivalenta) cu ce am scris mai sus:

public syncrhonized void incrementeaza()

{

numar = numar +1;

}

Cele doua functii sunt echivalente, doar ca de aceasta data, inainte de chemarea functiei se va obtine lockul, nu in interiorul functiei. Efectul e identic.

Sa luam acum un alt caz. Sa consideram ca “numar” este un membru static. Asta inseamna ca oricat de multe instante ale clasei in care lucram vom avea o singura adresa de memorie pentru “numar”. Deci sa consideram:

public class Test

{

static int numar;

public synchronized void incrementeaza()

{

numar=numar+1;

}

}

Observati synchronized. Vor exista probleme? De ce? Daca da, cand/cum apar?

Daca vom folosi o singura instanta a obiectului Test in ambele Threaduri, totul va fi ok. Cand se face synchronized pe this cele doua Threaduri nu pot intra in acelasi timp in metoda. Totusi, daca avem mai multe instante ale lui Test si chemam metoda pe instante diferite atunci synchronized(this) nu va avea practic niciun efect, ambele threaduri obtinand lock pe instanta ei, dar lucrand pe variabila numar care este comuna.

Exista doua solutii. Una ar fi sa facem functia incrementeaza “static” si sa o lasam synchronized.

public static synchronized void incrementeaza()

{

numar=numar+1;

}

Pentru a vedea mai clar ce se intampla am sa rescriu codul de mai sus:

public static void incrementeaza()

{

syncrhonized(Test.class)

{

numar=numar+1;

}

}

Deci, atunci cand metoda este satica si folosim synchronized lockul se va face pe obiectul Class. Ce este un obiect Class??? Cum exista un singur obiect Test.class in toata aplicatia, lockul se va face corect si nu vor putea intra mai multe threaduri sa execute numar=numar+1l deci nu vom avea, din nou, probleme.

Observam ca desi cele doua scrieri sunt mai mult sau mai putin echivalente, in cazul al doilea putem sa renuntam la modificatorul “static” al metodei fara sa afectam monitorul (obiectul pe care se face lock). Deci pentru a rezolva problema noastra ar trebui sa folosim a doua scriere.

Retinem: orice INSTANTA de obiect poate fi folosita intr-un block synchronized.

Retinem: putem folosi variabile, iar variabilele pot contine null la synchronized. In acesta caz vom obtine NullPointerException deoarece programul incearca sa faca .lock() pe obiectul nostru. lock() este o functie declarata si definita in clasa Object, deci apare in orice obiect.

Retinem: este garantat ca la iesirea (din orice motiv) dintr-un bloc synchronized lockul va fi eliberat iar toate threadurile care asteapta vor fi anuntate.

Retinem: daca mai multe threaduri asteapta pentru acelasi lock, nu putem sti care va obtine prima lockul, dar stim sigur ca doar una va obtine lockul.

Sa zicem ca avem:

public class Test

{

int numar;

public synchronized void incrementeaza()

{

numar=numar+1;

}

public synchronized void decrementeaza()

{

numar=numar-1;

}

}

Este aceasta clasa “thread safe”? De ce? Daca nu, cand/cum pot sa apara probleme?

Threadurile pot sa isi trimita mesaje unul altuia, prin intermediul obiectelor pe care se face sincronizarea.

Sa consideram urmatoarea clasa:

public class Test

{

int numar;

Object lockObject;

public synchronized void play()

{

numar=numar+1;

wait();

numar=numar-1;

}

public synchronized void afiseaza()

{

System.out,println(numar);

notifyAll();

System.out,println(numar);

}

}

Observam apelul lui wait(). Wait va bloca threadul curent (il trece in starea WAITING, sau TIMED\_WAITING daca ii dam si un argument). Sa consideram ca avem din nou threadurile A si B. A executa play(). A face numar = numar+1 (sa spunem ca avem valoarea 10 in numar, deci acum va fi 11). Apoi A intra pe wait si se blocheaza. Ce asteapta A?

A asteapta ca altcineva sa faca notify sau notifyAll. Sa zicem ca B incepe sa execute afiseaza. B scrie la consola 11 si apoi apeleaza notifyAll. NotifyAll va trimite un semnal catre TOATE threadurile care asteapta PE OBIECTUL PE CARE A FOST APELAT ca trebuie sa se trezeasca din wait si sa isi continue executia. In acest caz A se va trezi.

Ce credeti, exista vreo sansa ca B sa afiseze din nou 11? De ce da, de ce nu?

Observati vreo problema cu B care vrea sa intre in afiseaza, desi e sincronizata metoda?

In momentul in care facem wait(), este eliberat automat lockul pe obiectul pe care am facut lock. Asta inseamna ca pentru a face wait TREBUIE SA AVEM ACEL LOCK! Daca metoda play nu era sincronizata (pe this) atunci primeam o exceptie la rulare! ESTE FOARTE IMPORTANT ACEST ASPECT! Dupa ce am facut wait, lockul e eliberat si B poate sa intre linistit in afiseaza().

De asemenea, cand ne revenim din wait() TREBUIE SA OBTINEM DIN NOU lockul! B face notifayAll() deci sigur e trezit si A, dar inca nu elibereaza lockul deci A nu isi continua executia si este garantat faptul ca B apuca sa afiseze din nou 11. Nu exista nicio sansa ca numar sa redevina 10 inainte ca B sa il poata afisa. Dupa ce B termina executia lui afiseaza, lockul e eliberat si A il poate ocupa pentru a continua treaba.

Retinem: wait() elibereaza lockul pe care s-a facut wait(), nu si restul. Acest aspect este foarte important.

Retinem: notify()/notifyAll() notifica un thread ales aleator/toate threadurile care asteapta cu wait pe un monitor.

Daca wait(1000) este un wait care se trezeste automat dupa 1000 milisecunde, care e diferenta intre Thread.sleep(1000) si wait(1000)?

Pe exercitiul atasat, metoda handlerMechanismDemo() vom face niste modificari.

1. Rulati programul asa cum este acum. Puteti introduce un numa pe masura ce numaratoarea merge pentru a vedea ce se intampla.
2. Modificati programul astfel incat dupa ce este afisat “Milestone” sa se opreasca numaratoarea si sa se astepte pana cand userul introduce un numar de la tastatura. Observati mesajul “Called from Thread”...
3. Modificati programul astfel incat callMe sa fie chemat de threadul secundar.

Am vrut sa va exemplific diferenta intre executia unui handler pe Threadul care astepta evenimentul si pe Threadul care a declansat evenimentul.

Main:

<https://goo.gl/voio9G>

Zip:

<https://drive.google.com/open?id=0B5ar2tHw-X9vRS1PZlBsV3NRcms>