

Metode avansate de programare

Informatică Româna, Curs 9



Referinte pe care se bazeaza acest curs

- Oracle tutorials
- <http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>
- <http://www.javacodegeeks.com/2015/09/java-concurrency-essentials.html>
- <http://tutorials.jenkov.com/java-util-concurrent/executorservice.html>
- <http://stacktips.com/tutorials/java/countdownlatch-and-java-concurrency-example>
- <http://blogs.msmvps.com/peterritchie/2007/04/26/thread-sleep-is-a-sign-of-a-poorly-designed-program/>

Programarea concurentă și paralelă

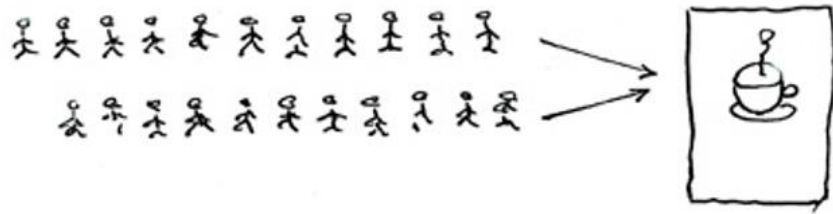
▪ Concurrent =

▪ Paralel =

Programarea concurentă și paralelă

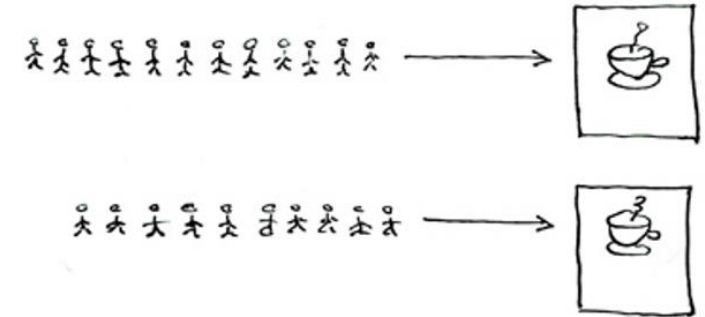
- **Concurent** = 2 cozi și un aparat de cafea

Concurrent = Two Queues One Coffee Machine



- **Paralel** = 2 cozi și două aparate de cafea

Parallel = Two Queues Two Coffee Machines



- este posibilă și la sistemele simple, care nu au *multConcurentaiprocessor* sau *mai multe nuclee de executie*

Utilitatea concurenței

- Simplificarea programării și o mai bună structurare a codului
- O aplicație poate să execute mai multe **taskuri (unitati logice)** *deodată*
- Eficiență în **utilizarea resurselor sistemului** (modelul de comunicare multithread înlocuiește soluțiile clasice de comunicare între procese)
- Eficiență în **execuția operațiilor consumatoare de timp** - pentru a nu bloca procesul principal.
- Îmbunătățirea interacțiunii la nivel de aplicație prin **proiectarea de interfețe performante, responsive** (*Interfata cu utilizatorul sa nu “inghețe” la execuția unor taskuri care durează mult (citirea datelor, descărcarea unui fisier) –responsive GUI.*)

Procese și fire de execuție

- In programarea concurentă exista două *unități de execuție* de bază:
procese (processes) si **fire de executie (threads)**
 - Ambele concepte implică execuția în “parallel” a unor secvențe de cod
 - **Procese**le sunt entități independente ce se execută independent și sunt gestionate de către nucleul sistemului de operare.
 - **Firele de execuție** sunt secvențe ale unui program (proces) ce se execută *aparent* în paralel în cadrul unui singur proces.

Fire de execuție – Threads

- Un **fir de execuție** este o succesiune secventiala de instructiuni care se executa *aparent* în paralel în cadrul unui process (un mecanism care poate executa task-urile asincron).
- Sunt denumite “procese usoare” (**lightweight processes**)
- Crearea unui fir de executie necesita mai putine resurse decat crearea unui process
- Un fir de execuție există în cadrul unui process - fiecare process are cel puțin un fir de execuție
- **Firele de executie partajeaza resursele procesului**, incluzând datele, câstigând eficiență în felul acesta, dar comunicarea între acestea devine problematică

Fire de execuție – Threads

- Toate firele de execuție văd același **heap**. Java alocă toate obiectele în heap, deci toate firele au acces la toate obiectele.
- **Fiecare fir are propria sa stivă**. Variabilele locale și parametrii unei metode se alocă în stivă, deci fiecare fir are propriile valori pentru variabilele locale și pentru parametrii metodelor pe care le execută.
- O funcționare **corectă** a unui program concurent (“thread safe”) nu are voie să se bazeze pe presupuneri de genul:



“stiu eu că obiectul acesta nu e accesat niciodată simultan de două fire de execuție diferite”.

“stiu eu că firul ăsta e mai rapid ca celălalt”

Costul programării multithreading

- Programul devine supraincarcat (**overheaded**) crescând foarte mult **complexitatea** acestuia.
- Costul:
 - **creării** si **distrugerii** firelor de executie.
 - **planificarii** firelor de executie, a **încarcarii** acestora, **stocarea statusurilor** dupa fiecare cuanta de timp.
- Costul daca toate threadurile sunt în acelasi process, acest aspect adauga o **complexitate** sporită codului pentru a ne asigura că **accesul la zona de date nu runinează aplicatia**.
- **Depanarea** unui program ce ruleaza pe mai multe fire de executie este foarte grea;
reproducerea acelorasi rezultate planificate este dificila...

Clasele programării multithread in Java. Clasa Object.

- Metodele clasei Object, *wait()*, *notify()* și *notifyAll()*, sunt utilizate în **programarea multithread**, având următoarea semnificație:
 - *wait()* - pune obiectul în așteptare până la apariția unui eveniment (notificare) cu sau fără indicarea duratei maxime de așteptare
 - *notify()* - permite anunțarea altor obiecte de apariția unui eveniment
 - *notifyAll()* - implementează notificarea mai multor obiecte la apariția unor evenimente

Clasele programării java multithread. Clasa Thread.

```
public class Thread extends Object implements Runnable {}
```

Principalele câmpuri și metode ale clasei Thread sunt:

- *void start()* - lansează în execuție noul thread, moment în care execuția programului este controlată de cel puțin două threaduri: threadul curent ce execută metoda start și noul thread ale cărui instrucțiuni sunt definite în metoda run()
- *void run()* – definește corpul threadului nou creat, întreaga activitate a threadului va fi descrisă prin suprascrierea acestei metode
- *static void sleep()* – pune în așteptare threadul curent pentru un anumit interval de timp (msecs)
- *void join ()* - se așteaptă ca obiectul thread ce apelează această metodă să se termine
- *suspend()* - suspendare temporară a threadului (*resume()* este metoda duală ce relansează un thread suspendat (implementările JDK ulterioare versiunii 1.2 au renunțat la utilizarea lor)

Clasele programării multithread in Java. Clasa Thread.

```
public class Thread extends Object implements Runnable {}
```

Continuare - Principalele câmpuri și metode ale clasei Thread sunt:

- *yield()* - realizează cedarea controlului de la obiectul thread, planificatorului JVM pentru a permite unui alt thread să ruleze
- *void interrupt()* – trimite o întrerupere obiectului thread ce o invocă (setează un flag de întrerupere a threadului activ).
- *static boolean interrupted()* - testează dacă threadul curent a fost întrerupt, resetează starea interrupted a threadului current
- *boolean isInterrupted ()* - testează dacă un thread a fost întrerupt fără a modifica starea threadului
- *boolean isAlive()* - permite identificarea stării obiectului thread
- *void setDaemon(boolean on)* - apelată imediat înainte de start permite definirea threadului ca daemon. Un thread este numit **daemon**, dacă metoda lui **run** conține un ciclu infinit, astfel încat acesta nu se va termina la terminarea threadului părinte.
- *getPriority()* - returnează prioritatea threadului curent
- *setPriority(newPriority)* -permite atribuirea pentru threadul curent a unei priorități dintr-un interval.

Metodele stop(), suspend() și resume(), definite în versiuni anterioare au fost eliminate deoarece în cazul unei proiectări defectuoase a codului pot provoca blocarea acestuia .

Crearea firelor de execuție (thread) in Java

- In orice program Java, aflat in executie, JVM creează automat **un obiect fir de executie**, avand rolul de a apela metoda *main*.
- Firele de executie pot fi create și explicit de către programator:
 - **Implement the Runnable Interface (preferred)**
 - **Extend the Thread class**

Metoda 1. `java.lang.Thread`

```
class Ball extends Thread {  
    // constructor; draw(); move(); etc.  
    public void run() {  
        // code to animate the ball  
    }  
}  
  
public class TestBall{  
    public static void main(String[] args) {  
        Ball b=new Ball();  
        b.start();  
    }  
}
```

Mod gresit de folosire a
mostenirii: a ball “is a” thread???

Ball b = new Ball(...);

- ▶ **To launch a runnable thread, call start()**
- ▶ NEVER call run() directly
 - ▶ start() creates the thread, sets up the context, & calls run()

Metoda 2. Interfata funcțională **Runnable**

```
class Ball implements Runnable {  
    private Pane box;  
  
    public Ball(Pane b) {  
        box = b;  
    }  
  
    public void run() {  
        // code to animate the ball  
    }  
}
```

```
Pane p=new Pane();  
Ball b=new Ball(p);  
Thread t=new Thread(b);  
t.start();
```

Metoda 2. Interfata funcțională **Runnable**

```
class Ball {  
    private Pane box;  
  
    public Ball(Pane b) {  
        box = b;  
    }  
}
```

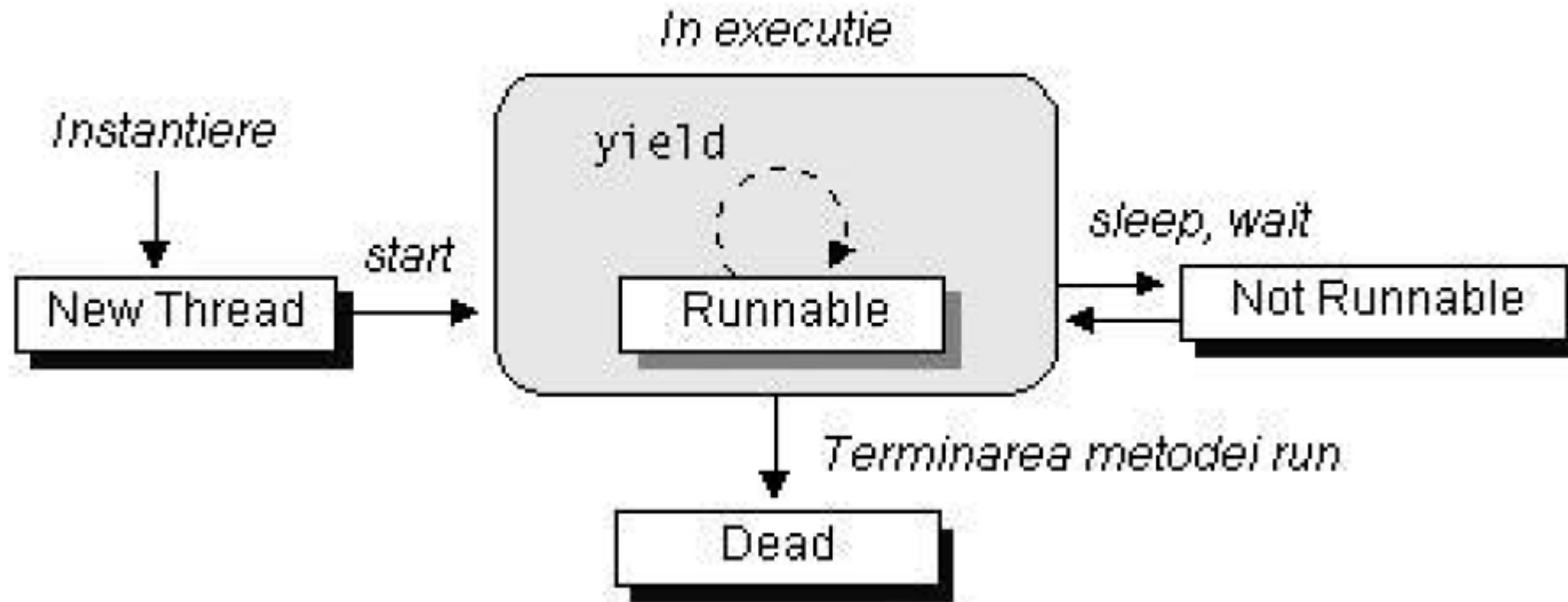
```
Ball b=new Ball(p);  
Thread t=new Thread(new Runnable() {  
    @Override  
    public void run() {  
        // ...  
    }  
});  
t.start();
```

```
Thread t2=new Thread(()->{ //...run method as Lambda })  
t2.start();
```


Stările unui fir de execuție

- **New Thread** – Obiectul fir de execuție a fost creat: `Thread counterThread = new Thread (...);`
- **Runnable** – După apelul metodei `start`, `counterThread.start();` Metoda `start` realizează următoarele operațiuni necesare rulării firului de execuție:
 - *aloca resursele sistem necesare*
 - *planifica firul de execuție la CPU pentru a fi lansat*
 - *apelează metoda `run` a obiectului reprezentat de firul de execuție*
- **Dead** – Calea normală prin care un fir se termina este prin ieșirea din metoda `run()`. Se poate forța terminarea firului apelând metoda `stop()` dar nu se recomandă folosirea sa, fiind o metoda “deprecated” în Java2.
- **Not Runnable - Blocked/Wait** – Un fir de execuție ajunge în această stare în una din următoarele situații:
 - este "adormit" prin apelul metodei `sleep`.
 - a apelat metoda `wait`, așteptând ca o anumită condiție să fie satisfăcută
 - este blocat într-o operație de intrare/ieșire

Ciclul de viață a unui thread



- ▶ `public class Thread { .. // enum in 1.5 is a special class for finite type.`
- ▶ `public static enum State { //use Thread.State for referring to this nested class`

Cooperarea firelor de executie (synchronization)

- O caracteristică importantă a firelor de executie este că ele vad același **heap**.
- Acest lucru poate duce la **multe probleme** în cazul în care un obiect oarecare (care e o resursa comuna pentru toate firele) este accesat din doua fire de executie diferite.
- **Metode de cooperare:**
 - **Mecanismul de excludere mutual (mutex – semafor)**
 - **Comunicare prin conditii**



Imaginați-vă la un ghiseu mai multe persoane care vor să obțină informații de la o singură persoană sau doi indieni care trag cu același arc!!!!

Problema producatorului - consumatorului

```
class BankAccount {  
    private double balance; //sold  
    public BankAccount(double bal) { balance = bal; }  
    public BankAccount() { this(0); }  
    public double getBalance() { return balance; }  
    public void deposit(double amt) { ... }  
    public void withdraw(double amt) { ... }  
}
```

```
BankAccount account =new BankAccount(0);
```

```
Thread t1=new Thread(()->account.deposit(50));
```

```
Thread t2=new Thread(()->account.deposit(50));
```

```
t1.start();
```

```
t2.start();
```

Problema producatorului - consumatorului

```
public void deposit(double amt) {  
    double temp = getBalance();  
    temp = temp + amt;  
    try {  
        Thread.sleep(300);  
    } catch (InterruptedException ie) {  
        System.err.println(ie.getMessage());  
    }  
    balance = temp;  
}
```

```
public void deposit(double amt) {  
    double temp = getBalance();  
    temp = temp + amt;  
    try {  
        Thread.sleep(300);  
    } catch (InterruptedException ie) {  
        System.err.println(ie.getMessage());  
    }  
    balance = temp;  
}
```

```
BankAccount account = new BankAccount(0);  
Thread t1 = new Thread(() -> account.deposit(50));  
Thread t2 = new Thread(() -> account.deposit(50));  
t1.start();  
t2.start();
```

```
t1.join();  
t2.join();  
System.out.println("after deposit balance = $" + account.getBalance());
```

In cazul exemplului de mai sus se spune ca programatorul a introdus in sistem o **conditie de cursa**. Astfel de conditii nu au voie sa apara in sistemele concurente.

Output:
after deposit balance = \$50.0
Process finished with exit code 0

Mecanismul de excludere mutuală

- **Soluție – mutex - monitor:**
 - la un moment dat un obiect poate fi manipulat de un singur fir de execuție și numai de unul.
 - Dacă un fir de execuție apelează o **metodă sincronizată** pentru un obiect se verifică dacă obiectul respectiv se află în starea “liber”.
 - Dacă da, obiectul e trecut în starea “ocupat” și firul începe să execute metoda, iar când firul termină execuția metodei obiectul revine în starea “liber”.
 - Dacă nu e “liber” când s-a efectuat apelul, înseamnă că există un alt fir ce execută o metodă sincronizată pentru același obiect (mai exact, un alt fir a trecut obiectul în starea “ocupat” apelând o metodă sincronizată sau utilizând un bloc de sincronizare). Într-o astfel de situație firul va aștepta până când obiectul trece în starea “liber”.

Excludere mutuală

- Cum specificam acest lucru in Java? Una dintre posibilitati ar putea fi:
 - metode **synchronized**: in timpul in care un thread executa instructiunile unei metode synchronized pentru un obiect, nici un alt thread nu poate executa o metoda declarata synchronized pentru ACELASI obiect.
 - utilizarea blocurilor de sincronizare



Synchronized account Exemplu

- <http://www.cs.sjsu.edu/~pearce/modules/lectures/j2se/multithreading/synch1.htm>

```
public synchronized void deposit(double amt) {
```

```
    double temp = balance;
```

```
    temp = temp + amt;
```

```
    try {
```

```
        Thread.sleep(300); // simulate production time
```

```
    } catch (InterruptedException ie) {
```

```
        System.err.println(ie.getMessage());
```

```
    }
```

```
    balance = temp;
```

```
}
```

```
BankAccount account =new BankAccount(0);
```

```
Thread t1=new Thread()->account.deposit(50));
```

```
Thread t2=new Thread()->account.deposit(50));
```

```
t1.start();
```

```
t2.start();
```


Impas (deadlock)

- *Excluderea mutuala* este necesara pentru rezolvarea unuei probleme de concurrenta, dar *nu si suficienta*:
- Un fir executa o metoda sincronizata (deci obiectul apelat e “ocupat”) **dar nu poate sa termine executia pana cand nu s-a indeplinit o anumita conditie.**
- Daca acea conditie poate fi indeplinita **doar cand un alt fir ar apela o metoda sincronizata a aceluiasi obiect**, situatia ar fi fara iesire:
 - Obiectul e tinut ocupat, iar firul care vrea sa apeleze metoda sincronizata a aceluiasi obiect pentru indeplinirea conditiei (deposit method), nu poate acest lucru (obiectul fiind “ocupat” iar metoda sincronizata).

Excluderea mutuala nu e suficienta

```
public class BankAccount {  
    public synchronized void deposit(double amt) {...}  
    public synchronized void withdraw(double amt) {...}  
}
```

```
class Consumer implements Runnable {  
    private BankAccount account;  
    private double amount;  
    public Consumer(BankAccount acct, double amt) { account = acct; amount=amt;}  
    public void run() {  
        account.withdraw(amount);  
    }  
}
```

```
class Producer implements Runnable {  
    private BankAccount account;  
    private double amount;  
    public Producer(BankAccount acct, double amt) { account = acct; amount=amt;}  
    public void run() {  
        account.deposit(amount);  
    }  
}
```

Excluderea mutuala nu e suficientă

```
BankAccount account = new BankAccount(50);
int slaveCount = 4;
Thread[] slaves = new Thread[slaveCount];
for(int i = 0; i < slaveCount; i++) {
    if (i == 2) {
        slaves[i] = new Thread(new Producer(account,50));
    } else {
        slaves[i] = new Thread(new Consumer(account,50));
    }
}
for(int i = 0; i < slaveCount; i++) {
    slaves[i].start();
}

for(int i = 0; i < slaveCount; i++) {
    try {
        slaves[i].join();
    } catch (InterruptedException ie) {
        System.err.println(ie.getMessage());
    } finally {
        System.out.println("slave " + i + " has died");
    }
}
System.out.print("Closing balance = ");
System.out.println("$" + account.getBalance());

    public void synchronized withdraw(double amt) {
        while (balance < amt) {
            System.out.println("Insufficient funds! Waiting");
            return;
        }
        double temp = balance;
        temp = temp - amt;
        try {
            Thread.sleep(200); // simulate consumption time
        } catch (InterruptedException ie) {
            System.err.println(ie.getMessage());
        }
        System.out.println("after withdrawl balance = $" + temp);
        balance = temp;
    }
}
```

wait - notifyall

```
public synchronized void deposit(double amt) {
    double temp = balance; temp = temp + amt;
    try {
        Thread.sleep(300); // simulate production time
    } catch (InterruptedException ie) { System.err.println(ie.getMessage()); }
    balance = temp;
    System.out.println("after deposit balance = $" + balance);
    notifyAll();
}
```

```
public synchronized void withdraw(double amt) {
    while (balance < amt) {
        System.out.println("Insufficient funds! waiting ... ");
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    double temp = balance; temp = temp - amt;
    try {
        Thread.sleep(200); // simulate consumption time
    } catch (InterruptedException ie) {
        System.err.println(ie.getMessage());
    }
    balance = temp; System.out.println("after withdrawl balance = $" + balance);
}
```

Comunicare prin conditii

Prototip	Descriere
<code>wait()</code>	Când un fir de execuție apelează această metodă pentru un obiect, firul va fi pus în așteptare. Aceasta înseamnă că nu se revine din apel, că ceva ține firul în interiorul metodei <code>wait()</code> . Acest apel este utilizat pentru “blocarea” unui fir până la îndeplinirea condiției de continuare. În timp ce un fir așteaptă în metoda <code>wait()</code> apelată pentru un obiect, obiectul receptor va trece temporar în starea “liber”.
<code>notifyAll()</code>	Când un fir de execuție apelează această metodă pentru un obiect, TOATE firele de execuție ce sunt “blocate” în acel moment în metoda <code>wait()</code> a ACELUIAȘI OBIECT sunt deblocate și pot reveni din apelul respectivei metode. Acest apel este utilizat pentru a anunța TOATE firele care așteaptă îndeplinirea condiției de continuare că această condiție a fost satisfăcută.

METODE UTILIZATE ÎN MECANISMUL DE COOPERARE PRIN CONDIȚII.