Laborator 9:

Fire de execuţie

Întocmit de: Adina Neculai

Îndrumător: Asist. Drd. Gabriel Danciu

20 noiembrie 2011

I. NOŢIUNI TEORETICE

A. Ce este un fir de execuţie?

Înainte de a defini conceptul de fir de execuţie sau thread, ne vom opri asupra conceptului de *multitasking*. Acesta se referă la capacitatea unui computer de a executa mai multe programe (task-uri, procese) în acelaşi timp. Având în vedere această noţiune, putem defini conceptul de *multithreading* care se referă la execuţia mai multor secvenţe de cod în acelaşi timp, în cadrul aceluiaşi program. O astfel de secvenţă de cod se numeşte *fir de execuţie* sau *thread*.

Un thread se execută în contextul unui program şi nu poate fi tratat decât în cadrul unui program. Exemplele prezentate până acum au dispus de un sigur fir de execuţie, adică de o singură structură secvenţială de instrucţiuni.

B. Lucrul cu fire de execuţie

Limbajul Java pune la dispoziţia programatorului, in pachetul *java.lang*, două clase şi o interfaţă pentru lucrul cu thread-uri:

- clasa *Thread*: lucru cu thread-uri ca entităţi separate;
- clasa *ThreadGroup*: crearea unor grupuri de thread-uri în vederea tratării acestora în mod unitar;
- interfaţa *Runnable*: lucru cu thread-uri ca entităţi separate.

1. Crearea unui thread derivând clasa Thread

Operaţiile următoare reprezintă numărul minim de paşi ce trebuie îndepliniţi pentru lucrul cu un thread creat prin moştenirea clasei Thread:

1. crearea unei clase obişnuite care moşteneşte clasa *Thread*.
   class MyThread extends Thread{
   }

2. suprascrierea metodei *public void run()* a clasei Thread în clasa derivată, MyThread. Instrucţiunile din această metodă trebuie să implementeze ceea ce se doreşte ca thread-ul să facă. Aceasta poate apela alte metode, folosi alte clase şi declara variabile ca orice metodă.Metoda *run()* este apelată atunci când se execută un thread.
   Pentru a înţelege mai bine se face o paralelă cu metoda *main*. Aceasta care este apelată atunci când se execută o aplicaţie Java. Mai mult, atunci când JVM este pornită, se porneşte o dată cu ea un thread care apelează metoda *main*.

3. instanţierea unui obiect thread prin intermediul operatorului *new*;
   MyThread obThread = new MyThread();

4. pornirea thread-ului instanţiat prin apelul metodei *start()* moştenită din clasa Thread.
   obThread.start();

Urmăriţi un exemplu de lucru cu thread-uri prin derivarea clasei Thread în secţiunea II A.

2. Crearea unui thread implementând interfaţa Runnable

Operaţiile care trebuie îndeplinite pentru a putea crea un thread prin implementarea interfeţei *Runnable* sunt următorele:

1. crearea unei clase care să implementeze interfaţa *Runnable*.
   class MyRunnable implements Runnable {
   }

2. clasa care implementează interfaţa *Runnable* trebuie să suprascrie toate metodele definite în aceasta, dar cum interfaţa conţine doar metoda *run()*, ea este singura metodă ce trebuie suprascrisă. Metoda *run()* implementează ceea ce se doreşte ca thread-ul să facă.

3. se instanţiază un obiect al clasei create utilizând operatorul *new*.
   MyRunnable myRunnableObj = new MyRunnable();

4. se instanţiază un obiect al clasei *Thread* utilizând un constructor ce are ca şi parametru un obiect de tipul clasei care implementează interfaţa *Runnable*.
   Thread thread = new Thread(myRunnableObj);

5. se porneşte thread-ul.

   thread.start();

Pentru o mai bună înţelegere urmăriţi exemplul prezentat în secţiunea II B.

3. Clasa Thread vs interfaţa Runnable

Clasa *Thread* defineşte o serie de metode ce pot fi suprascrise de clasa ce derivă din *Thread*. Singura care trebuie suprascrisă obligatoriu este *run()*. Însă, această condiţie se impune şi in cazul clasei care implementează interfaţa *Runnable*. Aşadar, dacă nu se doreşte suprascrierea altor metode din *Thread*, atunci este mai bine să folosim a doua metodă.

Mai mult, cum în Java nu este permisă moştenirea multiplă, creare unui thread prin implementarea interfeţei *Runnable* devine un avantaj pentru că acel thread poate să moştenească funcţionalităţi definite în alte clase Java.

C. Controlul unui fir de execuţie

Un thread se poate afla la un moment dat într-una din stările următoare: *nou creat, în execuţie, blocat, terminat.*

Clasa *Thread* conţine o serie de metode ce controlează execuţia unui thread. Iată câteva dintre cele mai importante:

Nume metodă	**Descriere**
String getName()	se obţine numele thread-ului
int getPriority()	se obţine prioritatea thread-ului
boolean isAlive()	determină faptul că un thread mai este în execuţie sau nu
void join()	forţează un thread să aştepte terminarea altuia
void run()	entry point pentru un thread
static void sleep(long millis)	suspendă execuţia unui thread pentru *millis* milisecunde
void start()	începe execuţia unui fir printr-un apel *run()*
void stop()	opreşte execuţia unui thread

Pentru mai multe informaţii despre metodele clasei *Thread* consultaţi API-ul clasei.

D. Priorităţi

Aparent thread-urile se execută simultan, dar la un moment dat doar un singur thread are acces la resursele de sistem. Fiecare thread obţine o perioadă de timp în care are acces la resurse, accesul făcându-se în mod organizat de către un planificator. Una dintre regulile pe care se bazează planificatorul o reprezintă mecanismul de priorităţi.

Având în vedere cele spuse mai sus, fiecărui thread i se asociază un nivel de prioritate (un număr întreg). Cu cât acest nivel este mai mare cu atât thread-ul va avea prioritate mai mare la resurse. Dacă totuşi mai multe thread-uri au acelaşi nivel de prioritate, mecanismul de priorităţi dă dreptul de execuţie fiecăruia printr-un algoritm circular.

Lucrul cu priorităţi este exemplificat în secţiunea II D.

E. Sincronizarea firelor de execuţie

Există cazuri în care mai multe thread-uri accesează aceeaşi resursă aparent în acelaşi timp. Pentru a evita astfel de situaţii, care pot crea confuzie, limbajul Java oferă conceptul de *monitor*. Acest lucru se realizează prin declararea unor metode, blocuri de cod ale unui obiect *synchronized*. Modificatorul *synchronized* permite unui singur thread accesul, la un moment dat, la acele metode sau blocuri de cod ale obiectului. Aşadar, dacă un thread apelează o metodă *synchronized* a unui obiect, un alt thread nu are acces la ea până când primul thread nu a terminat-o de executat.

Exemplul este prezentat în secţiunea II E.

II. PREZENTAREA LUCRĂRII DE LABORATOR

A. Crearea unui thread derivând clasa Thread

Exemplul este destul de simplu şi nu necesită explicaţii suplimentare o dată ce au fost urmăriţi paşii din secţiunea I B 1.

```
1 public class MyThread extends Thread{
2    private int countDown = 10;
3
4    public void run(){
5       System.out.println("\t begin run method...");
6       while(countDown >=0){
7          System.out.println("\t\t Steps letft to do: "+countDown);
```

```
8             countDown--;
9         }
10        System.out.println("\t end run method!");
11    }
12 }
```

```
1  public class DemoThread {
2
3     public static void main(String[] args) {
4        System.out.println("begin main...");
5        MyThread obThread = new MyThread();
6        System.out.println("start the thread");
7        obThread.start();
8        System.out.println("still in main...");
9     }
10
11 }
```

Un singur lucru pare să fie curios: apariţia mesajului *"still in main..."* imediat după mesajul *"start the thread"*. Acest lucru se întâmplă din cauza faptului că metoda *main* are propriul său fir de execuţie în care rulează. Prin apelul metodei *start()* a thread-ului se cere maşinii virtuale Java crearea şi pornirea unui nou thread. Din acest moment JVM preia controlul execuţiei programului creând contextul necesar unui thread şi în acest context se apelează metoda *run()*. Ca urmare se iese imediat din metoda *start()*.

B. Crearea unui thread implementând interfaţa Runnable

Exemplul prezentat mai jos este puţin mai complex în sensul că profităm de avantajele implementării unei interfeţe, putând totuşi moşteni orice altă clasă.

```
1  public class Display {
2     public void printMessage(String message){
3        System.out.println(message);
4     }
5  }
```

Clasa *MyRunnable* moşteneşte clasa *Display*, dar beneficiază de proprietăţile unei clase de tip thread prin implementarea interfeţei *Runnable*. Unele dintre îmbunătăţirile ce sunt aduse programului sunt mutarea instanţierii thread-ului (se trimite ca parametru obiectul curent, deci un obiect de tipul MyRunnable) şi pornirea acestuia în constructorul clasei *MyRunnable*.

```
1  public class MyRunnable extends Display implements Runnable {
2     private int countDown = 10;
3     private Thread thread;
4
5     public MyRunnable(){
```

```
6        thread = new Thread(this);
7        System.out.println("start␣the␣thread");
8        thread.start();
9     }
10
11    public void run() {
12       printMessage("\t␣begin␣run␣method...");
13       while (countDown >= 0) {
14          printMessage("\t\t␣Steps␣letft␣to␣do:␣" + countDown);
15          countDown--;
16       }
17       printMessage("\t␣end␣run␣method!");
18    }
19 }
```

Clasa *DemoRunnable* instanţiază un obiect de tipul MyRunnable. Astfel se apelează constructorul unde se creează şi se porneşte thread-ul.

```
1  public class DemoRunnable {
2     public static void main(String[] args) {
3        System.out.println("begin␣main...");
4        MyRunnable myRunnableObj = new MyRunnable();
5        System.out.println("still␣in␣main...");
6     }
7  }
```

C. Controlul unui fir de execuţie

Exemplul următor prezintă modul de utilizare al metodelor clasei *Thread*.

Clasa MyThread moşteneşte clasa *Thread* şi creează un constructor cu un parametru ce apelează constructorul clasei de bază. Metoda *run()* este identică cu cea din exemplele anterioare.

```
1  public class MyThread extends Thread{
2     private int countDown = 5;
3
4     public MyThread(String name){
5        super(name);
6     }
7
8     public void run(){
9        System.out.println("\t␣begin␣run␣method...");
10       while(countDown  >=0){
11          System.out.println("\t\t␣Steps␣letft␣to␣do:␣"+countDown);
12          countDown--;
13       }
14       System.out.println("\t␣end␣run␣method!");
15    }
16 }
```

Clasa *DemoThread* creează două thread-uri. Pe primul îl porneşte şi îl suspendă pentru 4000 de milisecunde. Observaţi starea în care se află thread-ul după suspendarea temporară a acestuia

(apelul metodei *isAlive()*). Apoi, se porneşte thread-ul al doilea şi se blochează resursele acestuia până când se reia execuţia lui (apelul metodei *resume()*). Observaţi starea thread-ului după apelul metodei *suspend()*.

În cele din urmă, se forţează blocarea thread-ului principal până când myThread2 işi termină execuţia (linia de cod 29).

```
1  public class DemoThread {
2     public static void main(String[] args) throws InterruptedException {
3        System.out.println("begin main thread...");
4        MyThread myThread1 = new MyThread("thread 1");
5        myThread1.start();
6        System.out.println(myThread1.getName());
7        try {
8           myThread1.sleep(4000);
9        } catch (InterruptedException e) {
10          e.printStackTrace();
11       }
12       if (myThread1.isAlive()){
13          System.out.println(myThread1.getName()+" is alive!");
14       }else{
15          System.out.println(myThread1.getName()+" is terminated!");
16       }
17
18       MyThread myThread2 = new MyThread("thread 2");
19       System.out.println(myThread2.getName());
20       myThread2.start();
21       myThread1.suspend();
22       if (myThread2.isAlive()){
23          System.out.println(myThread2.getName()+" is alive!");
24       }else{
25          System.out.println(myThread2.getName()+" is terminated!");
26       }
27
28       System.out.println("before main thread sleeps");
29       Thread.sleep(2000);
30       System.out.println("after main thread has slept");
31
32       myThread2.sleep(2000);
33       myThread2.resume();
34       try {
35          myThread2.join();
36       } catch (InterruptedException e) {
37          e.printStackTrace();
38       }
39       myThread2.sleep(1000);
40       System.out.println("end main thread!!!");
41    }
42 }
```

D. Priorităţi

Clasa *PriorityThread* implementează interfaţa *Runnable* şi are un constructor cu un parametru ce setează nivelul de prioritate al thread-ului şi care instanţiază un obiect de tip thread, prezent ca atribut al clasei. Metoda *run()* apelează metoda *toString()* a clasei *Thread* cât timp atributul *countDown* este mai mare ca 0.

```
1  public class PriorityThread implements Runnable {
2     private int countDown = 5;
3     private int priority;
4     public Thread thread;
5
6     public PriorityThread(int priority) {
7        this.priority = priority;
8        thread = new Thread(this);
9     }
10
11    public void run() {
12       thread.setPriority(priority);
13       System.out.println("\t begin run method...");
14       while(countDown  >=0){
15          System.out.println(thread.toString()+" countDown: "+countDown);
16          countDown--;
17       }
18       System.out.println("\t end run method!");
19    }
20 }
```

Clasa *DemoPriority* creează două obiecte de tipul PriorityThread cu parametri declaraţi ca şi constante în clasa *Thread*. Se porneşte execuţia acestora şi thread-ul principal este obligat să aştepte terminarea execuţiei celor două thread-uri anterioare înainte de a deveni inactiv.

Observaţi ordinea în care se pornesc thread-urile şi ordinea în care se execută.

```
1  public class DemoPriority {
2     public static void main(String[] args) {
3        System.out.println("begin main thread...");
4        PriorityThread threadMin = new PriorityThread(Thread.MIN_PRIORITY);
5        PriorityThread threadMax = new PriorityThread(Thread.MAX_PRIORITY);
6        threadMin.thread.start();
7        threadMax.thread.start();
8        try{
9           threadMax.thread.join();
10          threadMin.thread.join();
11       }catch (InterruptedException e) {
12          System.out.println("interrupted exception");
13       }
14       System.out.println("end main thread!!!");
15    }
16 }
```

E. Sincronizarea firelor de execuţie

Clasa *MySynchronized* are un constructor cu doi parametri. Unul este de tip String şi reprezintă numele thread-ului, iar cel de-al doilea este un număr întreg. Constructorul este cel care instanţiază thread-ul şi tot el este cel care îl porneşte.

Observaţi cele două metode care folosesc modificatorul *synchronized*. Prima metodă îl foloseşte ca modificator al funcţiei, iar ce-a de-a doua metodă, cea între comentarii, îl foloseşte ca şi bloc. Important de reţinut este că în varianta bloc *synchronized* se aplică doar pe obiecte.

```
1  public class MySynchronized implements Runnable{
2     private int count;
3     private Thread thread;
4
5     public MySynchronized(String name, int count){
6        this.count  = count;
7        thread = new Thread(this, name);
8        thread.start();
9     }
10
11    public synchronized int add(){
12       count++;
13       try{
14          Thread.sleep(1000);
15       }catch (InterruptedException e) {
16          System.out.println("main thread interrupted");
17       }
18       ++count;
19       return count;
20    }
21
22    /*public int add(){
23       synchronized (thread) {
24          count++;
25          try{
26             Thread.sleep(1000);
27          }catch (InterruptedException e) {
28             System.out.println("main thread interrupted");
29          }
30          ++count;
31          return count;
32       }
33    }*/
34
35    public void run(){
36       System.out.println(thread.getName() + " starting.");
37       System.out.println("\t the numer of "+thread.getName()+" is: "+add());
38       System.out.println(thread.getName() + " terminating.");
39    }
40 }
```

Clasa *DemoSynchronized* creează două obiecte de tipul *MySynchronized*.

```
1  public class DemoSynchronized {
```

```
2     public static void main(String[] args) {
3        System.out.println("begin main thread...");
4        MySynchronized thread1 = new MySynchronized("thread1",1);
5        MySynchronized thread2MySyncronized = new MySynchronized("thread2",2);
6        System.out.println("begin main thread...");
7     }
8  }
```

III. TEMĂ

1. Realizaţi un program care creează cel puţin două fire de execuţie ce calculează produsul elementelor aceluiaşi vector. Elementele vectorului se vor citi din fişierul *in.txt*, iar rezultatul se va afişa într-un alt fişier, *out.txt*. Cerinţe de implementare:

   - pentru calculul produsului folosiţi *synchronized* (ori modificatorul de acces ori ca bloc);
   - pentru implementarea thread-urilor puteţi folosi oricare dintre cele două metode prezentate în laborator;
   - atât pentru lucrul cu fişiere cât şi pentru lucrul cu thread-uri se vor folosi mecanisme de tratare a excepţiilor.