

10 . MULTI-THREADING

10.1. Concepte.....	<u>2</u>
10.1.1. Procese vs thread-uri.....	<u>2</u>
10.1.2. De ce este necesara executia concurenta.....	<u>2</u>
10.1.3. Modalitati de lucru cu fire de executie in Java.....	<u>3</u>
10.2. Utilizarea claselor de nivel inalt.....	<u>3</u>
10.2.1. Clasa javax.swing.Timer.....	<u>3</u>
10.2.2. Clasa java.util.Timer.....	<u>4</u>
10.3. Utilizarea claselor low-level.....	<u>5</u>
10.3.1. Functionarea multi-threading-ului in Java.....	<u>5</u>
10.3.1.1. Concepte.....	<u>5</u>
10.3.1.2. Starile posibile ale unui thread.....	<u>5</u>
10.3.2. Lucrul cu obiecte Thread.....	<u>6</u>
10.3.2.1. Crearea unui obiect Thread.....	<u>6</u>
10.3.2.2. Pornirea si rularea thread-urilor.....	<u>7</u>
10.3.2.3. Prioritatile thread-urilor.....	<u>8</u>
10.3.2.4. Plasarea voluntara a unui thread in asteptare.....	<u>9</u>
10.3.2.4.1. Situatii.....	<u>9</u>
10.3.2.4.2. Renuntarea la statutul de thread activ: metodele sleep() si yield().....	<u>9</u>
10.3.2.4.3. Asteptarea incheierii executiei unui alt thread: metodele join().....	<u>10</u>
10.3.2.5. Intreruperea unui thread.....	<u>11</u>
10.3.2.6. Alte facilitati si metode utile ale clasei Thread.....	<u>12</u>
10.4. Accesul concurrent la date comune.....	<u>13</u>
10.4.1. Identificarea problemei.....	<u>13</u>
10.4.2. Sincronizarea thread-urilor.....	<u>14</u>
10.4.3. Sincronizarea in cazul metodelor statice.....	<u>15</u>
10.4.4. Semnalizarea inter-thread-uri: asteptarea unui eveniment.....	<u>16</u>
10.5. Pachetul java.util.concurrent.....	<u>19</u>
10.5.1. Prezentare. Principii.....	<u>19</u>
10.5.2. Executia task-urilor folosind obiecte de tip Executor.....	<u>19</u>
10.6. Clase si metode thread-safe.....	<u>21</u>
10.7. BIBLIOGRAFIE.....	<u>21</u>

10.1. Concepte

10.1.1. Procese vs thread-uri

In sistemul de operare, un proces reprezinta o instanta a unui program care ruleaza si care are atasat un anumit context de executie (prioritate, spatiu de memorie folosit etc). Un sistem de operare multitasking este unul in care pot fi rulate simultan mai multe procese, dand iluzia paralelismului acestora prin comutarea fiecarui procesor de la un proces la altul, suficient de repede incat perceptia umana este aceea a executiei simultane a tuturor aplicatiilor care ruleaza.

In cadrul aceluiasi proces pot exista mai multe fire de executie (intotdeauna cel putin unul) care reprezinta portiuni de cod ale procesului care ruleaza concurrent – un fel de multi-tasking dar intern, in cadrul procesului. Cititorul se intreabă probabil de ce exista aceasta dubla ramificare - o aplicatie poate porni mai multe procese, iar un proces poate porni la randul sau mai multe thread-uri; aparent cea de-a doua ramificare este redundanta. Motivul este ca intre procese si thread-uri exista diferente importante:

- **din punct de vedere al usurintei colaborarii intre procese/thread-uri:**
 - procesele lucreaza in contexte de executie diferite - fiecare avand propriul sau spatiu de memorie ce-i contine datele. Din acest punct de vedere, rolul de arbitru al sistemului de operare il face pe acesta sa se asigure in primul rand de o buna izolare a proceselor unul de altul, si abia apoi de mecanisme prin care ele sa poata colabora. Un proces care nu se “comporta civilizat” trebuie sa compromita cat mai putin sau deloc functionarea celorlalte. Modalitatile de comunicare inter-proces, desi exista, nu sunt atat de firesti precum colaborarea intre fire de executie (nu exista, intr-un sistem de operare, conceptul de a accesa o variabila a unui proces - fie si pentru faptul ca variabilele sunt isi au rostul doar in etapa de design a unei aplicatii; odata compilata aplicatia, ele se materializeaza in zone de memorie si numele lor se pierd)
 - firele de executie pot lucra usor cu date comune, deoarece impart acelasi spatiu de memorie (cel al procesului de care apartin). Threadurile unui proces sunt parte a aceleiasi aplicatii si acceseaza aceleiasi variabile, ceea ce face colaborarea intre thread-uri facila si fireasca
- **din punct de vedere al costului de comutare a executiei de la un proces/thread la altul.** Executia concurrenta - atat pentru procese cat si pentru thread-uri - presupune comutarea procesorului de la un proces/thread la altul intr-o succesiune rapida. Aceasta implica salvarea starii procesului/thread-ului curent si incarcarea starii in care ramasese procesul/thread-ul urmator. Operatia - denumita *comutare de context* - este mult mai costisitoare pentru procese decat pentru thread-uri (de altfel, threadurile mai sunt denumite ocazional *lightweight processes*)

10.1.2. De ce este necesara executia concurrenta

Sa consideram cazul unui program cu un singur fir de executie. In cadrul sau toate instructiunile ruleaza secvential, fiecare instructiune fiind nevoita sa astepte incheierea executiei celei precedente. Acest lucru nu este convenabil in dese cazuri, deoarece aplicatiile moderne nu restrictioneaza utilizatorul la o singura operatie. Spre exemplu, in browser putem porni un download in timp ce navigam pe internet; in plus, la incarcarea unei pagini web sunt incarcate in paralel resursele prezente in aceasta (imagini, aplicatii Flash etc). Daca nu ar exista paralelism, odata ce pornim un download ar trebui sa-i asteptam incheierea, timp in care nicio alta operatie nu mai este posibila.

Firele de executie sunt un element cheie mai ales in cazul interfetelor grafice. Dupa cum se cunoaste, toata desenarea/redesenarea unui GUI este bazata pe evenimente; daca toata aplicatia ar rula intr-un singur thread, ar inseman ca, in timp ce se trateaza un eveniment, restul aplicatiei stagneaza - nu se vor mai redesena componente, utilizatorul nu va mai putea da click-uri sau efectua orice alta operatie (interfata grafica nu mai raspunde la comenzi) pana cand tratarea de eveniment se incheie. Acesta este motivul pentru care toate evenimentele ruleaza intr-un fir de executie separat – asa-numitul *event dispatching thread*. Chiar si asa insa, daca un tratarea unui eveniment dureaza prea mult, ea poate tine in loc restul de evenimente din dispatching thread si din nou interfata grafica nu mai raspunde!

Nota: in scopul rularii de cod in event dispatching thread (de exemplu, pentru a actualiza un progress bar din cadrul unei metode care ruleaza in alt thread) exista doua metode statice ale clasei *SwingUtilities*:

SwingUtilities.invokeLater(Runnable) si ***SwingUtilities.invokeLater(Runnable)***.

In concluzie, folosim thread-uri atunci dorim sa "detasam" o parte a programului nostru de restul aplicatiei, facand-o sa se execute in paralel (sau cel putin cu iluzia paralelismului...). Iata cateva exemple:

- lucru cu I/O (ex: retea): asteptarea si tratarea unui eveniment se poate face intr-un thread separat, iar intre timp programul principal ruleaza (altminteri ar fi fost blocat asteptand date de intrare)
- realizarea unei operatiuni consumatoare de timp si resurse, pastrand in acelasi timp o interfata grafica ce raspunde la comenzi utilizatorului (ex: la click pe un buton se incepe o operatie de durata)
- realizarea mai multor operatiuni simultan (sau dand impresia de simultaneitate). Spre exemplu, dorim sa afisam un progress bar ce evidentaiza stadiul unei operatii in paralel cu efectuarea acesteia
- optimizarea unei aplicatii care ruleaza pe un sistem multi-procesor prin definirea mai multor thread-uri care "muncesc" in paralel la aceeasi operatie (ex: calcule/statistici costisitoare)

10.1.3. Modalitati de lucru cu fire de executie in Java

In Java exista urmatoarele abordari ale lucrului cu fire de executie:

- utilizarea unor solutii de nivel inalt, care "imbraca" lucrul cu fire de executie oferind servicii la cheie. Amintim aici metodele `invoke*`() din clasa **SwingUtilities**; clasele **java.util.Timer** si **java.util.TimerTask** care permit rularea de cod intr-un fir de executie separat, sub forma unei executii unice sau periodice; de asemenea, clasa **javax.swing.Timer** poate genera evenimente `ActionEvent` periodice care listenerii sai, cu perioada configurabila de catre utilizator. Rularea unica sau periodica a unui job in paralel cu restul aplicatiei este o operatie frecventa in programare si de aceea multe situatii se vor putea rezolva apeland la aceste clase de nivel inalt
- utilizarea directa a claselor de baza, **Thread** si **Runnable**. Acestea reprezinta o solutie flexibila pentru aplicatii/necesitati simple si reprezinta de fapt baza pentru toate celelalte variante
- pentru aplicatii mai complexe sau necesitati ce merg dincolo de simpla rulare unica/periodica exista infrastructuri care organizeaza rularea unui numar mare de thread-uri, sub forma claselor din pachetul **java.util.concurrent** si subpachetele sale.

In continuare vor fi explorate pe rand diferitele posibilitati.

10.2. Utilizarea claselor de nivel inalt

10.2.1. Clasa javax.swing.Timer

Un obiect de tip `javax.swing.Timer` are capabilitatea de a produce o singura data sau periodic evenimente de tip `ActionEvent`. In acest scop el dispune de o lisa de listeneri, primul dintre ei fiind adaugat chiar prin intermediul unicului constructor al clasei: **public Timer(int delay, ActionListener)**. Utilizarile unui astfel de timer sunt nelimitate: schimbarea periodica a unui banner de reclama din cadrul unei aplicatii, actualizarea unui ceas afisat, actualizarea de informatii in background etc.

Un timer Swing prezinta urmatoarele elemente de interes:

- poate fi sau nu repetitiv. Metode relevante: **setRepeats(boolean)** si **isRepeats()**. Implicit timer-ul este repetitiv
- dispune de doua intervale configurabile (initial ambele sunt setate la valoarea primita ca prim argument in constructor):
 - intarzirea inaintea primei executii - metodele **setInitialDelay(int)** si **getInitialDelay()** (intervalul este exprimat in milisecunde)
 - intarzirea dintre doua executii consecutive, atunci cand timerul este repetitiv - metodele **setDelay(int)** si **getDelay()**
- poate fi pornit, oprit sau repornit, folosind metodele **start()**, **stop()** sau **restart()**

Atentie! Trebuie tinut cont de faptul ca toate instructiunile din metodele `actionPerformed()` ale listenerilor timerului se executa in acelasi thread - si mai exact in event dispatching thread! De aceea ele trebuie sa ruleze repede - in caz contrar, interfata grafica va raspunde mai greu sau deloc la comenzi.

```
ChangeListener task = new ChangeListener() {
    private int secunde=0;
    public void actionPerformed(ActionEvent e) {
```

```

        System.out.println("Au trecut " + (++secunde) + " secunde");
    }
};

javax.swing.Timer timer = new javax.swing.Timer(1000,task);
timer.start();

```

10.2.2. Clasa java.util.Timer

Pachetul *java.util* ofera programatorului doua clase ce poseda facilitati suplimentare fata de cele din *javax.swing.Timer*:

Timer si TimerTask. Un obiect *TimerTask* contine codul ce trebuie executat, iar obiectul *Timer* executa unul sau mai multe astfel de task-uri intr-un SINGUR thread separat, dupa reguli stabilite de programator (de unde si necesitatea ca un task sa nu ruleze pe o perioada lunga, in caz contrar el intarziind executia task-urilor ulterioare).

Pasii de lucru sunt urmatorii:

- se creeaza obiectul *Timer*, care creeaza la randul sau thread-ul ce va rula task-urile
- pentru fiecare task:
 - se creeaza un obiect *TimerTask* (de exemplu, prin extinderea clasei *TimerTask*)
 - se adauga task-ul la timer folosind una dintre metodele *schedule**() (vezi mai jos)

TimerTask este o clasa abstracta, subclasele fiind obligate sa implementeze metoda **run()**; acesta este locul in care trebuie plasat codul de executat al task-ului:

```

TimerTask task = new TimerTask() {
    public void run() {
        System.out.println("Task-ul a rulat");
    }
};

```

In exemplul de mai sus a fost folosita o clasa interioara anonyma care deriveaza *TimerTask*. Desigur ca putea fi folosita si alta strategie (crearea unei clase separate ce extinde *TimerTask*, fie ea independenta sau interioara).

Odata task-urile definite, programatorul poate opta pentru:

- **executie unica sau periodica.** Executia periodica se poate realiza:
 - cu frecventa constanta. Timer-ul se va stradui sa existe un numar mediu constant de executii in unitatea de timp
 - cu interval intre repetitii constant. Avand in vedere ca fiecare executie poate ocupa timp diferit, aceasta face ca frecventa medie de repetitie sa nu mai fie stabila
- **executie imediata sau ulterioara.** In cel de-al doilea caz, momentul executiei poate fi stabilit in doua moduri:
 - precizand o intarziere (*delay*) fata de momentul adaugarii task-ului la timer
 - precizand momentul exact al rularii sub forma unui obiect *Date*

Aceste moduri de executie se reflecta in setul de metode *schedule**() ale clasei *Timer*:

Specificarea momentului primei/unicei executii	Executie unica	Executie cu frecventa medie fixa	Interval fix intre repetitii consecutive
sub forma unui delay initial	<i>schedule(TimerTask,long delay)</i>	<i>scheduleAtFixedRate(TimerTask,long,interval)</i>	<i>schedule(TimerTask,long delay,long perioada)</i>
sub forma unui moment exact in timp	<i>schedule(TimerTask,Date moment)</i>	<i>scheduleAtFixedRate(TimerTask,Date,interval)</i>	<i>schedule(TimerTask,Date,interval)</i>

```

Timer t=new Timer();
t.schedule(task,2000); // task-ul va porni dupa 2 secunde si va rula 1 data
t.schedule(task,2000,1000); // task-ul va porni dupa 2 secunde si va rula periodic la 1 secunda
t.schedule(task,0,1000); // task-ul va porni imediat si va rula periodic la 1 secunda

```

In plus fata de metodele prezentate, mai exista cateva de interes:

- in clasa *TimerTask* exista metoda *cancel()*. Daca task-ul era in plina executie el nu va fi oprit, insa orice executie ulterioara va fi anulata. Daca la apelarea metodei *cancel()* task-ul nu rulase inca sau nu a fost inca planificat (prin apelarea unei metode *schedule()* a unui *Timer*) el nu va mai rula niciodata
- in clasa *Timer* exista de asemenea o metoda *cancel()*, care opreste timer-ul impreuna cu toate task-urile sale. Un timer, odata anulat, nu mai poate fi repornit si nu se mai pot planifica task-uri in cadrul sau

10.3. Utilizarea claselor low-level

10.3.1. Functionarea multi-threading-ului in Java

10.3.1.1. Concepte

A lucra cu clase low-level presupune operarea directa cu obiectele care corespund firilor de executie din masina virtuala. Pentru a le putea folosi corespunzator este necesara intelegerea felului in care opereaza multithreading-ul in Java.

Firile de executie sunt administrate intern de catre masina virtuala Java; unui fir de executie Java ii poate corespunde sau nu un fir de executie al sistemului de operare gazda - softul de masina virtuala decide acest lucru. Fiecare thread este creat pe baza unei instante a clasei *Thread* sau a uneia dintre subclasele sale.

Thread-urile se executa *concurrent* - ele “lupta” pentru resurse comune (procesor, memorie). Accesul la resurse este arbitrat de planificatorul de thread-uri (*thread scheduler*) - modulul masinii virtuale care efectueaza urmatoarele operatii:

- gestioneaza memoria, oferind fiecarui thread spatiul de memorie propriu necesar executiei
- gestioneaza procesorul. Fiind resursa partajata, acesta trebuie oferit pe rand fiecarui thread. Thread-ul care se executa la un moment dat poarta denumirea de *thread activ*; celelalte se afla in asteptare. In momentul in care executia thread-ului activ se intrerupe sau se incheie, scheduler-ul trebuie sa aleaga un alt thread activ dintre cele aflate in asteptare. Algoritmul de alegere a firului de executie activ este dependent de implementarea planificatorului; exista mecanisme prin care programatorul poate influenta alegerea (ex: prin setarea prioritatii thread-urilor)

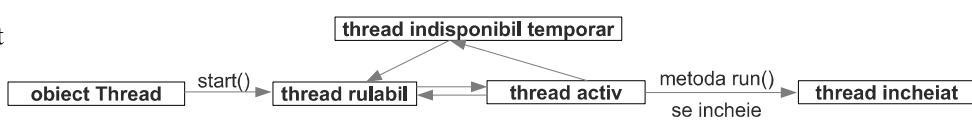
Nota: vorbim despre un singur procesor deoarece, chiar si atunci cand in sistem exista mai multe procesoare/core-uri, numarul de procese (si implicit de thread-uri) va fi in majoritatea cazurilor mult mai mare, determinand executie concurrenta la nivelul fiecarui procesor/core.

Firele de executie Java sunt si ele obiecte. Pentru a crea un fir de executie este necesara mai intai crearea unei instante a clasei *Thread* (sau a unei subclase). Clasa *Thread* dispune de metoda *run()*, in care trebuie plasat codul executat de acel thread in paralel cu restul programului; scopul existentei unui thread este rularea metodei sale *run()*. Prin apelarea metodei sale *start()*, obiectul *Thread* devine fir de executie, fiind adus in atentia scheduler-ului; acesta il va alege la un moment dat pentru a fi thread activ. Cat timp sta activ depinde de scheduler; este posibil ca intervalul respectiv sa fie suficient pentru ca thread-ul sa se execute in integralitatea sa, insa in cele mai dese cazuri el va fi interruput de catre scheduler inainte de a se incheia. Odata interruput, thread-ul revine in asteptare, si un alt thread va fi desemnat ca activ in locul lui. Ulterior, scheduler-ul va alege din nou thread-ul nostru si executia acestuia va avansa iarasi s.a.m.d. Pe ansamblu, “viata” unui thread consta dintr-o executie fragmentata a metodei sale *run()* - o alternanta intre perioade de timp in care asteapta sa fie ales de catre scheduler (si deci nu executa cod) si perioade de timp in care este activ (si deci executia metodei *run()* avanseaza). Thread-ul se incheie odata cu terminarea executiei metodei *run()*.

10.3.1.2. Starile posibile ale unui thread

Un thread se poate afla la un moment dat intr-una dintre urmatoarele stari posibile:

- **Nou** (simplu obiect Thread)
 - obiectul de tip thread a fost creat, insa nu exista un fir de executie al masinii virtuale corespunzator lui



- **Rulabil** (“*runnable*”) – la apelarea metodei **start()** a obiectului *Thread*, el se materializeaza intr-un fir de executie al masinii virtuale. Acesta este adaugat grupului de thread-uri aflate in asteptare si care concureaza pentru un procesor/core. Un astfel de thread este “in jocuri” - este luat in calcul de catre planificatorul de thread-uri, si la un moment dat va fi desemnat ca thread activ. **Atentie! Metoda start() nu porneste thread-ul, ci doar creeaza firul de executie corespunzator obiectului Thread si il aduce in atentia scheduler-ului (thread rulabil)!**
- **Activ** - reprezinta thread-ul curent care se executa, ca urmare a alegerii lui de catre planificatorul de thread-uri. Timpul petrecut ca thread activ poate acoperi parcial sau integral metoda *run()*; daca metoda se termina, thread-ul comuta in starea de thread incheiat, in caz contrar putand migra in starea de thread rulabil sau indisponibil
- **Incheiat** (“*dead*”) – un thread ajunge in aceasta stare la terminarea executiei metodei *run()*
- **Indisponibil temporar** (“*not runnable*”) – thread-ul este in desfasurare, insa este momentan oprit si nici nu concureaza (nu este luat in seama de catre scheduler) din motive ca:
 - asteptarea unei operatiuni I/O
 - thread-ul „doarme” ca efect al apelarii metodei *sleep()*; thread-ul va deveni din nou eligibil pentru rulare dupa exirarea intervalului specificat ca parametru al metodei
 - executia sa a fost suspendata folosind metoda *wait()*; thread-ul va deveni din nou rulabil numai dupa ce un alt thread apeleaza *notify()* sau *notifyAll()*
 - thread-ul astepta terminarea executiei unui alt thread ca urmare a apelarii metodei *join()*
 - thread-ul incerca sa apeleze o metoda *synchronized* a unui obiect al carui *lock* nu este disponibil (vezi sectiunea despre accesul la date comune)

10.3.2. Lucrul cu obiecte Thread

10.3.2.1. Crearea unui obiect Thread

Firele de executie al masinii virtuale Java sunt instante ale clasei **Thread**, subclasa directa a lui *Object*. Scopul declarat al unui thread este executarea metodei sale *run()*. In clasa *Thread*, metoda *run()* are implementare vida, si de aceea programatorul trebuie sa aiba posibilitatea de a specifica codul pe care il ruleaza thread-ul in cauza. Acest lucru poate fi efectuat in cel putin doua moduri:

- derivand clasa **Thread** si scriind (override) metoda *run()*, in care va fi plasat codul ce se doreste a fi executat:

```
class A extends Thread {
    public void run() {
        //codul dorit
    }
}
//...iar ulterior se instantiaza clasa:
Thread t=new A();
```

- scriind o clasa ce implementeaza interfata **Runnable**. Clasa nou-creata va fi astfel obligata sa aiba o metoda *run()*, in care trebuie introdus codul de executat; pe baza unei instante a clasei va fi creat un obiect *Thread* folosind constructorul **Thread(Runnable)**:

```
public class Alergator extends Sportiv implements Runnable {
    public void run() {
        //codul dorit
    }
}
//...ulterior, crearea obiectului de tip Thread se efectueaza astfel:
Thread t=new Thread(new Alergator());
```

Cel de-al doilea exemplu evidentiaza doua aspecte:

- este creata o instanta directa a clasei *Thread*; codul de executat este “inmanat” thread-ului prin intermediul obiectului de tip *Runnable* pasat ca parametru in constructor
- varianta implementarii interfetei *Runnable* este cea mai flexibila, deoarece poate fi folosita si atunci cand clasa noastra extinde deja o alta clasa

Ambele solutii de mai sus pot fi implementate cu ajutorul claselor interioare anonime:

```
// varianta cu extinderea clasei Thread
Thread t=new Thread(){
    public void run(){
        // codul dorit
    }
};

// varianta cu implementarea lui Runnable
Thread t=new Thread(new Runnable(){
    public void run(){
        // codul dorit
    }
});
```

Fiecare obiect *Thread* are un nume (de tip *String*), care poate fi obtinut cu metoda sa *getName()* si este configurabil in doua moduri:

- folosind constructorii lui *Thread* care primesc ca argument numele: *Thread(String)*, *Thread(Runnable, String)* etc
- folosind metoda *setName(String)*

Cand numele thread-ului nu este specificat explicit, la instantiere el va fi ales automat de forma Thread-1, Thread-2 etc in functie de numarul de thread-uri create deja.

Nota: numele thread-ului curent poate fi obtinut cu Thread.currentThread().getName().

10.3.2.2. Pornirea si rularea thread-urilor

Odata creat, un obiect *Thread* este doar atat – un obiect – fara a se fi materializat intr-un fir de executie in sistemul de operare. Pentru a crea intra-devar un thread este necesara apelarea metodei **start()** a obiectului, care adauga noul fir de executie in grupul de thread-uri aflate in asteptare ce concureaza pentru atentia scheduler-ului. Metoda *start()* se incheie imediat ce noul thread a fost creat si adaugat in grup, fara a astepta rularea acestuia:

```
// folosirea clasei Fir de mai sus
Thread t=new Thread(new Fir());
t.start(); // se creeaza un nou fir de executie, plasat in starea de thread rulabil
// mesaj afisat imediat dupa pornire (foarte posibil, inainte ca firul sa inceapa sa se execute!)
System.out.println("Thread-ul a pornit!");
```

Trebuie inteleas faptul ca noul thread nu va rula imediat, ci atunci cand scheduler-ul va decide acest lucru; decizia va fi luata in functie de prioritatea noului thread si a celorlalte aflate in asteptare si de alte diferite aspecte (multe dintre ele dependente de implementarea scheduler-ului!). Cand scheduler-ul alege prima data thread-ul si il desemneaza ca activ, metoda *run()* a acestuia din urma va incepe sa ruleze. Executia metodei *run()* este in general una foarte fragmentata; dupa debutul sau, ea va rula un scurt interval de timp, dupa care scheduler-ul va acorda procesorul altui thread. Ulterior, thread-ul nostru va primi din nou atentia scheduler-ului, o va pierde din nou s.a.m.d, astfel incat “viata” thread-ului (executia lui *run()*) va consta dintr-o lista de intervale de timp in care firul de executie capata procesor. Executia thread-ului se incheie la iesirea din metoda *run()*. Unui thread care s-a incheiat nu i se mai poate apela metoda *start()* - tentativa va genera o eroare de tip *IllegalThreadStateException*.

Atentie! Rularea directa a metodei run() nu este totuna cu apelarea metodei start()! A apela metoda *run()* a unui obiect thread inseamna rularea codului acestuia in thread-ul curent, pe cand a apela *start()* creeaza un nou fir de executie in care se va executa codul din *run()*.

Atunci cand rulam mai multe thread-uri, putem doar influenta modul in care ele se intretin - in general felul in care ele se combina depinde de scheduler si este impredictibil; fie exemplul de mai jos ce ruleaza doua thread-uri care afiseaza numerele de la 1 la 100:

```
public class Threaduri
{
```

```

public static void main(String[] a){
    Thread f1 = new Thread(new Fir(), "Primul fir");
    Thread f2 = new Thread(new Fir(), "Al doilea fir");
    f1.start();
    f2.start();
}
class Fir implements Runnable{
    public void run(){
        for(int i=0;i<100;i++){
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }
}

```

Programul de mai sus ofera o singura garantie: ca cele 3 fire de executie (*main* plus cele doua pornite din cadrul sau) vor rula pana la incheierea lor. In rest insa nu avem certitudini:

- desi *f1* este pornit inaintea lui *f2*, nu inseamna ca va deveni activ inaintea acestuia din urma; schedulerul este cel care va decide cand *f1* devine thread activ - posibil dupa *f2*!
- chiar daca *f1* porneste primul, nu putem sti daca el ruleaza pana la afisarea tuturor celor 100 de numere sau va fi interupt inainte
- ruland de mai multe ori programul constatam diferite moduri de intretesere a firelor de executie: uneori *f1* ruleaza complet si abia apoi se executa *f2*, alteori *f1* este interupt (nu in acelasi loc de fiecare data!) si ruleaza partial sau complet *f2*
- chiar daca pe o anumita platforma (calculator+sistem de operare) programul are de fiecare data aceeasi manifestare, daca schimbam calculatorul, sistemul de operare sau softul de masina virtuala este posibil ca output-ul sa difere!

ATENTIE! IN MULTI-THREADING EXISTA FOARTE PUTINE GARANTII! Nu va lasati inselati de faptul ca un program pare a functiona corect pe o anumita platforma - este posibil ca el sa esueze lamentabil pe altele!
Intotdeauna scrieti programul astfel incat sa nu depinda de platforma si scheduler!

10.3.2.3. Prioritatile thread-urilor

Fiecare thread dispune de o prioritate - o valoare numérica de care scheduler-ul tine cont in alegerea thread-ului activ. Prioritatile posibile sunt cuprinse intre **Thread.MIN_PRIORITY** si **Thread.MAX_PRIORITY**; prioritatea implicită a unui thread este determinată astfel:

- pentru firele de executie de sine statatoare - cum este *main* - prioritatea implicită este **Thread.NORM_PRIORITY**
- daca un thread este pornit din cadrul altuia, el mosteneste prioritatea de la thread-ul sau parinte

Prioritatea unui thread poate fi extrasă sau modificată cu ajutorul metodelor **int getPriority()** și **setPriority(int)**.

Prioritatile thread-urilor au efect in două privințe:

- in desemnarea thread-ului activ de catre scheduler.** Cand trebuie să aleagă thread-ul activ, schedulerul va prefera thread-urile de prioritate mai mare. Prioritatile actionează astfel:
 - atunci cand in grupul de thread-uri rulabile exista un singur thread de prioritate mai mare decat celelalte, acesta va fi cel ales de catre scheduler
 - atunci cand toate thread-urile aflate in asteptare sunt de aceeasi prioritate sau cand exista mai multe thread-uri de prioritate maxima, depinde de scheduler care dintre ele va fi desemnat ca activ; ele nu sunt neaparat alese in mod circular! De obicei planificatorul va incerca sa tina cont si de vechimea thread-ului in lista de asteptare, insa acest mod de actiune *nu este garantat*
- in timpul de procesor primit de catre thread-ul activ.** Timpul de procesor primit de catre un thread nu se afla in directa proportionalitate cu prioritatea sa; odata ales thread-ul activ, acesta va rula pana cand are loc una dintre urmatoarele situatii:
 - thread-ul se incheie (cazul cel mai evident)
 - thread-ul executa o instructiune care il plaseaza inapoi in grupul de asteptare (ex: metoda *sleep()*). Acest lucru se intampla fie cand thread-ul trebuie sa dea posibilitatea altor thread-uri de a se executa, fie atunci cand asteapta pentru a capata acces la o resursa momentan indisponibila

- thread-ul este interupt de catre scheduler. Planificatorul poate interupe un thread fie atunci cand intervalul de timp alocat lui se incheie (pentru sistemele de operare care lucreaza in time-slicing), fie atunci cand in timpul executiei sale apare in lista de asteptare un thread de prioritate superioara (pentru sistemele de tip preemptiv)

Algoritmul de programare a thread-urilor poate fi „preemptive” – executia thread-ului activ este interrupta la aparitia altuia cu prioritate mai mare. Pe unele sisteme non-preemptive, thread-ul cu prioritate maxima va rula pana cand executia sa se incheie (el neputand fi interupt de thread-uri cu prioritate egala sau mai mica), acest lucru ducand uneori la „starvation” – ramanerea in stare de asteptare pe perioade extinse a celorlalte thread-uri. In functie de sistemul de operare, este posibil ca acest efect sa fie evitat prin time-slicing (multiplexare in timp – fiecarui thread i se aloca un interval de timp, la expirarea caruia este trecut in starea runnable).

Constatam asadar ca nu avem garantia cooperarii pasnice intre thread-uri - este posibil ca unul dintre ele sa consume toate resursele in detrimentul celorlalte, pana cand executia sa se incheie; intre timp, celelalte sunt obligate sa stagnzeze. De aceea programatorul trebuie sa ia masuri pentru a evita situatia de *thread starvation* prin intreruperea voluntara a thread-ului activ (fie el si de prioritate maxima), creand intervale in care celelalte thread-uri sa se poata executa.

Nota: acest mod de a asigura resurse pentru thread-urile de prioritate mai mica este o forma de cooperative multithreading.

10.3.2.4. Plasarea voluntara a unui thread in asteptare

10.3.2.4.1. Situatii

Dupa cum s-a discutat anterior, exista situatii in care dorim ca thread-ul activ sa “abdice de la putere” - iata cateva motive posibile:

- pentru a da ocazia altor thread-uri (chiar si de prioritate mai mica) sa se execute. Pute folosi in acest scop metodele *sleep()* sau *yield()* prezентate mai jos
- pentru a astepta eliberarea unei resurse partajate - exista metodele *wait()* care, odata apelate unui obiect din thread-ul curent, plaseaza thread-ul in asteptare pana la apelarea unei metode *notify()* sau *notifyAll()* pentru acelasi obiect (dar din cadrul unui alt thread)
- pentru a astepta incheierea executiei unui alt thread - exista in acest scop metodele *join()*

Nota: pentru a innabusi din fasa o confuzie, trebuie precizat ca metodele *wait()*, *notify()* si *notifyAll()* fac parte din clasa *Object*, pe cand celelalte aparitin clasei *Thread*.

10.3.2.4.2. Renuntarea la statutul de thread activ: metodele *sleep()* si *yield()*

Clasa *Thread* dispune de metoda statica ***sleep(long)***, care plaseaza thread-ul curent in starea de indisponibil pentru o perioada de timp precizata in milisecunde. Thread-ul nu va concura cu celelalte in acest rastimp (nu va fi luat in calcul de catre scheduler). Metoda poate fi folosita in cel putin doua scopuri:

- multithreading cooperativ. Daca thread-ul activ are prioritate mai mare decat toate cele aflate in asteptare, prin “retragerea” sa temporara celelalte thread-uri capata sansa de a se executa, care altminteri nu le era garantata
- temporizare. Atunci cand un thread executa o anumita operatie periodic putem stabili un interval de pauza intre doua executii consecutive “adormind” thread-ul

Atentie! Metoda *sleep()* actioneaza numai asupra thread-ului curent (activ)! Un thread nu ii poate apela *sleep()* altuia!

Nota: cat timp este adormit, un thread nu eliberaza eventualele lock-uri (chei) de obiecte detinute! (vezi sectiunea dedicata accesului la resurse comune)

Un thread adormit poate fi trezit prematur prin intermediul unei cereri de intrerupere (vezi sectiunea corespunzatoare mai jos); in acest caz, metoda *sleep()* arunca o exceptie checked de tip *InterruptedException*, ce trebuie trataata atunci cand dorim ca thread-ul sa dea curs cererii de intrerupere. Aceasta este motivul pentru care apelul metodei *sleep()* este inclus de obicei intr-un bloc *try...catch*:

```

try{
    Thread.sleep(2000); // 2 secunde
}catch(InterruptedException e){
    // thread-ul se incheie ca urmare a solicitarii de intrerupere
    return;
}

```

Atunci cand intervalul de adormire se incheie, firul de executie revine in starea de thread *rulabil* (nu de thread activ!); de aceea executia sa nu va continua dupa perioada exacta specificata ca argument al lui *sleep()*, ci dupa un interval un pic mai mare - thread-ul trebuie sa fie ales mai intai de catre scheduler. Din acest motiv nu este indicat ca *sleep()* sa fie utilizat pentru a implementa operatii de mare precizie temporala; pe de alta parte, precizia sa este suficienta pentru majoritatea operatiilor de temporizare uzuale.

O alta modalitate de a ceda locul altor thread-uri este utilizarea metodei statice **Thread.yield()**, care insa nu ofera acelasi gen de garantii ca si *sleep()*. Apelul *Thread.yield()* are un efect asemanator cu *Thread.sleep(0)*: firul activ este trimis inapoi in grupul de asteptare, de unde va fi ales ulterior de catre scheduler. Din nefericire, nu avem nicio garantie ca thread-ul nostru nu va fi din nou cel ales: atunci cand el este de prioritate egala cu celelalte thread-uri din grup, este decizia scheduler-ului pe care dintre ele il va alege; daca are prioritate maxima, atunci va fi ales din nou cu certitudine! In concluzie, metoda nu poate fi folosita pentru a crea garantii.

10.3.2.4.3. Asteptarea incheierii executiei unui alt thread: metodele *join()*

Metoda **sleep()** prezentata anterior determina o asteptare cu o durata prestabilita; in programare insa, deseori un fir de executieiese din starea de thread activ tocmai pentru a astepta indeplinirea unei anumite operatii sau conditii in alt thread, fara a cunoaste a priori cand va surveni respectivul eveniment. Metodele **join()** ale clasei *Thread* permit unui thread sa astepte incheierea executiei altuia (sa-l numim *thread tinta*) intr-un mod cat mai putin costisitor; in lipsa lor, thread-ul care asteapta ar fi fost obligat sa verifice periodic incheierea thread-ului tinta, consumand astfel resurse in mod inutil.

Exista trei metode *join()* in clasa *Thread*:

- **join()** - plaseaza thread-ul curent in starea de temporar indisponibil pana cand executia thread-ului tinta se incheie
- **join(long interval)** - efect analog, dar thread-ul nu va astepta mai mult decat numarul de milisecunde specificat ca argument
- **join(long milisecunde, int nanosecunde)** - analog, dar cu precizie sporita la specificarea intervalului de asteptare

La incheierea metodei *join()* thread-ul care a apelat-o revine in starea de thread rulabil, urmand a fi ales de catre scheduler la un moment ulterior.

Atentie! Metoda join ii este apelata thread-ului tinta, dar are ca efect adormirea thread-ului curent!

Nota: metoda *join()* este de fapt implementata sub forma unei bucle in care thread-ul aflat in asteptare apeleaza *wait()* cat timp thread-ul tinta inca ruleaza (fapt determinat cu ajutorul metodei *isAlive()*). La incheierea executiei sale, thread-ul tinta apeleaza *this.notifyAll()*. (vezi sectiunea dedicata accesului concurrent la informatie)

Iata un exemplu mai elaborat. Avem o fereastra ce contine un buton si un progress bar. La apasarea pe buton este demarat un download - operatie a carei durata nu este cunoscuta (si oricum variaza de la o executie la alta) si nici nu poate fi monitorizata in dese cazuri. In consecinta, la apasarea butonului, progress bar-ul va fi comutat in mod *eterminate*, iar la incheierea download-ului va fi afisat un mesaj de notificare:

```

public class Download extends JFrame
{
    private JButton b = new JButton("Download");
    private JProgressBar p = new JProgressBar();
    public Download(){
        b.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                System.out.println("Downloading...");
                p.setIndeterminate(true);
            }
        });
    }
}

```

```
        Runnable r = new Runnable(){
            public void run(){
                try{
                    URL u = new URL("http://ftp.iasi.roedu.net/10MB");
                    BufferedReader r = new BufferedReader(
                        new InputStreamReader(u.openStream()));
                    while(r.readLine() != null){}
                }catch(Exception ex){
                    ex.printStackTrace();
                }
            }
        };
        final Thread download = new Thread(r);
        download.start();
        Thread notificare = new Thread(new Runnable(){
            public void run(){
                try{
                    download.join();
                }catch(Exception ex){
                    ex.printStackTrace();
                }
                p.setIndeterminate(false);
                p.setValue(p.getMaximum());
                System.out.println("Download incheiat!");
                JOptionPane.showMessageDialog(
                    Download.this,"Download incheiat!");
                }
            });
        notificare.start();
    }
});
```

Container pane = getContentPane();
pane.setLayout(new FlowLayout());
pane.add(b);
pane.add(p);
pack();
setVisible(true);
}

Cateva elemente din acest exemplu cer comentarii:

- download-ul se realizeaza creand un obiect de tip URL, a carui metoda *openStream()* produce un *InputStream*. Pe baza acestuia construim un *BufferedReader* ce permite citirea linie cu linie (mai eficienta)
 - din metoda *actionPerformed()* au fost pornite doua thread-uri separate - unul pentru operatia de download si celalalt pentru asteptarea incheierii download-ului si notificare. Thread-ul de notificare asteapta incheierea celui de download (apelandu-i *join()*) pentru a-si afisa mesajul. Sa ne amintim ca metodele de tratare a evenimentelor ruleaza toate in *event dispatching thread*; orice operatie de durata in acest thread trebuie evitata, deoarece ea tine pe loc intreaga functionare a interfetei grafice. In consecinta, atat download-ul (operatie de durata) cat si asteptarea notificarii (care se blocheaza pana la incheierea download-ului) trebuie trimise in thread-uri separate
 - thread-ul de download trebuie declarat ca *final* deoarece este accesat ulterior dintr-o clasa interioara anonyma (cea corespunzatoare obiectului de tip *Runnable* folosit pentru a construi thread-ul de notificare)
 - pentru a accesa fereastra de baza a aplicatiei din cadrul thread-ului de notificare (deoarece constituie parintele pentru fereastra de dialog de tip *JOptionPane*) se foloseste *Download.this*. Daca am fi folosit doar *this*, acesta s-ar fi referit la instanta de obiect *Runnable* (cea a clasei interioare)

10.3.2.5. Intreruperea unui thread

Clasa *Thread* ofera un mecanism prin care un thread poate solicita incheierea executiei altuia, sub forma metodei de instantia **interrupt()**. Trebuie intelese ca apelul metodei este nu o comanda, ci o "rugaminte" catre thread-ul tinta: daca acesta se va opri sau nu depinde de codul pe care programatorul l-a scris ca reactie la cererea de intrerupere.

Thread-ul poate receptiona cererea de intrerupere în două situații:

- **cand este indisponibil** – spre exemplu, in timpul executiei metodei `sleep()` sau `wait()`. Cererea de intrerupere va aduce thread-ul inapoi in starea de thread rulabil, iar metoda `sleep()/wait()` ce determinase indisponibilitatea va produce un `InterruptedException` (exceptie checked); codul de tratare a solicitarii de intrerupere trebuie plasat in `catch`-ul corespunzator
- **cand este thread activ**. Fiecare thread contine un flag (indicator) intern, care este activat in momentul primirii unei cereri de intrerupere; daca doreste sa reactioneze, thread-ul trebuie sa verifice din cand starea acestui indicator. Verificarea poate fi efectuata in doua moduri:
 - metoda `boolean Thread.interrupted()` (statica), care indica daca thread-ul curent a fost intrerupt si, in plus, anuleaza indicatorul de intrerupere la apelare
 - metoda `boolean isInterrupted()` a thread-ului in cauza (metoda de instanta), care lasa indicatorul activat

Iata un exemplu in care un thread afiseaza o succesiune de numere facand o pauza aleatoare intre fiecare doua valori consecutive. Thread-ul poate fi intrerupt numai cat timp este activ (catch-ul corespunzator trezirii din `sleep()` are corp vid); un alt thread incerca sa “prinda momentul propice” si sa-l opreasca.

```
public class Intreruperi extends Thread
{
    public void run() {
        for(int i=0;i<1000000;i++){
            if(Thread.interrupted()){
                System.out.println("Un fleac, m-au ciuruit..."); // thread-ul se incheie ca urmare a cererii de intrerupere
                return;
            }
            System.out.println(i);
            try{
                Thread.sleep((int)(Math.random()*1000));
            }catch(InterruptedException e){
                System.out.println("Cine ma striga in noapte? Sa strige in continuare..."); // poate intre timp thread-ul s-a incheiat...
            }
        }
    }

    public static void main(String[] a){
        Intreruperi i = new Intreruperi();
        i.start();
        try{
            for(int j=0;j<100000;j++){
                Thread.sleep((int)(Math.random()*100));
                if(!i.isAlive()) return; // poate intre timp thread-ul s-a incheiat...
                i.interrupt();
            }
        }catch(InterruptedException e){}
    }
}

/*
...posibil output la executie:
Cine ma striga in noapte? Sa strige in continuare...
387
Cine ma striga in noapte? Sa strige in continuare...
388
Cine ma striga in noapte? Sa strige in continuare...
Un fleac, m-au ciuruit...
*/
```

10.3.2.6. Alte facilitati si metode utile ale clasei Thread

Clasa `Thread` mai ofera cateva facilitati utile fata de cele prezentate anterior:

- putem avea acces la thread-ul curent folosind `Thread.currentThread()`. Trebuie intelese ca orice portiune de cod pe care o scriem va ajunge sa se execute numai daca thread-ul din care face parte este cel activ; asadar are sens referirea la “thread-ul curent” independent de context, folosind o metoda statica a clasei `Thread`. Solutia este foarte utila atunci cand nu avem o referinta explicita catre thread-ul curent - spre exemplu, atunci cand, din cadrul metodei `run()` a unui `Runnable`, dorim sa ne referim la thread-ul care o ruleaza
- putem determina daca un thread este inca activ folosind metoda sa `isAlive()`

- un thread poate fi sau nu de tip *daemon*. Un daemon thread este unul al carui prezent nu impiedica inchiderea aplicatiei. Masina virtuala va astepta inchiderea executiei tuturor thread-urilor non-daemon; odata ce toate thread-urile non-daemon ale aplicatiei s-au inchis, masina virtuala va inchide fortat thread-ul demon si se va opri. Metode relevante din clasa *Thread*: **boolean isDaemon()** si **void setDaemon(boolean)**. Implicit, un thread mosteneste statutul de daemon/non-daemon de la thread-ul care l-a creat

10.4. Accesul concurrent la date comune

10.4.1. Identificarea problemei

Pana acum a fost abordat doar cazul executiei independente a doua sau mai multe thread-uri, insa adevaratele probleme apar la accesarea concurrenta a acelasi resurse din fire de executie multiple. Atata timp cat valoarea unei variabile ramane nemodificata, ea poate fi citita de oricate thread-uri si de oricate ori; problema este insa modificarea de catre (cel putin) un thread si citirea de catre altele, deoarece citirea poate interveni in orice moment si este posibil sa surprinda obiectul pe parcursul modificarii sale, intr-o stare intermediara care nu ar trebui sa fie vizibila in exterior. Aceeași problema o ridică modificarea concurrenta a acelorasi date din doua sau mai multe thread-uri. Sa luam exemplu de mai jos:

```
class NumarPar{
    private int i = 0;
    public void next() { i++; i++; }
    public int get() { return i; }
}
public class Surpriza
{
    public static void main(String[] a){
        final NumarPar n = new NumarPar();
        Runnable r = new Runnable(){
            public void run(){
                while(true){
                    n.next();
                    int i = n.get();
                    if(i%2 != 0){
                        System.out.println("Ah-ha! Numar impar: "+i);
                        return;
                    }
                }
            }
        };
        new Thread(r).start();
        new Thread(r).start();
    }
}
```

Rulandu-l, observam ca mai devreme sau mai tarziu va fi citit un numar impar! Acest lucru este cauzat de faptul ca unul dintre thread-uri ajunge sa fie interrupție de către scheduler între cele două incrementări (si trimis înapoi în starea de thread rulabil), iar când celalalt devine activ gaseste obiectul într-o stare intermediară pe care nu ar fi trebuit să o vada. De aceea operațiile de modificare trebuie să fie „atomice” (indivizibile) dacă în același timp se face și citirea sau modificarea acelorași date de către alt thread.

Atentie! Operațiile implicit atomice din Java sunt putine; printre ele se numără atribuirea de valori variabilelor de tip primitiva byte, short, int, char, boolean, float, insă nu și pentru long sau double! Restul operațiilor pot fi interrupție pe parcursul desfasurării lor deoarece presupun mai mulți pași (ex: incrementarea $(x++)$ presupune citirea valorii lui x și depunerea în memorie și abia apoi incrementarea zonei de memorie ce constituie variabila).

In aproape toate sistemele de multithreading problema este rezolvata prin serializarea accesului la resursele comune: thread-urile sunt obligate sa acceseze pe rand portiunile de cod critice. Mecanismul este intalnit si sub denumirea de **mutex** (mutual exclusion - excludere reciproca).

10.4.2. Sincronizarea thread-urilor

Pentru a preveni situatiile de felul celor din exemplul anterior, programatorul trebuie sa aiba posibilitatea de a desemna portiuni de cod ca fiind executate indivizibil (atomic) - odata ce un thread a inceput executia unei astfel de sectiuni nu trebuie ca un alt thread sa poata opera pe aceleasi date, fie ca doreste sa le citeasca sau sa le modifice. Java ofera o rezolvare a problemei prin intermediul cuvantului cheie **synchronized**.

Fiecare obiect (instanta directa sau indirecta de *Object*) dispune de un unic asa-numit *monitor* sau *lock*, care poate fi gandit ca o **cheie** ce "incui" portiunile de cod declarate *synchronized* din cadrul obiectului. Fiecare portiune de cod *synchronized* este atasata cheii unui anumit obiect; pentru ca un thread sa poata executa codul *synchronized* el trebuie intai sa obtina cheia ("acquire the lock"). Cheia fiind unica per obiect, odata ce un thread a inceput sa execute codul *synchronized* in cauza, orice alt thread care doreste sa execute cod sincronizat atasat aceleiasi chei este obligat sa astepte eliberarea cheii. In acest fel se creeaza portiuni execute atomic ce previn accesul simultan al mai multor thread-uri la datele comune. Cheia devine un fel de stafeta ce trebuie pasata de la un thread la altul.

Cuvantul cheie **synchronized** poate fi folosit in doua moduri:

- **o intreaga metoda poate fi declarata synchronized.** O clasa poate contine mai multe astfel de metode. Pentru ca un thread sa poata executa o metoda *synchronized* este necesar ca el sa obtina mai intai cheia obiectului in cauza; cat timp cheia se afla in posesia acelui thread, alte thread-uri nu mai pot executa *nicio alta metoda synchronized* a obiectului, in schimb pot executa orice metoda ne-*synchronized*. Iata cum am putea remedia exemplul anterior astfel incat, daca un thread este intrerupt intre cele doua incrementari, alt thread sa nu poata executa metoda *next()* pana cand primul o incheie:

```
class NumarPar{
    private int i = 0;
    public synchronized void next(){ i++; i++; }
}
```

- **o simpla portiune de cod poate fi declarata synchronized**, specificand ca parametru obiectul „incuiat”. Aceasta posibilitate exista deoarece sincronizarea integrala a metodelor este costisitoare: un thread nu poate executa niciuna dintre metodele *synchronized* cat timp un altul detine cheia obiectului, ceea ce duce la timpi de asteptare mari si care prin cumulare duc la o viteza mai mica de executie a aplicatiei. Vrem sa restrangem codul *synchronized* la un minim - ceea ce inseamna sa putem delimita ca atomice doar cateva instructiuni din cadrul unei metode:

```
public void f(){
    //...instructiuni...
    synchronized(this){ ...cod executat atomic... }
    //...alte instructiuni...
}
```

Pentru ca un thread sa poata executa portiunea de cod *synchronized*, el este nevoit sa intre in posesia cheii obiectului specificat ca parametru. In acest fel portiuni de cod din clase diferite se pot exclude reciproc cand vine vorba de accesul la acelasi obiect.

Cuvantul cheie *synchronized* nu trebuie folosit fara discernamant si nu trebuie sa ne creeze o falsa senzatie de siguranta; sa consideram implementarea clasei *NumarPar* in care doar metoda *next()* este *synchronized*. Observam doua lucruri:

- timpul necesar atingerii aceleiasi valori a lui *i* devine mult mai mare decat in cazul ne-sincronizat. Se confirma faptul ca penalitatea de viteza introdusa de sincronizare este importanta, asadar trebuie sa folosim *synchronized* numai acolo unde acesta se justifica
- desi acum cele doua operatii de incrementare se executa atomic, tot ni se semnalizeaza aparitia unui numar impar! (dar mai tarziu decat inainte). Ne explicam acest lucru prin faptul ca, in timp ce un thread executa cod *synchronized* si il incrementeaza pe *i*, celalalt thread executa metoda *get()* - nesincronizata - si care "prinde" variabila *i* intre cele doua incrementari! Dorim ca citirea valorii lui *i* sa nu poata surveni pe parcursul modificarii sale, si de aceea si metoda *get()* trebuie si ea sincronizata:

```
class NumarPar{
    private int i = 0;
    public synchronized void next(){ i++; i++; }
    public synchronized int get(){ return i; }
}
```

O alternativa ar fi fost declararea ca *synchronized* a portiunii de cod care efectueaza modificarea si apoi citirea valorii din metoda *run()*:

```
public void run(){
    while(true){
        int i;
        synchronized(n) {
            n.next();
            i = n.get();
        }
        if(i%2 != 0){
            System.out.println("Ah-ha! Numar impar: "+i);
            System.exit(0);
        }
    }
}
```

10.4.3. Sincronizarea in cazul metodelor statice

Pana acum au fost prezentate doar modalitatile de “incuiere” a metodelor de instanta; exista insa si posibilitatea de a sincroniza metode statice. A cui cheie va fi folosita insa? (fiind vorba de cod care nu depinde de vreodata a clasei)

Java foloseste in acest caz cheia obiectului de tip *Class* corespunzator clasei in care se afla metoda.

Există asadar două tipuri de cheie:

- **per obiect** – fiecare obiect al unei clase dispune de o unică cheie, care incuiă toate metodele de instantă declarate ca *synchronized*
- **per clasa** – fiecarei clase încarcată în mașina virtuală îi corespunde un obiect *Class* care are la rândul său o cheie. Aceasta este cea folosită pentru a incui metodele statice

Fie exemplul generării de numere pare, rescris astfel încât să folosească, în locul unui obiect de tip *NumarPar*, un camp static al clasei:

```
class NumerePareStatic extends Thread{
    private static int i=0;
    public void next(){ i++; i++; }
    public void run(){
        while(true){
            next();
            if(i%2 != 0){
                System.out.println("Numar impar: "+i); System.exit(0);
            }
        }
    }
    public static void main(String[] a){
        new NumerePareStatic().start(); new NumerePareStatic().start();
    }
}
```

După cum era de așteptat, la rulare obținem destul de repede un număr impar. Un prim impuls de rezolvare a problemei ar putea fi declararea metodei *run()* ca *synchronized*:

```
public synchronized void run() { ... }
```

Aparent masura luata ar trebui sa asigure executia atomica a operatiilor din metoda si deci imposibilitatea generarii unui numar impar. Surpriza insa! Obtinem in continuare numere impare, si aproape la fel de repede! Dupa un moment de gandire, realizam ca metoda `run()` este sincronizata folosind cheia obiectului curent si nu a clasei; fiecare dintre cele doua thread-uri are cate o cheie si executia codului sincronizat este conditionata de *acea* cheie - ceea ce inseamna ca fiecare thread depinde de fapt de o alta cheie si deci cand unul se executa nu il exclude pe celalalt! Avem nevoie sa sincronizam portiunile de cod critice folosind *aceeasi* cheie - si de aceea vom scrie clasa astfel incat metodele care modifica/acceseaza variabila *i* sa fie statice si sincronizate:

```
class NumerePareStatic extends Thread{
    private static int i=0;
    public static synchronized void next(){ i++; i++; }
    public static synchronized int get(){ return i; }

    public void run(){
        while(true){
            next();
            int x = get();
            if(x%2 != 0){
                System.out.println("Numar impar: "+i); System.exit(0);
            }
        }
    }

    public static void main(String[] a){
        new NumerePareStatic().start(); new NumerePareStatic().start();
    }
}
```

Astfel, cat timp un thread executa `next()`, celalalt nu poate apela `get()`. O solutie alternativa era declararea ansamblului de operatii modificare+citire ca fiind atomic:

```
class NumerePareStatic extends Thread{
    private static int i=0;
    public static void next(){ i++; i++; }

    public static int get(){ return i; }

    public void run(){
        while(true){
            int x;
            synchronized(getClass()){
                next();
                x = get();
            }
            if(x%2 != 0){
                System.out.println("Numar impar: "+i); System.exit(0);
            }
        }
    }
}
```

A se observa ca parametrul lui `synchronized` este `getClass()`, care returneaza obiectul *Class* corespunzator clasei curente; alternativ putem folosi `NumerePareStatic.class`. In acest fel threadurile se sincronizeaza pe acelasi obiect (rezistam tentatiei de a folosi `synchronized(this)`, care ar fi dus din nou la utilizarea a doua chei diferite).

Remarca: argumentul lui `synchronized` putea fi orice alt obiect, atata timp cat era unul singur! Unicul sau scop este excluderea reciproca a thread-urilor care executa acea portiune de cod; asadar am fi putut scrie la fel de bine `synchronized(System.out)`!

10.4.4. Semnalizarea inter-thread-uri: asteptarea unui eveniment

Exista cazuri in care un thread nu poate avansa in lipsa indeplinirii unei conditii care depinde de altul. Spre exemplu, un fir de executie citeste date din retea si le depune intr-un obiect; un alt fir preia datele din obiect si calculeaza pe baza lor

statistici. Generalizand, astfel de scenarii sunt numite *producer-consumer*: un thread (“producer”) produce/obtine informatia iar celalalt (“consumer”) o asteapta si o foloseste ori de cate ori apare.

Un thread de tip *consumer* trebuie sa astepte disponibilitatea informatiei necesare, ceea ce presupune testarea unei conditii (“a aparut informatie noua?”). In acest scop el are la dispozitie strategii precum:

- o bucla in care verifica continuu daca conditia in cauza este indeplinita, actionand numai in momentul in care aceasta comuta pe *true*. Aceasta strategie este denumita “busy wait” si este cea mai proasta dintre cele posibile, deoarece thread-ul consuma resurse in timpul asteptarii:

```
while(!dateDisponibile) {} // cat timp nu avem date, iteram in gol, consumand timp de procesor
prelucrareDate();           // odata devenite disponibile datele, le putem prelucra
```

- o bucla in care thread-ul *consumer* verifica din cand in cand indeplinirea conditiei; intre doua verificari thread-ul este trimis in starea de indisponibil folosind *Thread.sleep()*. Solutia are dezavantajul ca thread-ul nu va detecta imediat indeplinirea conditiei, ci abia la urmatoarea trezire, si deci reactia sa nu va fi instantanee; per ansamblu, apar timpi morți:

```
while(!conditieIndeplinita) { Thread.sleep(100); } // "nu sunati! ies eu din cand..."
```

- utilizarea metodelor *wait()*, *notify()* si *notifyAll()*, prin care un thread poate semnaliza altora schimbarea starii unui obiect ce contine date comune. Aceasta ultima varianta are atat avantajul de a folosi putine resurse, cat si acela al reactiei aproape imediate la indeplinirea conditiei asteptate (ex: datele s-au schimbat sau au devenit disponibile)

Primele doua variante sunt prea ineficiente pentru a fi dezvoltate; ne vom concentra pe solutia low-level de semnalizare inter-thread-uri oferita de metodele *wait()*, *notify()* si *notifyAll()*.

Cele trei metode fac parte din clasa *Object* si ii pot fi apelate oricarui obiect Java. Cand apelam *wait()* pentru un obiect, thread-ul curent (cel care a efectuat apelul) va comuta in starea de thread indisponibil si va reveni din ea atunci cand un alt thread apeleaza una dintre metodele *notify()* sau *notifyAll()* pentru *aceasi* obiect. Fiecare obiect dispune de o lista de thread-uri “adormite” atasate lui; fiecare apel de metoda *wait()* aplicat obiectului adauga thread-ul curent in aceasta lista.

Atentie! Atunci cand un thread apeleaza metoda *wait()*, *notify()* sau *notifyAll()* a unui obiect, el trebuie sa detina cheia obiectului in cauza! In caz contrar apelul va produce o exceptie de tip *IllegalMonitorStateException*. Din acest motiv metodele respective sunt apelate din contexte *synchronized*.

Imediat dupa ce thread-ul apeleaza *wait()* el **elibereaza cheia obiectului** si trece in starea de indisponibil. Thread-ul va reveni din aceasta stare in cea de thread rulabil ca urmare a unuia dintre urmatoarele evenimente:

- este apelata una dintre metodele *notify()* sau *notifyAll()* pe obiectul caruia thread-ul i-a apelat metoda *wait()*
- se solicita intreruperea thread-ului aflat in asteptare, caz in care metoda *wait()* arunca *InterruptedException*
- thread-ul este trezit din alte motive. In functie de sistemul de operare si de implementarea masinii virtuale, pot exista asa-numitele *spurious wakeups* - treziri inopinate ale unui thread aflat in asteptare, care nu sunt cauzate de niciuna dintre actiunile de mai sus

Atunci cand un thread aflat in asteptare primeste o notificare, el “se trezeste” - revine din starea de asteptare in cea de thread rulabil, insa pentru a continua executia are nevoie sa obtina din nou cheia obiectului.

Nota: comparatie intre *wait()* si alte metode care determina indisponibilitatea thread-ului curent:

- *diferenta dintre join() si wait() este ca join() asteapta incheierea unui alt thread, pe cand wait() asteapta doar indeplinirea unei conditii, fara ca thread-ul tinta sa-si incheie executia.*
- *wait() si sleep() se manifesta diferit in privinta cheilor detinute; wait() elibereaza cheia obiectului inainte de a face thread-ul indisponibil, in schimb la apelarea lui sleep() cheia ramane in posesia thread-ului pe intreaga durata a indisponibilitatii!*

Diferenta dintre metodele *notify()* si *notifyAll()* este urmatoarea:

- *notify()* va trezi **numai unul** dintre thread-urile aflate in asteptare atasate obiectului. Nu este sub controlul nostru care dintre thread-uri va fi trezit. Procedam astfel atunci cand ne este egal care thread va fi reactivat
- *notifyAll()* va trezi **toate** thread-urile in asteptare atasate obiectului. Deisur ca nu vor putea rula toate deodata - unul singur poate fi ales de catre scheduler la un moment dat - insa unul cate unul le va veni randul. Procedam astfel atunci cand dorim sa fim siguri ca *un anumit thread* a fost trezit; cum nu putem dicta care sa fie acela, pentru a avea garantia ca thread-ul *nostru* va rula, le trezim pe toate si le lasam sa concureze.

Iata un exemplu in care un thread genereaza continuu numere aleatoare si un alt thread preia aceste numere. Ambele thread-uri efectueaza o pauza aleatoare intre doua operatii consecutive; masura a fost introdusa pentru a simula situatiile reale, in care primului thread (producer-ul) ii ia timp sa produca datele, iar celui de-al doilea (consumer-ul) ii ia timp sa le prelucreze, cele doua thread-uri fiind astfel obligate din cand in cand sa se astepte unul pe altul.

```

public class GeneratorNumere extends Thread{
    private Integer nr = null;
    public synchronized void numarNou(){
        while(nr!=null){ // cat timp numarul nu a fost preluat de cititor, asteptam
            System.out.println("Generatorul asteapta extragerea numarului curent...");
            try{ wait(); }catch(InterruptedException e){}
        }
        nr = (int)(Math.random()*100);
        System.out.println("Generatorul a produs numarul "+nr);
        notify(); // instiintam cititorul ca a aparut un numar nou
    }

    public synchronized Integer extragereNumar(){
        while(nr==null){ // cat timp nu exista inca numar, asteptam
            System.out.println("Cititorul asteapta generarea unui nou numar...");
            try{ wait(); }catch(InterruptedException e){}
        }
        int x = nr; nr = null; // salvam valoarea curenta pt a o returna si anulam nr
        notify(); // instiintam generatorul ca a fost extrasă valoarea curenta
        return x;
    }

    public void run(){
        while(true){ // generam continuu numere, cu pauza aleatoare intre ele
            numarNou();
            try{Thread.sleep((int)(Math.random()*1000));}catch(InterruptedException e){}
        }
    }
    public static void main(String[] a){
        new CititorNumere(new GeneratorNumere());
    }
}
class CititorNumere extends Thread{
    private GeneratorNumere g;
    public CititorNumere(GeneratorNumere x){ g=x; g.start();start();} // pornire ambele threaduri
    public void run(){
        while(true){
            System.out.println("Cititorul a primit numarul "+g.extragereNumar());
            try{Thread.sleep((int)(Math.random()*1000));}catch(InterruptedException e){}
        }
    }
}
/* fragment de output posibil:
Generatorul a produs numarul 12
Cititorul a primit numarul 12
Cititorul asteapta generarea unui nou numar...
Generatorul a produs numarul 3
Generatorul asteapta extragerea numarului curent...
Cititorul a primit numarul 3
Generatorul a produs numarul 28
Cititorul a primit numarul 28
Cititorul asteapta generarea unui nou numar...
*/

```

In exemplul de mai sus evidențiem urmatoarele particularități de implementare:

- campul de tip *Integer* al clasei *GeneratorNumere* reprezintă obiectul partajat între cele două thread-uri - locul în care este plasat noul număr generat și care mai apoi trebuie preluat de către cititor.
- metoda *numarNou()* este apelată de către generator, pe când metoda *extragereNumar()* de către thread-ul de citire. Ambele metode sunt *synchronized* (și utilizează cheia aceluiași obiect!), din două motive: în primul rand, dorim ca generatorul să încheie toate operațiile de creare a numarului înainte ca cititorul să-l poată accesa; în al doilea rand, pentru a apela *wait()* thread-ul trebuie să detină cheia obiectului.
- obiectul a cărui cheie este folosită pentru sincronizare și notificare nu este cel de tip *Integer*, astăzi cum poate ne-am aștepta, ci cel de tip *GeneratorNumere*. Aceasta deoarece obiectul *Integer* primește valoarea *null* odată ce a fost extras de către cititor și, în plus, la fiecare generare el este de fapt un *nou* obiect. În fond, putem folosi cheia oricărui obiect pentru excludere reciprocă a thread-urilor, cat timp toate se referă la același obiect
- a se observă modul în care, înainte de a genera un număr nou, generatorul așteaptă “consumarea” celui vechi de către cititor. Aceeași strategie este aplicată de către metoda de citire a numarului. Așteptarea în buclă pare redundantă (gândim: “în fond, odată ce thread-ul a apelat *wait()*, el nu va fi trezit de către *notify()*/*notifyAll()* apelat de celalalt thread?”); să ne amintim însă că un thread care a apelat *wait()* se mai poate trezi și din alte motive decât *notify()*, și deci e posibil ca acea condiție pe care el o așteaptă să nu fie încă indeplinită

10.5. Pachetul *java.util.concurrent*

10.5.1. Prezentare. Principii

Pachetul *java.util.concurrent* împreună cu subpachetele sale a fost introdus în Java 5 pentru a simplifica managementul lucrului cu grupuri (potențial mari) de thread-uri în aplicații complexe care lucrează pe statii multiprocesor/multicore.

Pachetele amintite oferă servicii precum:

- soluții de gestionare a accesului concurent la informație sub forma obiectelor de tip *Lock* (subpachetul *java.util.concurrent.locks*), ce reprezintă o soluție mai flexibilă și mai bogată decât mecanismul de *wait()*/*notify()*
- soluții pentru automatizarea rularii de thread-uri multiple sub forma de pachete de thread-uri (*thread pools*), prin intermediul obiectelor de tip *executor*
- clase colecție destinate accesului concurent și care oferă performanțe superioare celor produse de metodele clasei *Collections*

Vom prezenta în continuare infrastructura de clase și interfețe executor.

10.5.2. Executia task-urilor folosind obiecte de tip Executor

Pachetul *java.util.concurrent* introduce conceptul de **executor** - un obiect al cărui scop este executia de task-uri. Un task este un obiect ce conține codul ce se dorește executat. Executorului îl se înmanează task-ul de indeplinit și acesta îl rulează fie în thread-ul curent, fie într-un thread nou (în cele mai dese cazuri va fi preferată cea de-a două variantă). În acest fel programatorul este degrevat de crearea manuală de thread-uri, aceasta fiind automatizată și - în plus - existând o multitudine de servicii pe care diversele tipuri de executor le fac posibile (ex: așteptarea executiei unui task, posibilitatea anularii unui task pornit, programarea task-urilor cu anumite delay-uri sau periodic etc)

Un task de executat poate lua cel puțin două forme, în funcție de natura lui:

- un obiect a cărui clasa implementează **Runnable**. Aceasta reprezintă simplu cod de executat
- un obiect a cărui clasa implementează **Callable**. Aceasta reprezintă un task care produce un rezultat (sau aruncă o excepție în cazul imposibilității furnizării acestuia)

Orice obiect de tip executor este o instanță a unei clase care implementează una dintre următoarele interfețe:

- **Executor** - dispune de una singură metodă **void execute(Runnable)**. Argumentul este task-ul de executat, iar executorul va rula metoda *run()* a acestuia
- **ExecutorService** - reprezintă o subinterfață a lui *Executor*. Obiectele de acest tip permit, suplimentar, managementul executiei unui task (creare de noi task-uri (“task submission”), monitorizarea executiei task-urilor, încheierea acestora)

- **ScheduledExecutorService** - reprezinta o subinterfata a lui *ExecutorService*. Obiectele de acest tip au capabilitatea suplimentara de programare a task-urilor in moduri asemanatoare cu cele din clasa *java.util.Timer* (one-time sau periodic, cu sau fara intarziere initiala)

Un executor poate alege diverse strategii pentru rularea unui task (sa nu uitam ca toate cele de mai sus sunt simple interfete...):

- rularea metodei *run()* a task-ului direct in thread-ul din care a fost apelata metoda *execute()*. Aceasta reprezinta doar o posibilitate, dar clasele concrete din *java.util.concurrent* nu o folosesc
- intr-un *worker thread* - un thread in cadrul caruia se executa, pe rand, metodele *run()* ale diverselor task-uri pe care executorul le primeste de-a lungul timpului (recunoastem aici strategia lui *event dispatching thread* din AWT/Swing)
- intr-un thread pool - un grup de *worker thread*-uri carora executorul le atribuie task-urile pe masura ce le primeste. In acest fel se minimizeaza overhead-ul pornirii/opririi de noi thread-uri (la aparitia/incheierea task-ului), ce presupune operatii costisitoare cu memoria. In cele mai frecvente implementari executorul foloseste un *fixed thread pool*: numarul de worker thread-uri este fix, thread-urile fiind pornite de la bun inceput si stau in asteptare cat timp nu exista task-uri

Crearea obiectelor de tip *Executor* (sau derivate) se poate realiza in doua moduri:

- prin instantierea directa a claselor concrete ce implementeaza interfetele executor. Solutia este flexibila insa presupune customizarea obiectului in raport cu necesitatile programatorului. Iata cateva exemple de clase concrete executor:
 - **ThreadPoolExecutor** - un executor care implementeaza *ExecutorService* si care ruleaza task-urile folosind un grup de *worker threads* administrat intern
 - **ScheduledThreadPoolExecutor** - un executor ce foloseste aceeasi strategie ca cel anterior insa implementeaza interfata *ScheduledExecutorService*
 - **ForkJoinPool** (incepand cu Java 7) - un executor care implementeaza conceptul de *work-stealing*: threadurile care si-au "terminat treaba" incerca sa le "ajute" pe cele care inca lucreaza, preluand sub-task-uri ale acestora
- utilizarea metodelor statice din clasa **Executors**, care produc atat diversele tipuri de executori existenti (clase concrete predefinite) cu configuratiile uzuale, cat si alte tipuri de executor. Cateva exemple:
 - **Executors.newSingleThreadExecutor()** - un executor de tip *ExecutorService* ce utilizeaza un singur worker thread
 - **Executors.newFixedThreadPool(int nrFire)** - un executor de tip *ExecutorService* ce utilizeaza un numar prestabil de worker threads, dat de argumentul pasat metodei
 - **Executors.newScheduledThreadPool(int nrFire)** - analog cu exemplul anterior, insa produce un *ScheduledExecutorService*

```
public class Downloader implements Runnable{
    private String adresa;
    public Downloader(String a){ adresa = a; }
    public void run(){
        System.out.println("Se descarca "+adresa);
        try{
            URL u = new URL(adresa); InputStream i = u.openStream();
            int x;int n=0;
            do{ x = i.read(); n++; } while(x!=-1);
            System.out.println("Download incheiat pentru "+adresa+"; "+n+" octeti cititi");
        }catch(Exception e){
            e.printStackTrace();
        }
    }
    public static void main(String[] a){
        String[] adrese = { "http://www.a.ro", "http://www.a1.ro", "http://www.x.ro" };
        ExecutorService ex = Executors.newFixedThreadPool(2); // un executor cu 2 worker threads
        ExecutorService ex = Executors.newSingleThreadExecutor(); // un executor cu 1 worker thread
        // ExecutorService ex = Executors.newScheduledThreadPool(2); // un executor cu 2 worker threads
        for(String adr:adrese){ ex.submit(new Downloader(adr)); }
    }
}
```

La rularea exemplului de mai sus observam cum primele doua download-uri pornesc aproape simultan, iar cel de-al treilea va fi pornit abia dupa ce unul dintre primele doua s-a incheiat. Acest fapt este cauzat de numarul limitat de worker threads (doua); executorul accepta mai multe task-uri, insa numarul maxim de task-uri ce vor putea rula simultan este egal cu numarul de worker threads. Daca in exemplul de mai sus am fi folosit un executor ce utilizeaza un singur thread (linia comentata din metoda `main()`) am fi constatat cum download-urile ruleaza secvential - fiecare dintre ele este obligat sa astepte incheierea precedentului.

10.6. Clase si metode *thread-safe*

O metoda sau clasa *thread-safe* este una care poate fi accesata din mai multe thread-uri fara a cauza probleme de acces concurrent. Exista diferite solutii pe care programatorul le poate utiliza:

- obiecte immutable. Un obiect immutable este unul a carui stare interna nu se mai modifica odata ce a fost creat, si care in consecinta poate fi accesat de oricate thread-uri fara riscul coruperii datelor sale. Un exemplu celebru este clasa `String`
- clase colectie ale caror metode sunt sincronizate: clasa `Collections` ofera metode care produc variante sincronizate ale colectiilor uzuale (ex: `Collections.synchronizedList(List)`, `Collections.synchronizedMap(Map)` etc). Aceste obiecte au toate metodele synchronized; trebuie insa avute in vedere doua aspecte:
 - sincronizarea intregului cod din metode introduce importante penalitati de viteza, care degradeaza performantele colectiei
 - sincronizarea fiecarei metode in parte nu face implicit utilizarea clasei thread-safe! Spre exemplu, sa consideram cazul unui thread care verifica dimensiunea colectiei sincronizate apelandu-i `size()` si apoi extrage primul element folosind `get(0)`; este posibil ca intre cele doua apele thread-ului curent sa fie inrerupt de catre scheduler si un alt thread sa intervină si sa goleasca colectia! Ca urmare, cand primul thread va redeveni activ, incercarea de citire a primului element va esua, in ciuda verificarii initiale. Sincronizarea fiecarei metode in parte asigura consistenta interna a colectiei, insa nu garanteaza executia atomica a doua sau mai multe apele de metode ale colectiei; programatorul trebuie sa declare ca synchronized intregul calup de apele
- clase gandite pentru acces concurrent. Aceste clase reduc la un minim cantitatea de cod sincronizat, oferind performante superioare solutiei anterioare. Amintim aici:
 - clasa `StringBuffer` - o clasa in care portiunile-cheie de cod sunt sincronizate, permitand accesarea aceluiasi obiect `StringBuffer` din mai multe thread-uri fara a-i corupe continutul
 - colectiile concurente din pachetul `java.util.concurrent`, care ofera performante superioare celor sincronizate complet. Exemple: `ConcurrentHashMap`, `ConcurrentLinkedQueue`

10.7. BIBLIOGRAFIE

- Java Tutorial - Concurrency: <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
- Java Concurrency:
 - <http://tutorials.jenkov.com/java-concurrency/index.html>
 - <http://tutorials.jenkov.com/java-util-concurrent/index.html>