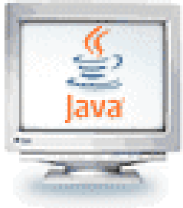
Programare orientată pe obiecte

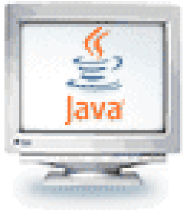
1. Fire de lucru (*Threads*) în Java

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA
Computer Science



Utilitatea firelor de lucru

- Acolo unde trebuie să se întâmple mai multe lucruri simultan
  - D.e., o aplicaţie multimedia poate necesita ca procesele audio, video şi de control să se execute în paralel
  - Există adesea perioade de aşteptare a răspunsului sistemelor de I/E mai lente, timp în care procesorul poate face altceva
- Programe cum sunt sistemele server/client sunt mult mai uşor de proiectat şi scris folosind fire de lucru
- Algoritmi matematici, cum sunt sortarea, căutarea numerelor prime etc. utilizând prelucrarea paralelă
- Pe sistemele multi-procesor, maşinile virtuale Java pot rula firele de lucru pe procesoare diferite şi pot obţine astfel prelucrare paralelă adevărată şi creşteri de performanţe semnificative faţă de platformele mono-procesor



Multithreading în Java

- Toate programele Java în afara aplicaţiilor simple cu intrare-ieşire pe consolă sunt aplicaţii multithreading
- Procesele grele (***heavyweight***) se rulează direct sub sistemul de operare al maşinii locale și pot conține mai multe subprocese
- Procesele ușoare (***lightweight***) conțin un singur proces (curs secvențial) care este lansat din procesul principal, dar cu fire de lucru (**threaduri**) multiple
- Threadurile sunt fire de lucru paralele care rulează înăuntrul unui program
  - Ele partajează o zonă de memorie comună
- ***Multithreading*** în Java se referă la un program care execută simultan mai multe threaduri

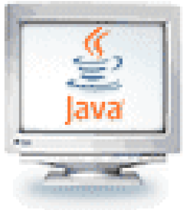
Proprietăţile firelor de lucru în Java

- Fiecare fir de lucru îşi începe execuţia la o locaţie bine cunoscută, predefinită
- Fiecare fir de lucru îşi execută codul începând de la locaţia de start, într-o secvenţă ordonată, predefinită (pentru un set de date de intrare dat)
- Fiecare fir de lucru îşi execută codul independent de celelalte fire de lucru din program
- Firele de lucru par a avea un anumit grad de simultaneitate în execuţie
- Firele de lucru au acces la diferite tipuri de date

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

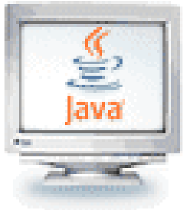
Crearea unui Thread

- Un fir de lucru (*thread*) în Java începe prin crearea unei instanțe a clasei **java.lang.Thread**
- Metodele din clasa **Thread** pentru manipularea firelor de execuție sunt, de exemplu:
  - **start()**
  - **yield()**
  - **sleep()**
  - **run()**
- Acțiunea firului de execuție începe la invocarea metodei **run()**
- La apelul metodei **run()** se crează o nouă stivă de apel pentru threadul executat
  - În Java, fiecare thread are propria stivă de apeluri



Crearea unui Thread

- Definirea și instanțierea unui thread poate fi făcută în unul din cele două feluri:
  - Extinderea clasei **java.lang.Thread**
  - Implementarea interfeței **Runnable**
- Singurul motiv pentru care are sens să se extindă clasa Thread este cazul când se doreste realizarea unei versiuni mai specializate a clasei **Thread**
  - Atunci când dorește un comportament specializat al firului de execuție
- În restul cazurilor (majoritatea cazurilor) când se dorește doar să se specifice ce anume trebuie să execute threadul, se definește o clasă care implementează interfața **Runnable**

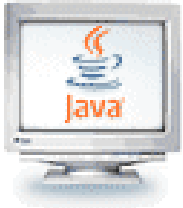


Definirea unui Thread

- Prin extinderea clasei java.lang.Thread
  - Modul cel mai simplu de definire a codului de rulat într-un thread separat este:
    - Extinderea clasei Thread
    - Suprascrierea metodei run()
  - Exemplu:

```
class MyThread extends Thread {
      public void run() {
        System.out.println("Important job running in MyThread");
      }
}
```

  - Limitarea acestei abordări este că noua clasă nu va mai putea extinde vreo altă clasă din moment ce a extins clasa Thread

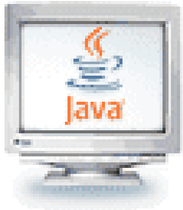


Definirea unui Thread

- Prin implementarea interfeței java.lang.Runnable
  - Această variantă oferă flexibilitatea de a extinde orice altă clasă, păstrându-și proprietatea de a putea fi executată într-un fir de lucru separat
  - Exemplu:

```
class MyRunnable implements Runnable {
    public void run() {
      System.out.println("Important job running in MyRunnable");
    }
}
```

- Indiferent de modalitatea aleasă, rezultă cod ce poate fi executat într-un fir de lucru separat



Instanțierea unui Thread

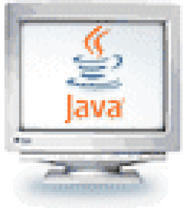
- Fiecare thread începe prin instanțierea unui obiect de clasă Thread
  - Indiferent cum s-a implementat metoda run(), prin extinderea clasei Thread sau prin implementarea interfetei Runnable, este nevoie de un obiect de tipul Thread care să facă treaba
- Pentru varianta când s-a extins clasa Thread, instanțierea este simplă:

```
MyTread t = new MyThread();
```

- Pentru varianta când s-a implementat interfața Runnable, este nevoie de următorii pași:

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
```

  - În acest caz, obiectul Runnable este dat ca și argument al constructorului Thread pentru a ști unde se află implementarea metodei run()



Instanțierea unui Thread

- Același obiect Runnable poate fi pasat ca argument la mai multe threaduri, de exemplu:

```
public class TestThreads {
    public static void main (String [] args) {
        MyRunnable r = new MyRunnable();
        Thread foo = new Thread(r);
        Thread bar = new Thread(r);
        Thread bat = new Thread(r);
    }
}
```

  - Făcând astfel, mai multe threaduri vor rezolva simultan aceeași problemă
- Până aici avem o instanță thread și știm care metodă run() se va executa, dar nu avem încă o execuție
- Pentru execuție e nevoie de invocarea metodei start()

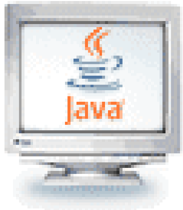
Începerea unui thread

- Pentru începerea unui thread se apelează:

  t.start();

  - În acest moment se creează și o nouă stivă de apeluri asociată threadului t
  - Doar din momentul apelului metodei start() threadul este considerat în viață (*alive*), chiar dacă metoda run() este posibil să nu se fi executat încă
  - Un thread este considerat mort (*no longer alive*) după ce metoda run() și-a terminat execuția
  - Pentru a verifica starea unui thread (dacă metoda run() și-a terminat sau nu execuția) se poate folosi metoda:

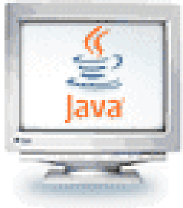
    t.isAlive()



Începerea unui thread

- Ce se întâmplă concret la apelul metodei start():
  - Un nou thread începe, având o stivă nouă de apeluri (fiecare thread are stiva proprie de apeluri)
  - Starea thread-ului se schimbă de la **new** la **runnable**
  - Când thread-ul prinde ocazia să se execute, metoda run() se va rula
- Exemplu:

```
class FooRunnable implements Runnable {
    public void run() {
        for(int x = 0; x < 3; x++)
            System.out.println("Runnable running");
    }
}
```

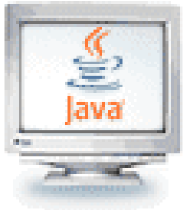


Începerea unui thread

```
public class TestThreads {
    public static void main (String [] args) {
            FooRunnable r = new FooRunnable();
            Thread t = new Thread(r);
            t.start();
    }
}
```

Rezultate afișate:

Runnable running
Runnable running
Runnable running

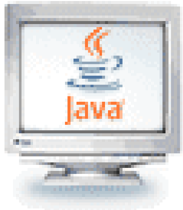


Începerea unui thread

- Pentru a ști care thread se execută, putem interoga apelând metoda getName() din clasa Thread
- În următorul exemplu vom da nume threadului și îl vom afișa în metoda run() :

```
class NameRunnable implements Runnable {
        public void run() {
            System.out.println("NameRunnable running");
            System.out.println("Run by "+ Thread.currentThread().getName());
        }
}
public class NameThread {
        public static void main (String [] args) {
            NameRunnable nr = new NameRunnable();
            Thread t = new Thread(nr);
            t.setName("Fred");
            t.start();
        }
}
```

Rezultate afișate:
NameRunnable running
Run by Fred



Începerea unui thread

- Procesul de începere a unui thread:

```
public static void main(String [] args) {
   //  running
   //  some code
   //  in main()
   method2();
   //  running
   //  more code
}

static void method2() {
   Runnable r = new MyRunnable();
   Thread t = new Thread(r);
   t.start();
   //  do more stuff
}
```

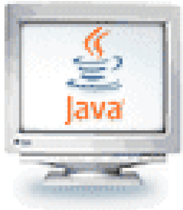
main
stack A
1) main() begins

method2
main
stack A
2) main() invokes method2()

run
stack B
(thread t)

method2
main
stack A
(main thread)

3) method2() starts a new thread



Rularea mai multor threaduri

- În continuare este prezentat un exemplu cu o singură interfață Runnable și trei fire de lucru

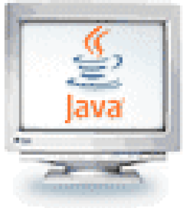
```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 0; x <3; x++) {
            System.out.println("Run by "+ Thread.currentThread().getName() + ", x is " + x);
        }
    }
}
public class ManyNames {
    public static void main(String [] args) {
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);
        one.setName("Fred");  two.setName("Lucy");  three.setName("Ricky");
        one.start(); two.start(); three.start();
    }
}
```

<u>Rezultate afișate</u>:

Run by Lucy, x is 0
Run by Fred, x is 0
Run by Ricky, x is 0
Run by Fred, x is 1
Run by Fred, x is 2
Run by Lucy, x is 1
Run by Ricky, x is 1
Run by Lucy, x is 2
Run by Ricky, x is 2

Rularea mai multor threaduri

- Atenție: aceste rezultate au fost obținute la rularea pe o anumită mașină, **dar această ordine de execuție nu este garantată!**
  - Nu există nici o garanție în specificațiile Java că threadurile își vor începe execuția în ordinea în care a fost apelată metoda `start()`
  - Nu există garanția că o dată ce un thread și-a început execuția, o va continua până la final
  - Și nici garanția că o buclă se va termina de executat înainte ca un alt thread să înceapă
- Singura garanție este următoarea:
  - Fiecare thread va începe execuția și fiecare thread o va finaliza



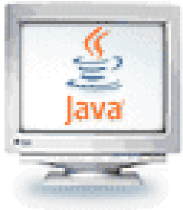
Rularea mai multor threaduri

- În interiorul unui thread lucrurile se petrec într-o ordine predictibilă, dar acțiunile threadurilor multiple pot fi amestecate într-o ordine neprevăzută
- Dacă același cod se rulează de mai multe ori, sau pe mașini diferite, rezultatul poate să fie diferit
- De exemplu, dacă modificăm numărul de iterații la 400:

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 0; x <400; x++) {
            System.out.println("Run by "+
            Thread.currentThread().getName() + ", x is " + x);
        }
    }
}
```

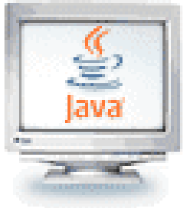
Rezultate afișate:

...
Run by Fred, x is 345
Run by Ricky, x is 313
Run by Lucy, x is 341
Run by Ricky, x is 314
Run by Lucy, x is 342
Run by Ricky, x is 315
Run by Fred, x is 346
Run by Lucy, x is 343
Run by Fred, x is 347
...



Ordinea execuției threadurilor

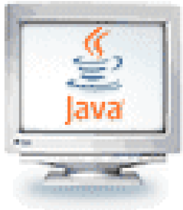
- Planificatorul ordinii de execuție a threadurilor (*Thread Scheduler*) ține de JVM
  - Decide care thread să se execute la un anumit moment de timp (din mulțimea threadurilor eligibile)
  - Scoate threadurile din starea de ***running*** la terminarea execuției lor
- O singură stivă de apeluri poate fi executată la un moment dat pe o unitate de procesare
- Există un comportament de coadă în planificarea ordinii de execuție, în sensul că o dată ce un thread și-a terminat bucata lui de rulat, este pus în capătul cozii unde își așteaptă din nou rândul pentru execuție
- Chiar dacă nu putem controla ordinea de execuție, există totuși unele unelte pentru a influența această ordine



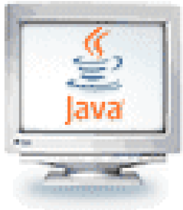
Ordinea execuției threadurilor

Câteva metode care ajută la influențarea ordinii de execuție a threadurilor sunt:

- Din clasa **java.lang.Thread** :
  - public static void sleep(long millis) throws InterruptedException
  - public static void yield()
  - public final void join() throws InterruptedException
  - public final void setPriority(int newPriority)
- Din clasa **java.lang.Object**:
  - public final void wait() throws InterruptedException
  - public final void notify()
  - public final void notifyAll()

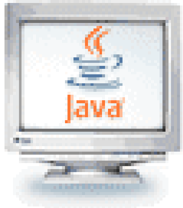


Ciclul de viaţă al unui fir de lucru



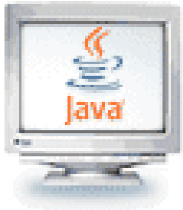
Starea threadurilor

- Stările pe care un thread le poate avea sunt:
  - new – instanța de thread a fost creată, dar încă nu a fost apelată metoda start()
  - runnable – în momentul apelării metodei start(), threadul intră în această stare în care este eligibil pentru execuție
  - running – un thread intră în starea de running când planificatorul ordinii de execuție alege acest thread pt execuție; acesta este momentul în care metoda run() a threadul se execută
  - waiting/blocked/sleeping – instanța este în viață (alive) dar nu e runnable; el poate reveni la starea de runnable mai târziu, dacă, de exemplu un anumit eveniment este interceptat (I/E, etc.)
  - dead – când metoda run() a threadului si-a terminat execuția
- Responsabilitatea planificatorului de threaduri este de a schimba starea threadului



Thread.sleep()

- Thread.sleep() este o metodă statică din clasa Thread care pune în pauză un fir de lucru din interiorul căruia se face invocarea metodei
  - Pune în pauză threadul pentru un timp egal cu numărul de milisecunde dat ca argument
  - Remarcaţi că metoda poate fi invocată dintr-un program obişnuit pentru a insera o pauză în singurul fir al programului respectiv
    - Pentru o aplicație obișnuită (single-threaded), metoda `run()` din thread corespunde metodei `main()`
- Metoda sleep() aruncă o excepţie de tipul InterruptedException atunci când un fir "adormit" este întrerupt
  - Interceptează excepţia
  - Termină firul de lucru



Thread.sleep()

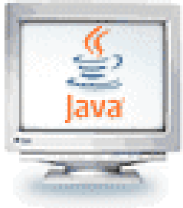
- Prevenirea execuției unui thread folosind metoda **`Thread.sleep()`**
  - Exemplu modificat unde forțăm alternarea execuției threadurilor

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 0; x < 3; x++) {
            System.out.println("Run by "
                            + Thread.currentThread().getName());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) { }
        }
    }
}
```

Rezultate afișate:

Run by Ricky
Run by Lucy
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Ricky
Run by Fred
Run by Lucy

- De reținut că ieșirea nu este repetitivă, pentru că nu se știe cu certitudine cât timp durează rularea thread-ului până când este pus în starea de ***sleep***

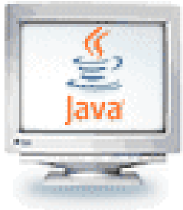


Terminarea firelor de lucru

- Un fir de lucru se termină la terminarea metodei sale **run()**
- Notificaţi firul de lucru că ar trebui să își înceteze execuția folosind

```
t.interrupt();
```

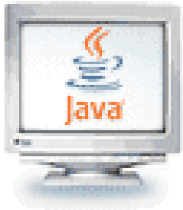
- **interrupt()** nu face ca firul de lucru să se termine – metoda doar setează un câmp boolean în structura de date a lui
- Java nu forţează terminarea unui fir atunci când acesta este întrerupt
- Este treaba firului de lucru ce anume face atunci când este întrerupt
- Întreruperea reprezintă un mecanism general pentru a "obţine atenţia" firului de lucru întrerupt



Terminarea firelor de lucru

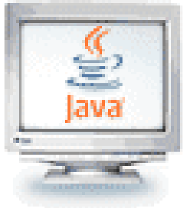
- Metoda run() ar trebui să verifice ocazional dacă firul de lucru a fost întrerupt
  - Folosiţi metoda interrupted()
  - Un fir de lucru care a fost întrerupt ar trebui să elibereze resursele pe care le foloseşte, să "cureţe" şi să îşi înceteze execuția

```
public void run()
{
   for(int i=1; i<= REPETITIONS && !Thread.interrupted(); i++)
      {
         //Do work
      }
   //Clean up
}
```



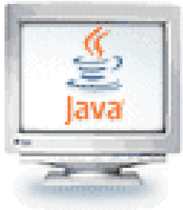
Prioritățile threadurilor și yield()

- Pentru a ințelege cum funcționează metoda yield() trebuie să înțelegem mai intâi conceptul de prioritate a threadurilor
- Threadurile se execută într-o anumită ordine (cu o anumită prioritate)
- Prioritatea este de obicei reprezentată printr-un număr de la 1 (minimă) la 10 (maximă)
- Planificatorul de threaduri se bazează în cazul majorității mașinilor virtuale pe o planificare bazată pe priorități, lucru ce implică un fel de secționare temporală (*time slicing*)
  - Prin folosirea secționării temporale, fiecărui thread i se alocă un anumit timp pentru execuție, după care este trimis în starea de *runnable* pentru a da loc altui thread să se execute



Prioritățile threadurilor și yield()

- Nu toate specificațiile JVM cer implementarea secționării temporale
  - Deși multe JVM folosesc secționarea temporală, unele folosesc un planificator de threaduri ce permite unui thread să execute în întregime metoda **run()**
- Nu vă bazați pe prioritătile threadurilor când proiectați o aplicație multithreading, deoarece comportamentul de planificare bazat pe priorități nu este garantat
  - Ex.: dacă un thread intră în starea *runnable*, dar are prioritate mai mare decât threadul curent care rulează, atunci threadul curent este trimis în starea *runnable* și thread-ul cu prioritatea cea mai mare este ales să ruleze
- Folositi prioritățile threadurilor ca o modalitate de a îmbunătăți eficiența programului



Setarea priorității unui thread

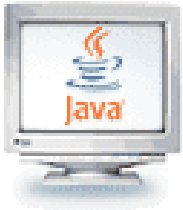
- Un thread primește implicit o prioritate în momentul în care este creat
  - Ex.: MyThread t = new MyThread(); // prioritatea implicita are valoarea 5
  - Deoarece threadul **main** se execută în momentul în care este instanțiat threadul **t**, acesta va avea aceeași prioritate ca **main**
- Setarea manuală a priorității unui thread se face folosind metoda setPriority(…) astfel:

```
MyThread t = new MyThread();
t.setPriority(8);
t.start();
```

  - Valorile priorităților sunt de obicei între 1 (minimă) și 10 (maximă)
  - Pentru a verifica exact intervalul permis de JVM folosiți constantele: Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, Thread.MAX_PRIORITY

Metoda statică Thread.yield()

- Metoda yield() are rolul de a duce threadul curent din stadiul de *running* în *runnable* pentru a permite threadurilor cu **aceeași prioritate** să ruleze și ele
- În realitate însă, metoda yield() nu garantează acest lucru
  - Chiar dacă yield() cauzează trecerea threadului curent în starea runnable, se întâmplă adesea ca același thread să fie ales spre execuție

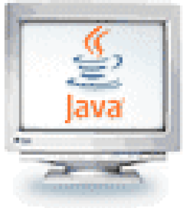


Metoda non-statică Thread.join()

- Metoda join() permite unui thread să se poziționeze la sfârșitul altui thread ("*join onto the end*")
- Dacă un thread B nu poate să își facă treaba decât după ce threadul A a fost executat, atunci vrem să se facă "join" între threadurile A și B
  - Acest lucru înseamnă că threadul B nu devine *runnable* decât după ce threadul A a fost executat
- Ex.:

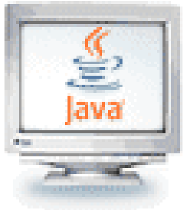
```
Thread t = new Thread();
t.start();
t.join();   // sau t.join(500);
```

  - Acest cod ia threadul în lucru (în acest caz - main) și face "join" cu threadul t
  - Metoda join() cu parametru se interpretează astfel: threadul principal așteaptă până când t este finalizat, dar dacă durează mai mult de 500 ms, acesta devine din nou *runnable*



Sincronizarea codului

- Sincronizarea metodelor previne accesarea simultană a codului dintr-o metodă de mai multe threaduri
  - Ce s-ar întâmpla dacă două threaduri ar încerca simultan să seteze starea unui obiect?
- Cuvântul cheie **syncronized** poate fi folosit
  - Ca modificator al metodei
  - La începutul unui bloc de cod
- În timp ce un singur thread are dreptul de a accesa codul sincronizat al unei instanțe, restul codului (nesincronizat) poate fi accesat simultan de mai multe threaduri
- Când un thread intră în starea *sleep* codul sincronizat blocat de el nu va fi accesibil altor threaduri

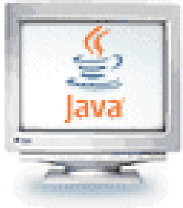


Sincronizarea codului. Exemplu

- Un cont bancar cu două persoane împuternicite să aibă acces la acest cont

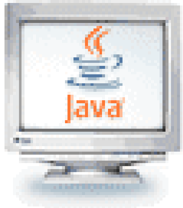
```
class Account {
   private int balance = 50;
   public int getBalance() {
        return balance;
   }
   public void withdraw(int amount) {
        balance = balance - amount;
   }
}
```

- Din cont se pot efectua operații de retragere de numerar. Acestea sunt limitate la suma exactă de 10. Astfel, pașii de efectuat ar fi:
  - Verifică bilanțul
  - Dacă sunt suficienți bani (ex. minim 10), efectuează retragerea



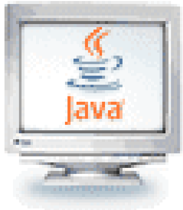
Sincronizarea codului. Exemplu

- Ce se întâmplă dacă intervine ceva între acești pași?
  - Ex.: Cei doi utilizatori vor să retragă bani în același timp, pentru amândoi bilanțul arată că este credit suficient, primul retrage suma, dar când al doilea încearcă să retragă, nu mai are bani suficienți în cont!
- Logica ce ar trebui urmată pentru implementarea problemei:
  1. Obiectul runnable să țină o referință către un singur cont
  2. Se pornesc două threaduri reprezentând acțiunile celor două persoane, și ambele threaduri fac referire la același obiect runnable
  3. Bilanțul inițial e de 50 și suma de scos e exact 10
  4. În metoda run() avem un ciclu ce se repetă de 5 ori. În fiecare buclă
     - Efectuăm retragerea (doar în cazul în care există destul credit)
     - Afișăm un mesaj dacă contul este 'corupt' (lucru care nu ar trebui să se întâmple niciodată, pentru că verificăm de fiecare bilanțul contului)



Sincronizarea codului. Exemplu

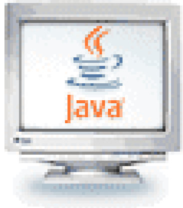
5. Metoda makeWithdrawal() într-o clasă de test ar trebui să efectueze următoarele:
   - Verifică bilanțul pentru a verifica dacă este suficient credit pentru o retragere
   - Dacă este suficient, afișează numele persoanei care operează contul
   - Pune threadul pe *sleep* 500ms, simulând timpul necesar pentru o retragere
   - La trezire, efectuează retragerea și afișează un mesaj în acest sens
   - Dacă nu au fost bani suficienți, afișează un mesaj care să atenționeze asupra acestui lucru



Sincronizarea codului. Exemplu

```
public class AccountDanger implements Runnable {
  private Account acct = new Account();
  public static void main (String [] args) {
    AccountDanger r = new AccountDanger();
    Thread one = new Thread(r);
    Thread two = new Thread(r);
    one.setName("Fred");
    two.setName("Lucy");
    one.start();
    two.start();
  }
  public void run() {
    for (int x = 0; x < 5; x++) {
      makeWithdrawal(10);
      if (acct.getBalance() < 0) {
      System.out.println("account is overdrawn!");
}}}
```

```
private void makeWithdrawal(int amt) {
if (acct.getBalance() >= amt) {
    System.out.println(Thread.currentThread().
    getName()+ " is going to withdraw");
    try {
      Thread.sleep(500);
    } catch(InterruptedException ex) { }
    acct.withdraw(amt);
    System.out.println(Thread.currentThread().
    getName()+ " completes the withdrawal");
} else {
    System.out.println("Not enough in account for "
    + Thread.currentThread().getName()
    + " to withdraw " + acct.getBalance());
}}}
```



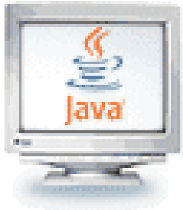
Sincronizarea codului. Exemplu

- Rezultate afișate:

1. Fred is going to withdraw
2. Lucy is going to withdraw
3. Fred completes the withdrawal
4. Fred is going to withdraw
5. Lucy completes the withdrawal
6. Lucy is going to withdraw
7. Fred completes the withdrawal
8. Fred is going to withdraw
9. Lucy completes the withdrawal
10. Lucy is going to withdraw
11. Fred completes the withdrawal
12. Not enough in account for Fred to withdraw 0
13. Not enough in account for Fred to withdraw 0
14. Lucy completes the withdrawal
15. account is overdrawn!
16. Not enough in account for Lucy to withdraw -10
17. account is overdrawn!
18. Not enough in account for Lucy to withdraw -10
19. account is overdrawn!

Soluție pentru evitarea coruperii contului:
Să ne sigurarăm că pașii

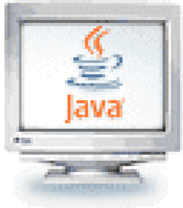
1. de verificare a bilanțului și
2. de retragere se efectuează împreună (= operațe atomică)



Sincronizarea codului. Exemplu

- Nu se poate garanta că un singur thread va rula până la finalizarea operației atomice, dar putem garanta că, chiar dacă threadul își mai schimbă starea din modul running, nici un alt thread nu va putea să opereze cu aceste date
  - Cu alte cuvinte, chiar dacă threadul lui Lucy intră în modul sleep după verificarea bilanțului, putem să îi interzicem lui Fred să verifice și el bilanțul până când Lucy finalizează operația de retragere
- Cum protejăm datele?
  - Facem variabilele *private*
  - Sincronizăm codul care modifică variabilele
- Putem rezolva problema punând modificatorul syncronized metodei makeWithdrawal() astfel:

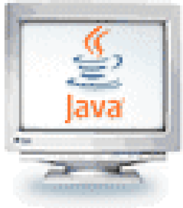
```
private synchronized void makeWithdrawal(int amt) {
    //same code
}
```



Sincronizarea codului. Exemplu

- Noile rezultate afișate:

Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0



Sincronizare și blocare (*lock*)

- Sincronizarea funcționează cu lacăte (*locks*)
- Fiecare obiect Java are încorporat un singur lock care intervine doar atunci când obiectul deține metode sincronizate
- Doar metode (sau blocuri) pot fi sincronizate, nu și variabilele sau clasele
- Nu toate metodele necesită să fie sincronizate
- O clasă poate avea atât metode sincronizate, cât și nesincronizate
- Dacă două metode încearcă să execute o metodă sincronizată, și ambele folosesc aceeași instanță, threadurile vor putea să execute doar pe rând această metodă

Interacțiunea dintre threaduri

- Ultimul lucru de știut despre threaduri este cum interacționează între ele pentru a-și comunica diferite aspecte (ex. *lock status*)
- Metodele responsabile pentru comunicarea între threaduri sunt: **wait(), notify(), notifyAll()** din clasa **Object**
- Un exemplu ar fi o aplicație de mail:
  - Aplicația are două threaduri: unul responsabil cu trimiterea mailurilor (T1), iar celălalt cu procesarea lor (T2)
  - T2 trebuie să verifice frecvent dacă dacă există vreun mail de procesat
  - Folosind ***mecanismul wait-notify***, T2 poate verifica existența mailurilor de procesat, și dacă nu gasește nici unul, poate spune astfel: "nu o să-mi irosesc timpul verificând în continuu, ies să fac altceva, iar când T1 (mail delivery) aduce un mail, îmi va trimite o notificare să știu să trec în starea *runnable* și să îmi fac treaba"

Comunicarea cu obiectele cu metodele wait() și notify()

- Metoda **wait()** lasă threadul să spună: nu este nimic de lucru pentru mine asa ca pune-mă în starea *waiting*, sau adaugă-mă în lista de așteptare
- Metoda **notify()** este folosită pentru a transmite un semnal către un thread din lista de așteptare a obiectului
  - Nu se poate specifica pe care *waiting* thread să îl notifice
- Metoda **notifyAll()** trimite semnal tuturor threadurilor în starea *waiting*
- Metodele **wait(), notify(), notifyAll()** trebuie să fie apelate dintr-un cod sincronizat!
  - Un thread invocă una din aceste metode pe un obiect anume și threadul trebuie să dețină lock-ul acelui obiect

Exemplu: Comunicarea cu obiectele cu metodele wait() și notify()

```
1. class ThreadA {
2.  public static void main(String [] args) {
3.     ThreadB b = new ThreadB();
4.     b.start();
5.
6.     synchronized(b) {
7.       try {
8.         System.out.println("Waiting for b to
                              complete...");
9.         b.wait();
10.      } catch (InterruptedException e) {}
11.      System.out.println("Total is: " +
                              b.total);
12.  }
13. }
14.}
15.
16. class ThreadB extends Thread {
17.   int total;
18.
19.   public void run() {
20.       synchronized(this) {
21.           for(int i=0; i<100; i++) {
22.               total += i;
23.           }
24.           notify();
25.       }
26.   }
27. }
```

Exemplu: Comunicarea cu obiectele cu metodele wait() și notify()

- Explicații:
  - Programul conține două threaduri:
    - ThreadA – threadul principal
    - ThreadB – threadul care calculează suma numerelor de la 0 la 99
  - Linia 4: la apelul metodei start(), ThreadA continuă cu linia următoare de cod din clasa lui, ceea ce înseamnă că ar putea ajunge la linia 11 înainte ca ThreadB să finalizeze de calculat suma. Pentru a preveni acest lucru folosim metoda wait() la linia 9
  - La linia 6 codul este sincronizat cu obiectul b deoarece, pentru a putea apela metoda wait() pentru obiectul b, ThreadA trebuie să dețină lock-ul obiectului b
    - Pentru ca un thread să poată apela metodele wait() sau notify(), acesta trebuie să dețină lock-ul obiectului
    - Când un thread așteaptă (este în starea *wait*), el cedează pentru moment lock-ul obiectului în favoarea altui thread

Exemplu: Comunicarea cu obiectele cu metodele wait() și notify()

- Remarcați la liniile 7-10 folosirea mecanismului try-catch în jurul metodei wait()
  - Un thread poate fi întrerupt prin apelul metodei wait() care aruncă o excepție. Astfel, excepția aruncată trebuie prinsă și tratată

```
try {
   wait();
} catch(InterruptedException e) {
    // Do something about it
}
```

  - Odată intrat în starea waiting, threadul așteaptă până când operatorul trimite prima notificare, moment în care intră din nou în posesia lock-ului și își poate continua execuția
- Rezultatele afișate de programul din exemplu sunt:

  Waiting for b to complete...

  Total is: 4950

Animaţii cu threaduri

- Animaţiile sunt o sarcină uzuală pentru firele de lucru
- Firele de lucru
  - Pot efectua sarcini diferite în paralel
  - Sunt folosite la controlul animaţiei
- Exemplu:
  - Un fir de lucru: realizează desenarea fiecărui cadru
  - Alt fir de lucru: tratează interacţiunile cu utilizatorul

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science