Programare concurentă în Java.

Java multithreading.

Obiective:
- înțelegerea conceptelor legate de firele de execuție;
- implementarea de aplicații cu fire de execuție;

Objectives:
- understanding the concepts related to execution threads;
- embedding threads into specific applications;

Fire de execuție

Programarea concurentă sau *multithreading*-ul este un mecanism asemănator (considerat de tip *lightweigt*) şi reprezintă capacitatea unui program de a executa mai multe secvenţe de cod din acel program în acelaşi timp. O astfel de secvenţă de cod se numeşte *fir de execuţie* sau *thread*.

Datorită posibilităţii creării mai multor thread-uri, un program Java poate să execute mai multe sarcini simultan (sau concurent), cum ar fi animaţia unei imagini, transmiterea unei melodii la placa de sunet, comunicarea cu un server, etc.

Pentru ca programatorul Java să poată dezvolta programe multithread de bază, limbajul Java oferă în pachetul implicit *java.lang*, două clase şi o interfaţă Java:

- interfaţa *java.lang.Runnable*
- clasa *java.lang.Thread*
- clasa *java.lang.ThreadGroup*

Clasa *Thread* şi interfaţa *Runnable* oferă suport pentru lucrul cu thread-uri ca entităţi separate, iar clasa *ThreadGroup* permite crearea unor grupuri de thread-uri în vederea tratării acestora într-un mod unitar.

Suportul pentrun mecanismul bazat pe executori este inclus in pachetul *java.util.concurrent*.

Mecanismul de programare concurentă este nativ în limbajul de programare Java. Metoda *main()* este metoda apelată de Java Runtime Environment (JRE) în momentul în care se execută o aplicaţie Java de-sine-stătătoare. Codul inclus în această metodă este inclus intr-un un fir de execuție implicit care rulează atâta timp cât metoda *main()* nu a ajuns încă la final. Orice aplicație Java, oricât de simplistă ar fi, solicită din partea sistemului rularea a cel puțin 1 fir de execuție. JVM rulează atâta timp cât cel puțin unul dintre toate procesele asociate este în stare de rulare. Execuția unui program se termină când sau se apelează metoda *exit()* a clasei *System* sau când ultimul proces asociat iese din starea de rulare.

Controlul programatic al firelor de execuție se bazează pe implementarea unei metode speciale numite *run()* care este apelată de către JVM cu scopul de a rula instrucțiunile specifice menționate de către programator.

Interfaţa *java.lang.Runnable*

O modalitate de creare a unui thread este de a utiliza interfaţa *Runnable*.
Interfaţa *Runnable* declară o singură metodă, şi anume:

```
public void run();
```

Orice clasă care foloseşte (implementează) această interfaţă trebuie să definească implementarea metodei mai sus

amintite, definind asfel comportamentul firului de execuţie.

Clasa *java.lang.Thread*

Definirea unei clase proprii care moşteneşte clasa *Thread* reprezintă o altă modalitate de a crea un fir de execuţie. Vor fi prezentate în continuare metodele membre ale clasei *Thread*.

Metode constructor

- *Thread()*

=> crează un nou obiect de tip thread

- *Thread(Runnable target)*

=> crează un nou obiect de tip thread

- *Thread(Runnable target, String name)*

=> crează un nou obiect de tip thread, cu numele primit ca parametru

- *Thread(String name)*

=> crează un nou obiect de tip thread, cu numele primit ca parametru

- *Thread(ThreadGroup group, Runnable target)*

=> crează un nou obiect de tip thread şi îl asociază grupului primit ca parametru

- *Thread(ThreadGroup group, Runnable target, String name)*

=> crează un nou obiect de tip thread, cu numele primit ca parametru şi îl asociază grupului primit ca parametru

- *Thread(ThreadGroup group, Runnable target, String name, long stackSize)*

=> crează un nou obiect de tip thread, cu numele primit ca parametru, îl asociază grupului primit ca parametru şi are mărimea stivei primită ca parametru

- Thread(ThreadGroup group, String name)

=> crează un nou obiect de tip thread, cu numele primit ca parametru şi îl asociază grupului primit ca parametru

Metode membre principale

- *static int activeCount()*

=> returnează numărul de thread-uri active din grupul curent de thread-uri

- *void checkAccess()*

=> determina dacă firul curent care rulează are permisiunea să modifice acest thread

- *static Thread currentThread()*

=> returnează o referinţă la thread-ul curent

- *void destroy()*

=> distruge necondiţionat acest thread

- *static void dumpStack()*

=> tipăreşte stiva thread-ului curent

- *static int enumerate(Thread[] tarray)*

=> copiază în şirul de thread-uri primit ca parametru toate firele de execuţie active din grupul curent de thread-uri şi din subgrupurile acestuia

- *ClassLoader getContextClassLoader()*

=> returnează *ClassLoader*-ul firului de execuţie curent

- *String getName()*

=> returnează numele firului de execuţie curent

- *int getPriority()*

=> returnează prioritatea thread-ului curent

- *ThreadGroup getThreadGroup()*

=> returnează grupul de care aparţine thread-ul curent

- *static boolean holdsLock(Object obj)*

=> returnează *true* dacă thread-ul curent are cheia obiectului specificat

- *void interrupt()*

=> întrerupe firul curent de execuţie

- *static boolean interrupted()*

=> testează dacă firul curent a fost întrerupt

- *boolean isAlive()*

=> testează dacă firul curent este activ

- *boolean isDaemon()*

=> testează dacă firul curent este de tip *daemon*

- *boolean isInterrupted()*

=> testează dacă firul curent a fost întrerupt

- *void join()*

=> aşteaptă ca firul curent să fie terminat

- *void join(long millis)*

=> aşteaptă cel mult numărul de milisecunde specificat pentru a termina firul de execuţie curent

- *void join(long millis, int nanos)*

=> aşteaptă cel mult numărul de milisecunde şi nanosecunde specificat pentru a termina firul de execuţie curent

- *void resume()*

=> metoda depreciata, reia execuţia firului de execuţie curent

- *void run()*

=> dacă firul de execuţie a fost construit folosind un obiect *Runnable*, se apelează metoda *run()* a acelui obiect. În caz contrar, nu face nimic.

- *void setContextClassLoader(ClassLoader cl)*

=> setează *ClassLoader* -ul firului de execuţie curent

- *void setDaemon(boolean on)*

=> marchează firul de execuţie curent ca fiind *daemon* sau de tip utilizator

- *void setName(String name)*

=> setează numele firului de execuţie curent

- *void setPriority(int newPriority)*

=> setează prioritatea firului de execuţie curent

- *static void sleep(long millis)*

=> întrerupe funcţionarea firului curent de execuţie pentru minim numărul de milisecunde menţionat

- *static void sleep(long millis, int nanos)*

=> întrerupe funcţionarea firului curent de execuţie pentru minim numărul de milisecunde + nanosecunde menţionat

- *void start()*

=> porneşte firul de execuţie curent. Aceasta metodă apelează metoda *run()* a firului de execuţie curent

- *String toString()*

=> returnează reprezentarea textuala a thread-ului curent (se includ informaţii despre nume, prioritate, grup de care aparţine

- *static void yield()*

=> forţează thread-ul curent să cedeze temporar accesul la procesor pentru a permite şi altor fire de execuţie să ruleze

Clasa *ThreadGroup*

Această clasă este folosită pentru a permite gruparea şi tratarea unitara a mai multor fire de execuţie. Vor fi prezentate în continuare metodele membre ale clasei *ThreadGroup*.

Metode constructor

- *ThreadGroup(String name)*

=> crează un nou grup de fire de execuţie, cu numele primit ca parametru

- *ThreadGroup(ThreadGroup parent, String name)*

=> crează un nou sub-grup de fire de execuţie, având ca părinte grupul primit ca parametru; numele este de asemenea primit ca parametru

Metode membre principale

- *int activeCount()*

=> returnează o estimare a numărului de fire de execuţie active din grupul curent

- *int activeGroupCount()*

=> returnează o estimare a numărului de sub-grupuri active din grupul curent

- *void checkAccess()*

=> determina dacă firul curent de execuţie are permisiunea de a modifica acest grup de thread-uri

- *void destroy()*

=> distruge grupul de thread-uri şi toate eventualele sub-grupuri

- *int enumerate(Thread[] list)*

=> copiază în şirul de thread-uri primit ca parametru toate firele de execuţie active din grupul curent şi din eventualele sub-grupuri

- *enumerate(ThreadGroup[] list)*

=> copiază în şirul de grupuri de thread-uri primit ca parametru toate sub-grupurile active ale grupului curent

- *int getMaxPriority()*

=> returnează prioritatea maximă a firelor de execuţie conţinute de grupul curent

- *String getName()*

=> returnează numele grupului de fire de execuţie

- *ThreadGroup getParent()*

=> returnează părintele grupului de fire de execuţie curent

- *void interrupt()*

=> întrerupe toate firele de execuţie din grupul curent

- *boolean isDaemon()*

=> testează dacă grupul curent de fire de execuţie este de tip *daemon*

- *boolean isDestroyed()*

=> testează dacă grupul curent de fire de execuţie este distrus

- *void list()*

=> tipăreşte (la consola) informaţii legate de grupul curent de fire de executieboolean

- *parentOf(ThreadGroup g)*

=> testează dacă grupul curent de fire de execuţie este părintele grupului primit ca parametru

- *String toString()*

=> returnează informaţia textuala care descrie grupul de fire de execuţie curent

- *void uncaughtException(Thread t, Throwable e)*

=> este o metodă apelata de JVM când un thread din grupul curent de fire de execuţie se opreşte din cauza unei excepţii ne-semnalizate.

Crearea unui fir de execuţie utilizând interfaţa *Runnable*

Operaţiile de creare a unui fir de execuţie (*thread*) utilizând interfaţa *Runnable* sunt:

- se crează o clasă care implementează interfaţa *Runnable*.

```
class MyRunnable implements Runnable{
        //implementarea clasei
}
```

- clasa care implementează interfaţa *Runnable* trebuie să definească metoda *public void run()* cu codul care se doreşte a fi executat:

```
public void run(){
        //implementarea codului specific  firului de execuţie
}
```

- se instanțiază un obiect al clasei create utilizând operatorul *new*:

```
MyRunnable myRunnable=new MyRunnable();
```

- se crează un obiect din clasa *Thread* utilizând un constructor ce are ca şi parametru un obiect de tip *Runnable*. Astfel se asociază un thread cu o metodă *run()*:

```
Thread thread=new Thread(myRunnable);
```

- în final se porneşte thread-ul, prin apelarea metodei *start()*:

```
thread.start();
```

Crearea unui fir de execuţie pornind de la clasa *Thread*

În cazul utilizării clasei *Thread*, avem următoarele operaţii de îndeplinit:

- se crează o clasă derivată din clasa *java.lang.Thread*:

```
class MyThread extends Thread{
        //i mplementarea clasei
}
```

- se suprascrie (override) metoda *public void run()* moştenită din clasa *Thread* în clasa derivată. Aceasta metodă trebuie să implementeze codul specific al firului de execuție.

```
public void run(){
        //implementarea comportamentului specific al firului de execuţie
}
```

- se instanțiază clasa definită:

```
Thread thread=new MyThread();
```

- se porneşte thread-ului instanțiat, prin apelul metodei *start()* moştenite de la clasa *Thread*. Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program necesar unui thread după care să apeleze metoda *public void run()* în mod automat.

```
thread.start();
```

Stările firelor de execuţie

Un fir de execuţie se poate afla la un moment dat într-una din următoarele stări:

- *new* (nou creat)
- *running* (în execuţie, starea spre care aspira toate firele de execuţie)
- *blocked* (waiting, starea de aşteptare, adormire, suspendare, blocare)

- *ready* (gata de execuţie, prezent în coada de aşteptare)
- *dead* (terminat)

Un fir de execuţie aflat în starea *dead* nu mai poate fi repornit. Această încercare este privită ca o eroare de execuţie lansând excepţia:

java.lang.IllegalThreadStateException

După ce un fir de execuţie este terminat, el continua să existe în continuare ca un obiect oarecare Java, metodele din clasa să putând fi în apelate prin intermediul obiectului *dead*. Singurul atribut care se pierde prin terminare este legat de comportamentul de fir de execuţie.

Prioritatea firelor de execuţie

Execuția de thread-uri multiple pe un singur CPU, e numită *scheduling* (planificare). Java suportă un mecanism foarte simplu deterministic de planificare bazat pe priorități fixe. Algoritmul de planificare al thread-urilor e bazat pe prioritatea relativă față de alte thread-urile în execuție (runnable).
Java defineşte trei constante simbolice pentru selectarea priorităţilor firelor de execuţie:

- *public final static int MAX_PRIORITY; //10*
- *public final static int MIN_PRIORITY; //1*
- *public final static int NORM_PRIORITY; //5*

Sincronizarea firelor de execuție

- posibilitatea accesării simultane de către mai multe fire de execuție a unor resurse (variabile, metode)
- dacă cel mult un thread are acces la o secvență de cod la un anumit moment <=> *excludere mutuală*
- procese de tip *producer - consumer*

Monitor = un obiect care asigură că o variabilă partajată poate fi accesată într-un moment dat de cel mult un fir de execuţie (lock, deadlock)

- utilizat în sincronizare
- *wait()*: starea de rulare => blocare pe termen nedeterminat
- *notify(), notifyAll() =>* reactivarea firelor blocate care așteaptă după un monitor

Executori

- *Executor*: interfață simplă care permite rularea de procese asociate firelor de execuție
- *ExecutorService*: (derivată din *Executor*) adaugă facilități de gestionare a ciclului de viață a proceselor și a executorilor
- *ScheduledExecutorService*: (derivată din *ExecutorService*) permite în plus executarea viitoare sau periodică a proceselor

Lucru individual

1. Să se creeze o aplicație Java în cadrul căreia există o clasă ce implementează interfața Runnable. Constructorul clasei permite definirea unui nume asociat fiecărui obiect instanțiat din clasa respectivă și de asemenea clasa are un atribut static ce contorizează numărul de obiecte instanțiate. Metoda *run()* a clasei va afișa numele obiectului de un număr de ori egal cu valoarea contorului și cu o întârziere de 1000msec între afișări.

Dintr-o clasă separată, creați mai multe fire de execuție cu obiecte diferite din clasa descrisă anterior și analizați rezultatele afișate.

==========

2. Creați o aplicație Java ce folosește accesul sincronizat la metode pentru excludere mutuală. Creați 3 fire de execuție ce apelează simultan metode de incrementare și decrementare a unei variabile dintr-o altă clasă. Verificați dacă rezultatele sunt cele așteptate. Eliminați blocurile sincronizate și re-evaluați rezultatele.

==========

3. Scrieți o aplicație Java ce calculează cel mai mare divizor comun a două numere mari (>100.000). Aplicația va citi de la tastatură în mod continuu perechi de câte două numere și va lansa în execuție câte un thread pentru fiecare pereche citită. Rezultatele vor fi afișate în consolă în momentul finalizării metodei de calcul a cmmmdc.

==========

4. Se dă o urnă ce conține un singur tichet câștigător și alte 1000 de tichete necâștigătoare. În jurul urnei se află un număr de N persoane, N – citit de la tastatură, care extrag bilete din această urnă în mod concomitent, iar biletele nu sunt introduse înapoi în urnă. În momentul în care una dintre persoane extrage biletul câștigător, jocul se oprește. Alternative: 1) persoanele extrag bilete secvențial; 2) biletele sunt introduse înapoi în urnă.

==========

5. Desenați o grilă de dimensiune 3x3 ce conține 9 numere distincte așezate aleator în cele 9 căsuțe. Utilizatorul trebuie să apese pe aceste numere în ordine crescătoare după ce a dat click pe un buton de start. Afișati pe ecran un contor de timp de la momentul în care utilizatorul început jocul. Apăsarea unui număr greșit duce la restartarea jocului. În caz de câștig, afișati un mesaj corespunzător și timpul necesar câștigării jocului

==========

6. Creați o aplicație Java ce afișează pe ecran cercuri de diferite raze și în poziții aleatoare. Cercurile vor fi afișate după un anumit interval de timp aleator (nu mai mult de 10 secunde). Utilizatorul trebuie să dea click cu mouse-ul pe aceste cercuri – moment în care ele vor dispărea de pe suprafața de joc. Odată ce un cerc a dispărut de pe ecran, un nou cerc își va porni contorul până la afișare. Atenție la suprapunerile dintre cercuri.

Individual work

1. Write a Java application which contains a class which implements the Runnable interface. The class' constructor sets the name of the instantiated object. Also, there is a class variable which counts the number of instantiated objects from that class. The *run()* method of the class will print out the object's name for a number of times equal to the counter's value and delayed by 1000 msec.

From a separate class, create multiple threads build from separate objects of the previously described class. Analyze the results.

==========

2. Write a Java app which uses the synchronized method acces for mutual exclusion. Create 3 separate threads which simultaneously call methods to increment and decrement a separate class' class variable. Check if the results are what you expect them to be. Remove the synchronized blocks and reevaluate the results.

==========

3. Write a Java app which computes the greates common divisor for large numbers (>100.000). The app will continuosly read from the keyboard pairs of numbers and launch threads for each of the pairs. The results will be displayed in the console as soon as they are available.

==========

4. There is an urn which contains 1000 losing tickets and 1 golden winning ticket. There are N people around the urn (N is read from the keyboard) which simultaneously extract tickets from it. The tickets are not placed back into the urn. When one person extracts the golden ticket, the game stops.

Alternatives: 1) the persons extract the tickets in a predefined order; 2) the tickets are placed back into the urn.

==========

5. Draw a board of 3x3 squares each containing a random number placed randomly in one of the 9 squares. The user must click on each square in ascending order of the numbers. The game starts when the user presses the START button. The screen also displays a time counter which start when the user starts the game. The game is restarted if the user presses a wrong square. If the user wins the game display a congratulatory message along with the time needed to win the game.

==========

6. Write a Java app which displays circles of random radius and at random positions. The circles will be displayed after a random time interval (no longer than 10 seconds). The user must click on these circles and make them disappear from the screen. When a circle is clicked, a new one will start its countdown-to-appearance timer. Pay attention to overlapping areas.