Programare concurentă - fire de execuție (Threads)

Limbajul Java suportă nativ noțiunea de fir de execuție ([thread](http://en.wikipedia.org/wiki/Thread_(computing))), adică mai multe clase ce rulează în paralel, dar care fac parte din aceeași aplicație. Un exemplu concret ar fi un server care acceptă și administrează mai multe conexiuni în același timp. Există două moduri de implementare al unui fir de execuție:

* prin extinderea clasei [java.lang.Thread](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html);
* prin implementarea interfeței [java.lang.Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html).

Clasa **Thread** ca și interfața **Runnable** definesc o metodă numită *run()*. Această metodă este metoda de start pentru thread-ul nou, analog metodei *public static void main(String[])* pentru thread-ul principal. Această metodă *run()* poate fi apelată în două moduri:

* apelând direct metoda *run()*, în care caz execuția se face ca un apel obișnuit de metodă;
* apelând metoda *start()*, care pornește un thread nou care va începe execuția cu metoda *run()*.

**public** **class** **PrintThread** **extends** Thread{

**private** int index;

**public** PrintThread(int \_index){

index = \_index;

}

**public** void run(){

**for**(int i=0; i<5; i++){

System.out.println("This is thread " + index);

**try**{

*//pause for 0.5 seconds (500 ms)*

Thread.sleep(500);

}**catch**(InterruptedException \_ie){

System.out.println(\_ie.getMessage());

}

}

}

}

Acesta este apelul obișnuit al metodei *run()*. Programul va afișa întâi de 5 ori textul pentru thread-ul 1, apoi de 5 ori pentru thread-ul 2, apoi pentru thread-ul 3, etc:

**public** **class** **NormalStarter**{

**public** **static** void main(String[] \_args){

**for**(int i=0; i<5; i++){

PrintThread \_thread = **new** PrintThread(i + 1);

\_thread.run();

}

}

}

Acesta este apelul metodei *run()* ca thread nou. Programul va afișa în același timp textele de la toate thread-urile:

**public** **class** **ThreadStarter**{

**public** **static** void main(String[] \_args){

**for**(int i=0; i<5; i++){

PrintThread \_thread = **new** PrintThread(i + 1);

\_thread.start();

}

}

}

Același lucru folosind interfața **Runnable**:

**public** **class** **PrintRunnable** **implements** Runnable{

**private** int index;

**public** PrintRunnable(int \_index){

index = \_index;

}

**public** void run(){

**for**(int i=0; i<10; i++){

System.out.println("This is thread " + index);

**try**{

*//pause for 1 second (1000 ms)*

Thread.sleep(1000);

}**catch**(InterruptedException \_ie){

System.out.println(\_ie.getMessage());

}

}

}

}

**public** **class** **ThreadStarterRunnable**{

**public** **static** void main(String[] \_args){

**for**(int i=0; i<5; i++){

*// polymorphism: PrintRunnable is a Runnable*

Runnable \_runnable = **new** PrintRunnable(i + 1);

PrintThread \_thread = **new** Thread(\_runnable);

\_thread.start();

}

}

}

Sincronizarea thread-urilor - semaforul

Există situații, în aplicații multithread-ed, în care mai multe thread-uri accesează aceeași resursă. În unele din aceste situații, dacă unele metode sau zone de program ale acestor resurse comune sunt accesate de mai multe thread-uri în același timp, pot apărea condiții limită care duc la un comportament incorect al aplicației. Aceste zone trebuie să fie executate *atomic*, adică în timpul execuției lor, nici un alt thread nu trebuie să întrerupă thread-ul curent sau să execute aceeași bucată de cod.

În mașina virtuală, acest lucru se realizează cu ajutorul unui sistem de monitoare. Monitorul este un câmp ascuns definit în clasa Object (astfel încât orice obiect poate fi folosit pe post de monitor) care contorizează thread-urile care accesează una sau mai multe bucăți de program. Aceste bucăți de program se numesc *sincronizate*. Dacă un thread urmează să intre într-o zonă sincronizată, atunci mașina virtuală verifică dacă monitorul e liber, adică dacă nici un alt thread nu execută instrucțiuni din vreo zonă sincronizată de acel monitor. Dacă monitorul e ocupat (adică dacă alt thread îl deține), arunci thread-ul curent se oprește, așteptând eliberarea lui. Dacă este liber, atunci thread-ul curent îl ocupă, începând execuția codului.

În Java, cuvântul cheie pentru sincronizarea unei bucăți de cod este *synchronized*.

**public** **class** **Stack**{

**public** **static** **final** int MAX\_STACK\_SIZE = 128;

**private** Object[] stack;

**private** int stackTop;

**public** Stack(){

**this**(MAX\_STACK\_SIZE);

}

**public** Stack(int \_maxSize){

stack = **new** Object[\_maxSize];

stackTop = 0;

}

**public** **synchronized** boolean empty(){

**return** stackTop == 0;

}

**public** **synchronized** boolean full(){

**return** stackTop == stack.length;

}

**public** Object pop() **throws** Exception{

**synchronized**(**this**){

**if**(!empty()){

**return** stack[--stackTop];

}

}

**throw** **new** Exception("Stack empty");

}

**public** void push(Object \_obj) **throws** Exception{

**synchronized**(**this**){

**if**(!full()){

stack[stackTop++] = \_obj;

}

}

**throw** **new** Exception("Stack full");

}

}

În exemplul de mai sus, dacă două thread-uri apelează în același timp metodele *push(Object)* și *pop()*, atunci, dacă acestea nu ar fi sincronizate, ar putea apărea situații în care contorul *stackTop* ar putea fi incrementat de un thread, apoi decrementat de al doilea înainte ca primul să efectueze scrierea în vectorul *stack*. Dar folosind cuvântul cheie *synchronized*, ne-am asigurat că nici una din metodele declarate astfel și nici una din bucățile de program sincronizate nu vor fi executate în același timp de thread-uri diferite.

Regulă: O clasă nu poate executa o porțiune de program sincronizată al cărei monitor aparține altui thread DAR poate executa o metodă sincronizată de un monitor care îi aparține. Ca exemplu, zona sincronizată din metoda *push(Object)* apelează metoda *full()* care este la rândul ei sincronizată.

Dacă o metodă este declarată *synchronized*, acest lucru este echivalent cu a declara tot conținutul metodei într-un bloc *synchronized(this)*. Altfel spus, o metodă declarată *synchronized* este automat sincronizată folosind ca monitor obiectul curent.

Dacă o metodă statică este declarată *synchronized*, acest lucru este echivalent cu a declara tot conținutul metodei într-un bloc *synchronized(<nume\_clasa>.class)*. Altfel spus, o metodă declarată *synchronized* este automat sincronizată folosind ca monitor obiectul de tip **Class** asociat clasei:

**public** **class** **StaticSync**{

**public** **static** **synchronized** void printSmth(){

System.out.println("Smth");

}

**public** **static** void printSmthElse(){

**synchronized**(StaticSync.class){

System.out.println("SmthElse");

}

}

}

*Race conditions* - bariera

În exemplul anterior, dacă considerăm două thread-uri, unul care pune elemente pe stivă și unul care le consumă, observăm că dacă oricare din thread-uri este mai rapid decât celălalt, se ajunge în situația în care se aruncă o excepție, ori pentru că stiva e plină, ori pentru că s-a golit. Aici apare ceea ce se numește *race condition*, adică există una sau două zone de program executate de două thread-uri diferite și în care unul din thread-uri trebuie să ajungă înaintea celuilalt ca programul să se desfășoare corect. În exemplul anterior, dacă stiva este goală, atunci thread-ul care pune un element pe stivă trebuie să ajungă la metoda *push(Object)* înainte ca celălalt thread să apeleze metoda *pop()*, în caz contrar generându-se o excepție.

Această problemă se rezolvă folosind un sistem de bariere, adică un sistem care oprește unul din thread-uri într-un anumit loc până când o condiție este îndeplinită (de cele mai multe ori, când alt thread ajunge în locul potrivit). În Java, acest lucru se realizează cu ajutorul metodelor *wait()*/ *wait(long)*/ *wait(long, int)* și *notify()*/ *notifyAll()*.

Metodele *wait()*/ *wait(long)*/ *wait(long, int)* blochează thread-ul curent până când un alt thread apelează una din metodele *notify()*/ *notifyAll()*, sau până când timpul dat ca argument a expirat.

Regulă: Apelurile metodelor de tip *wait()* se fac obligatoriu într-un bloc *synchronized* și se folosește ca referință obiectul monitor:

**public** **class** **Stack**{

**public** **static** **final** int MAX\_STACK\_SIZE = 128;

**private** Object[] stack;

**private** int stackTop;

**public** Stack(){

**this**(MAX\_STACK\_SIZE);

}

**public** Stack(int \_maxSize){

stack = **new** Object[\_maxSize];

stackTop = 0;

}

**public** **synchronized** boolean empty(){

**return** stackTop == 0;

}

**public** **synchronized** boolean full(){

**return** stackTop == stack.length;

}

**public** Object pop() **throws** InterruptedException{

**synchronized**(**this**){

**while**(empty()){

**this**.wait();

}

**this**.notifyAll();

**return** stack[--stackTop];

}

}

**public** void push(Object \_obj) **throws** InterruptedException{

**synchronized**(**this**){

**while**(full()){

**this**.wait();

}

stack[stackTop++] = \_obj;

**this**.notifyAll();

}

}

}

Metodele de tip *notify()* reactivează threadurile oprite folosind metodele *wait()*. Acestea nu trebuie obligatoriu să fie apelate dintr-un bloc sincronizat, dar este recomandat. Se folosește ca referință tot obiectul de tip monitor utilizat pentru apelul metodelor *wait()*.

Un thread reactivat folosind *notify()* trebuie să aștepte eliberarea monitorului pentru a executa zona sincronizată, ca orice alt thread. Observați utilizarea acestei funcționalități în metoda *pop()*, unde s-a apelat *notifyAll()* înainte de extragerea elementului de pe stivă, pe linia următoare. Totuși, clasa funcționează corect, pentru că thread-ul care introduce pe stivă un element nou nu va intra în execuție până când thread-ul care apelează *notifyAll()* nu iese din zona sincronizată, adică după extragerea unui element de pe stivă.

Resurse externe:

1. <http://docs.oracle.com/javase/tutorial/essential/concurrency/>
2. <http://www.tutorialspoint.com/java/java_multithreading.htm>