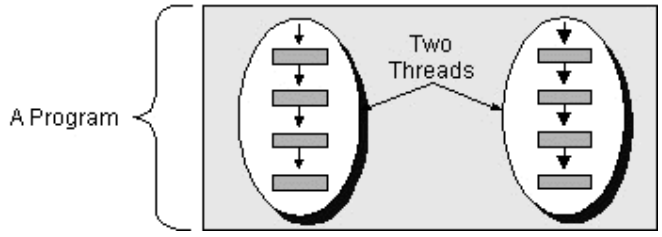
Curs 8
Fire de executie

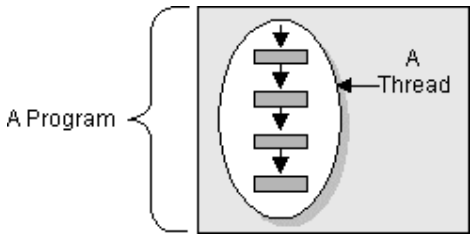
Ce este un fir de executie ?

Firele de executie fac trecerea de la programarea secventiala la programarea concurenta. Un program secvential reprezinta modelul clasic de program : are un început, o secventa de executie a instructiunilor sale si un sfârsit. Cu alte cuvinte, la un moment dat programul are un singur punct de executie. Un program aflat în executie se numeste *proces*. Un sistem de operare monotasking (MS-DOS) nu este capabil sa execute decât un singur proces la un moment dat în timp ce un sistem de operare multitasking (UNIX, Windows) poate rula oricâte procese în acelasi timp (concurent), alocând periodic cuante din timpul de lucru al CPU fiecarui proces. Am reamintit acest lucru deoarece notiunea de fir de executie nu are sens decât în cadrul unui sistem de operare multitasking.
Un fir de executie este similar unui proces secvential în sensul ca are un început, o secventa de executie si un sfârsit. Diferenta între un fir de executie si un proces consta în faptul ca un fir de executie nu poate rula independent ci trebuie sa ruleze în cadrul unui proces.

Definitie

> Un *fir de executie* este o succesiune sceventiala de instructiuni care se executa în cadrul unui proces.





Un program îsi poate defini însa nu doar un fir de executie ci oricâte, ceea ce înseamna ca în cadrul unui proces se pot executa simultan mai multe fire de executie, permitând executia concurenta a sarcinilor independente ale acelui program.
Un fir de executie poate fi asemanat cu o versiune redusa a unui proces, ambele rulând simultan si independent pe o structura secventiala de executie a instructiunilor lor. De asemenea executia simultana a firelor de executie în cadrul unui proces este similara cu executia concurenta a proceselor: sistemul de operare va aloca ciclic cuante din timpul procesorului fiecarui fir de executie pâna la terminarea lor. Din acest motiv firele de executie mai sunt numite si *procese usoare*.

Care ar fi însa deosebirile între un fir de executie si un proces ? In primul rând deosebirea majora consta în faptul ca firele de executie nu pot rula decât în cadrul unui proces. O alta deosebire rezulta din faptul ca fiecare proces are propria sa memorie (propriul sau spatiu de adrese) iar la crearea unui nou proces (fork) este realizata o copie exacta a procesului parinte : cod + date; la crearea unui fir de executie nu este copiat decât codul procesului parinte; toate firele de executie au deci acces la aceleasi date, datele procesului original. Asadar un fir de executie mai poate fi privit si ca un *context de executie* în cadrul unui proces parinte.

Firele de executie sunt utile în multe privinte, însa uzual ele sunt folosite pentru executarea unor operatii consumatoare de timp fara a bloca procesul principal : calcule matematice, asteptarea eliberarii unei resurse, acestea realizându-se de obicei în fundal.

Crearea unui fir de executie

Ca orice alt obiect Java, un fir de executie este o instanta a unei clase. Firele de executie definite de o clasa vor avea acelasi cod si, prin urmare, aceeasi secventa de instructiuni. Crearea unei clase care sa defineasca fire de excutie poate fi facuta prin doua modalitati:

1. **prin extinderea clasei Thread**
2. **prin implementarea interfetei Runnable**

Orice clasa ale carei instante vor fi executate într-un fir de executie trebuie declarata ca fiind **`Runnable`**. Aceasta este o interfata care contine o singura metoda, si anume metoda **`run`**. Asadar, orice clasa ce descrie fire de executie va contine o metoda `run` în care este implementat codul ce va fi executat de firul de executie. Interfata `Runnable` este conceputa ca fiind un protocol comun pentru obiectele care doresc sa execute un cod pe durata existentei lor (care reprezinta fire de executie).

Cea mai importanta clasa care implementeaza interfata `Runnable` este clasa **`Thread`**. Clasa `Thread` implementeaza un fir de executie generic care, implicit, nu face nimic. Cu alte cuvinte metoda `run` nu contine nici un cod.

Orice fir de executie este o instanta a clasei `Thread` sau a unei subclase a sa.

Extinderea clasei Thread

Cea mai simpla metoda de a crea un fir de executie care sa realizeze ceva este prin extinderea clasei `Thread` si supradefinirea metodei `run` a acesteia. Formatul general al unei astfel de clase este:

```
public class SimpleThread extends Thread {

        public SimpleThread(String nume) {
                super(nume);
                //apelez constructorul superclasei Thread
        }
        public void run() {
                //codul executat de firul de executie
        }
}
```

Prima metoda a clasei este constructorul, care primeste ca argument un sir ce va reprezenta numele firului de executie creat în momentul când constructorul este apelat.

```
SimpleThread t = new SimpleThread("Java")
//creeaza un fir de executie cu numele Java
```

In cazul în care nu vrem sa dam nume firelor de executie pe care le cream atunci putem renunta la definirea acestui constructor si sa ramânem doar cu constructorul implicit, fara argumente, care creeaza un fir de executie fara nici un nume. Ulterior acesta poate primi un nume cu metoda `setName(String)`.
Evident, se pot defini si alti constructori, acestia fiinde utili când vrem sa trimitem diversi parametri firului de executie.
A doua metoda este metoda `run`, "inima" oricarui fir de executie în care scriem efectiv codul pe care trebuie sa-l execute firul de executie.
Un fir de executie creat nu este automat pornit, lansarea sa în executie se realizeaza prin metoda **`start`**, definita de asemenea în clasa `Thread`.

```
SimpleThread t = new SimpleThread("Java")
t.start()
//creeaza si lanseaza un fir de executie
```

Sa consideram în continuare un exemplu în care definim un fir de executie ce afiseaza numerele întregi dintr-un interval cu un anumit pas. Firul de executie este implementat de clasa `Counter`.

```
class Counter extends Thread { //clasa care defineste firul de executie
        private int from, to, step;

        public Counter(int from, int to, int step) {
                this.from = from;
                this.to = to;
                this.step = step;
        }

        public void run() {
                for(int i = from; i <= to; i += step)
                        System.out.print(i + " " );
        }
}

public class TestCounter {     //clasa principala
        public static void main(String args[]) {
                Counter cnt1, cnt2;

                cnt1 = new Counter(0, 10, 2);
                //numara de la 0 la 100 cu pasul 5

                cnt2 = new Counter(100, 200, 10);
                //numara de la 100 la 200 cu pasul 10

                cnt1.start();
                cnt2.start();
                //pornim firele de executie
                //ele vor fi distruse automat la terminarea lor
        }
}
```

Gândind secvential, s-ar crede ca acest program va afisa prima data numerele de la 0 la 100 cu pasul 5, apoi numerele de la 100 la 200 cu pasul 10, întrucât primul apel este catre contorul `cnt1`, deci rezultatul afisat pe ecran ar trbui sa fie: `0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 100 110 120 130 140 150 160 170 180 190 200 .`
In realitate însa, rezultatul obtinut va fi o intercalare de valori produse de cele doua fire de executie ce ruleaza simultan. La rulari diferite se pot obtine rezultate diferite deoarece timpul alocat fiecarui fir de executie poate sa nu fie acelasi, el fiind controlat de procesor într-o maniera "aparent" aleatoare: `0 100 5 110 10 120 15 130 20 140 25` **`150 160 170 180 190 200`** `30 35 40 45 50 55 60 65 70 75 80 85 90 95 100`

Implementarea interfetei Runnable

Ce facem însa când dorim sa cream o clasa care instantiaza fire de executie dar aceasta are deja o superclasa, stiind ca în Java nu este permisa mostenirea multipla ?

```
class FirExecutie extends Parinte, Thread // ilegal !
```

In acest caz nu mai putem extinde clasa `Thread` ci trebuie sa implementam direct în clasa noastra interfata **`Runnable`**. Clasa `Thread` implementeaza ea însasi interfata `Runnable` si, din acest motiv, la extinderea ei obtineam o implementare implicita a interfetei. Asadar, interfata `Runnable` permite unei clase sa fie active, fara a extinde clasa `Thread`.
Interfata Runnable se gaseste în pachetul `java.lang` si este definita astfel:

```
public interface Runnable {
        pulic abstract void run( );
}
```

Prin urmare, o clasa care instantiaza fire de executie prin implementarea interfetei `Runnable` trebuie obligatoriu sa implementeze metoda `run`.
Formatul general al unei clase care implementeaza interfata `Runnable` este:

```
public class SimpleThread implements Runnable {

        private Thread simpleThread = null;

        public SimpleThread() {
                if (simpleThread == null) {
                        simpleThread = new Thread(this);
                        simpleThread.start();
        }

        public void run() {
                //codul executat de firul de executie
        }
}
```

Spre deosebire de modalitatea anterioara, se pierde însa tot suportul oferit de clasa `Thread` pentu crearea unui fir de executie. Simpla instantiere a unei clase care implemeneaza interfata `Runnable` nu creeaza nici un fir de executie. Din acest motiv crearea firelor de executie prin instantierea unei astfel de clase trebuie facuta explicit. Cum se realizeaza acest lucru ?
In primul rând trebuie declarat un obiect de tip `Thread` ca variabila membra a clasei respective. Acest obiect va reprezenta firul de executie propriu zis al carui cod se gaseste în clasa noastra.

```
private Thread simpleThread = null;
```

Urmatorul pas este instantierea si initializarea firului de executie. Acest lucru se realizeaza ca pentru orice alt obiect prin instructiunea `new`, urmata de un apel la un constructor al clasei `Thread`, însa nu la oricare dintre acestia. Trebuie apelat constructorul care sa primeasca drept argument o instanta a clasei noastre. Dupa creare, firul de executie poate fi lansat printr-un apel la metoda `start`.(Aceste operatiuni sunt scrise de obicei în constructorul clasei noastre pentru a fi executate la initializarea unei instante, dar pot fi scrise oriunde în corpul clasei sau chiar în afara ei)

```
simpleThread = new Thread( this );
simpleThread.start();
```

Specificarea argumentului `this` în constructorul clasei `Thread` determina crearea unui fir de executie care la lansarea sa va cauta în clasa noastra metoda `run` si o va executa. Acest constructor accepta ca argument orice instanta a unei clase "Runnable". Asadar metoda `run` nu trebuie apelata explicit, acest lucru realizându-se automat la apelul metodei `start`.
Apelul explicit al metodei `run` nu va furniza nici o eroare, însa aceasta va fi executata ca orice alta metoda, deci nu într-un fir de executie. Sa rescriem acum exemplul anterior (afisarea numerele întregi dintr-un interval cu un anumit pas), folosind interfata `Runnable`. Vom vedea ca implementarea interfetei `Runnable` permite o flexibilitate sporita în lucrul cu fire de executie.

Varianta 1 (standard)

Crearea firului de executie se realizeaza în constructorul clasei Counter

```
class Counter implements Runnable {
        private Thread counterThread = null;
        private int from, to, step;
        public Counter(int from, int to, int step) {
                this.from = from;
                this.to = to;
                this.step = step;
                if (counterThread == null) {
                        counterThread = new Thread(this);
                        counterThread.start();
                }
        }
        public void run() {
                for(int i = from; i <= to; i += step)
                        System.out.print(i + " " );
        }
}

public class TestThread2 {
        public static void main(String args[]) {
                Counter cnt1, cnt2;
                //lansez primul fir de executie (prin constructor)
                cnt1 = new Counter(0, 100, 5);
                //lansez al doilea fir de executie (prin constructor)
                cnt2 = new Counter(100, 200, 10);
        }
}
```

Varianta 2

Crearea firului de executie se realizeaza în afara clasei Counter:

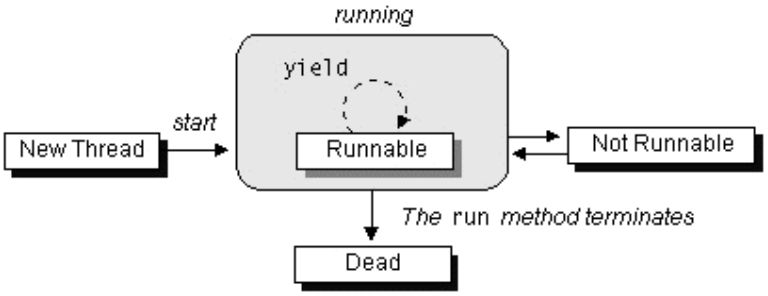
```
class Counter implements Runnable {
        private int from, to, step;
        public Counter(int from, int to, int step) {
                this.from = from;
                this.to = to;
                this.step = step;
        }
        public void run() {
                for(int i = from; i <= to; i += step)
                        System.out.print(i + " " );
        }
}

public class TestThread2 {
        public static void main(String args[]) {
                Counter cnt1, cnt2;
                cnt1 = new Counter(0, 100, 5);
                cnt2 = new Counter(100, 200, 10);
                new Thread( cnt1 ).start();
                //lansez primul fir de executie
                new Thread( cnt2 ).start();
                //lansez al doilea fir de executie
        }
}
```

Ciclul de viata al unui fir de executie

Fiecare fir de executie are propriul sau ciclu de viata : este creat, devine activ prin lansarea sa în executie si, la un moment dat, se termina. In continuare vom vedea mai îndeaproape starile în care se

poate gasi un fir de executie. Diagrama de mai jos ilustreaza generic aceste stari precum si metodele care provoaca tranzitia dintr-o stare în alta:



Asadar, un fir de executie se poate gasi în una din urmatoarele patru stari:

1. **New Thread**
2. **Runnable**
3. **Not Runnable**
4. **Dead**

Starea "New Thread"

Un fir de executie se gaseste în aceasta stare imediat dupa crearea sa, cu alte cuvinte dupa instantierea unui obiect din clasa `Thread` sau dintr-o subclasa a sa.

```
Thread counterThread = new Thread ( this );
//counterThread se gaseste in starea New Thread
```

In aceasta stare firul de executie este "vid", el nu are alocate nici un fel de resurse sistem si singura operatiune pe care o putem executa asupra lui este lansarea în executie, prin metoda `start`. Apelul oricarei alte metode în afara de `start` nu are nici un sens si va provoca o exceptie de tipul `IllegalThreadStateException`.

Starea "Runnable"

Dupa apelul metodei `start` un fir de executie va trece în starea "Runnable", adica se gaseste în executie.

```
counterThread.start();
//counterThread se gaseste in starea Runnable
```

Metoda `start` realizea urmatoarele operatiuni necesare rularii firului de executie:

- aloca resursele sistem necesare
- planifica firul de executie la CPU pentru a fi lansat
- apeleaza metoda run a obiectului reprezentat de firul de executie

Un fir de executie aflat în starea Runnable nu înseamna neaparat ca acesta se gaseste efectiv în executie, adica instructiunile sale sunt interpretate de procesor. Acest lucru se întâmpla din cauza ca majoritatea calculatoarelor au un singur procesor iar acesta nu poate rula simultan toate firele de executie care se gasesc în starea Runnable. Pentru a rezolva aceasta problema interpretorul Java implementeaza o planificare care sa partajeze dinamic si corect procesorul între toate firele de executie care sunt în starea Runnable. Asadar, un fir de executie care "ruleaza" poate sa-si astepte de fapt rândul la procesor.

Starea "Not Runnable"

Un fir de executie ajunge în acesata stare în una din urmatoarele situatii:

- este "adormit" prin apelul metodei **`sleep`**.
- a apelat metoda **`wait`**, asteaptând ca o anumita conditie sa fie satisfacuta
- este blocat într-o operatie de intrare/iesire

"Adormirea" unui fir de executie

Metoda **`sleep`** este o metoda statica a clasei `Thread` care provoaca o pauza în timpul rularii firului curent aflat în executie, cu alte cuvinte îl "adoarme" pentru un timp specificat. Lungimea acestei pauze este specificata în milisecunde si chiar nanosecunde.

```
public static void sleep( long millis )
        throws InterruptedException
public static void sleep( long millis, int nanos )
        throws InterruptedException
```

Intrucât poate provoca exceptii de tipul `InterruptedException` apelul acestei metode se face într-un bloc de tip `try-cacth`:

```
try {
        Thread.sleep(1000);
        //face pauza de o secunda
} catch (InterruptedException e) {
        . . .
}
```

Observati ca metoda fiind statica apelul ei nu se face pentru o instanta anume a clasei `Thread`. Acest lucru este foarte normal deoarece, la un moment dat, un singur fir este în executie si doar pentru acesta are sens "adormirea" sa.
In intervalul în care un fir de executie "doarme", acesta nu va fi execut chiar daca procesorul devine disponibil. Dupa expirarea acestui interval firul revine în starea Runnable, iar daca procesourul este în continuare disponibil îsi contiunua executia.
Pentru fiecare tip de intrare în starea "Not Runnable", exista o secventa specifica de iesire din starea repectiva, care readuce firul de executie în starea Runnable. Acestea sunt:

- Daca un fir de executie a fost "adormit", atunci el devine Runnable doar dupa scurgerea intervalului de timp specificat de instructiunea `sleep`.
- Daca un fir de executie asteapta o anumita conditie, atunci un alt obiect trebuie sa îl informeze daca acea conditie este îndeplinita sau nu; acest lucru se realizeaza prin instructiunile `notify` sau `notifyAll`
  (vezi "Sincronizarea firelor de executie").
- Daca un fir de executie este blocat într-o operatiune de intrare/iesire atunci el redevine Runnable atunci când acea operatiune s-a terminat.

Starea "Dead"

Este starea în care ajunge un fir de executie la terminarea sa. Un fir de executie nu poate fi oprit din program printr-o anumita metoda, ci trebuie sa se termine în mod natural la terminarea metodei `run` pe care o executa. Spre deosebire de versiunile curente ale limbajului Java, în versiunea 1.0 exista metoda `stop` a clasei `Thread` care termina fortat un fir de executie, însa ea a fost eliminata din motive de securitate.
Asadar, un fir de executie trebuie sa-si "aranjeze" singur propria sa "moarte".

Terminarea unui fir de executie

Dupa cum am vazut, un fir de executie nu poate fi terminat fortat de catre program ci trebuie sa-si "aranjeze" singur terminarea sa. Acest lucru poate fi realizat în doua modalitati:

- Prin scrierea unor metode `run` care sa-si termine executia în mod natural; la terminarea metodei `run` se va termina automat si firul de executie, acesta intrând în starea Dead. Acesta este cazul din exemplul considerat anterior:

```
public void run() {
        for(int i = from; i <= to; i += step)
                System.out.print(i + " " );
}
```

  Dupa afisarea numerelor din intervalul specificat metoda se termina si odata cu ea si firul de executie repsectiv.

- Prin folosirea unei variabile de terminare. In cazul când metoda run trebuie sa execute o bucla infinita atunci aceasta trebuie controlata si printr-o variabila care sa opreasca aceasta bucla atunci când dorim ca firul de executie sa se termine. Uzual, aceasta este o variabila membra a clasei care descrie firul de executie care fie este publica, fie este asociata cu o metoda care îi schimba valoarea.

Sa consideram exemplul unui fir de executie care trebuie sa numere secundele scurse pâna la apasarea tastei Enter. Vom scrie mai întâi programul folosind metoda `stop`:

Terminarea unui fir de executie folosind metoda "învechita" stop

```
import java.io.*;
public class TestThread {
        public static void main(String args[]) throws IOException {
                WaitKey thread = new WaitKey();
                thread.start();
                System.in.read();       //astept apasarea tastei Enter
                thread.stop(); //opresc firul de executie
                System.out.println("S-au scurs " + thread.sec + " secunde");
        }
}
class WaitKey extends Thread {
        public int sec = 0;
        public void run() {
                while (true) {
                        try {
                                Thread.sleep(1000);    //pauze de o secunda
                                sec ++;                //s-a mai scurs o secunda
                        } catch(InterruptedException e){}
                }
        }
}
```

Observam ca metoda `run` nu se termina natural, ea ruleaza la infinit asteptând sa fie terminata fortat. Acest lucru l-am realizat aici cu metoda `stop`. Aceasta metoda este însa "învechita" (deprecated) iar la compilarea programului vom obtine un mesaj de avertizare în acest sens. Putem evita metoda `stop` prin folosirea unei variabile de terminare.

Terminarea unui fir de executie folosind o variabila de terminare

```
import java.io.*;
public class TestThread {
        public static void main(String args[]) throws IOException {
                WaitKey thread = new WaitKey();
                thread.start();
                System.in.read();       //astept apasarea tastei Enter
                thread.running = false;
                System.out.println("S-au scurs " + thread.sec + " secunde");
        }
}

class WaitKey extends Thread {
        public int sec = 0;
        public boolean running = true; //variabila de terminare
        public void run() {
                while ( running ) {
                        try {
                                Thread.sleep(1000);
                                sec ++;
                        } catch(InterruptedException e){}
                }
        }
}
```

Metoda isAlive

Aceasta metoda este folosita pentru a vedea daca un fir de executie a fost pornit si nu s-a terminat înca. Metoda returneaza:

- true - daca firul este în una din starile Runnable sau Not Runnable
- false - daca firul este în una din starile New Thread sau Dead

Intre starile Runnable sau Not Runnable, repectiv New Thread sau Dead nu se poate face nici o diferentiere.

```
                WaitKey thread = new WaitKey();
                // isAlive retuneaza false (starea este New Thread)

                thread.start();
                // isAlive retuneaza true (starea este Runnable)

                System.in.read();

                thread.running = false;
                // isAlive retuneaza false (starea este Dead)
```

Nu este necesara distrugerea explicita a unui fir de executie. Sistemul Java de colectare a gunoiului se ocupa de acest lucru. El poate fi fortat sa dezaloce resuresele alocate unui thread prin atribuirea cu null a variabilei care referea instanta firului de executie: `myThread = null .`

Stabilirea prioritatilor de executie

Majoritatea calculatoarelor au un sigur procesor, ceea ce înseamna ca firele de executie trebuie sa-si împarta accesul la acel procesor. Executia într-o anumita ordine a mai multor fire de executie pe un singur procesor se numeste *planificare (scheduling)*. Sistemul Java de executie a programelor implementeaza un algoritm simplu, determinist de planificare, cunoscut sub numele de *planificare cu prioritati fixate*.
Fiecare fir de executie Java primeste la crearea sa o anumita prioritate. O prioritate este de fapt un

numar întreg cu valori cuprinse între MIN_PRIORITY si MAX_PRIORITY. Implicit prioritatea unui fir de executie nou creat are valoarea NORM_PRIORITY. Aceste trei constante sunt definite în clasa `Thread`:

```
public static final int MIN_PRIORITY  - prioritatea minima
public static final int NORM_PRIORITY - prioritatea implicita
public static final int MAX_PRIORITY  - prioritatea maxima
```

Schimbarea ulterioara a prioritatii unui fir de executie se realizeaza cu metoda `setPriority` a clasei `Thread`.
Planificatorul Java lucreaza în modul urmator : daca la un moment dat sunt mai multe fire de executie în starea Runnable, adica sunt pregatite pentru a fi executate, planificatorul îl va alege pe cel cu prioritatea cea mai mare pentru a-l executa. Doar când firul de executie cu prioritate maxima se termina sau este suspendat din diverse motive va fi ales un fir de executie cu o prioritate mai mica. In cazul în care toate firele au aceeasi prioritate ele sunt alese dupa un algoritm simplu de tip "round-robin".
De asemenea, planificarea este complet *preemptiva* : daca un fir cu prioritate mai mare decât firul care se executa la un moment dat solicita procesorul, atunci firul cu prioritate mai mare este imediat trecut în executie iar celalalt trecut în asteptare. Planificatorul Java nu va întrerupe însa un fir de executie în favoarea altuia de aceeasi prioritate, însa acest lucru îl poate face sistemul de operare în cazul în care acesta aloca procesorul în cuante de timp (un astfel de SO este Windows 95/NT).
Asadar, un fir de executie Java cedeaza procesorul în una din situatiile :

- un fir de executie cu o prioritate mai mare solicita procesorul
- metoda sa `run` se termina
- vrea sa faca explicit acest lucru apelând metoda `yield`
- timpul alocat pentru executia sa a expirat (pe SO cu cuante de timp)

In nici un caz corectitudinea unui program nu trebuie sa se bazeze pe mecansimul de planificare a firelor de executie, deoarece acesta poate fi imprevizibil si depinde de la un sistem de operare la altul.

Un fir de executie de lunga durata si care nu cedeaza explicit procesorul la anumite intervale de timp astfel încât sa poata fi executate si celelalte fire de executie se numeste fir de executie *egoist* si trebuie evitata scrierea lor, întrucât acapareaza pe termen nedefinit procesorul, blocând efectiv executia celorlalte fire de executie pâna la terminarea sa. Unele sistemele de operare combat acest tip de comportament prin metoda alocarii procesorului în cuante de timp fiecarui fir de executie, însa nu trebuie sa ne bazam pe acest lucru la scrierea unui program. Un fir de executie trebuie sa fie "corect" fata de celelalte fire si sa cedeze periodic procesorul astfel încât toate sa aiba posibilitatea de a se executa.

Exemplu de fir de executie "egoist"

```
//un fir de executie care numara pana la 100.000 din 100 în 100
class Selfish extends Thread {
        public Selfish(String name) {
                super(name);
        }
        public void run() {
                int i = 0;
                while (i < 100000) {
                        //bucla stransa care acapareaza procesorul
                        i ++;
                        if (i % 100 == 0)
                                System.out.println(getName()+" a ajuns la "+i);
                }
        }
}
```

```
//clasa principala
public class TestSelfishThread {
        public static void main(String args[]) {
                Selfish s1, s2;
                s1 = new Selfish("Firul 1");
                s1.setPriority (Thread.MAX_PRIORITY);
                s2 = new Selfish("Firul 2");
                s2.setPriority (Thread.MAX_PRIORITY);
                s1.start();
                s2.start();
        }
}
```

Firul de executie s1 are prioritate maxima si pâna nu-si va termina executia nu-i va permite firului s2 sa execute nici o instructiune, acaparând efectiv procesorul. Rezultatul va arata astfel:

```
Firul 1 a ajuns la 100
Firul 1 a ajuns la 200
Firul 1 a ajuns la 300
. . .
Firul 1 a ajuns la 99900
Firul 1 a ajuns la 100000
Firul 2 a ajuns la 100
Firul 2 a ajuns la 200
. . .
Firul 2 a ajuns la 99900
Firul 2 a ajuns la 100000
```

Rezolvarea acestei probleme se face fie prin intermediul metodei statice `yield` a clasei `Thread` care determina firul de executie curent sa se opreasca temporar, dând ocazia si altor fire sa se execute, fie prin "adormirea" temporara a firului curent cu ajutorul metodei `sleep`. Metoda `run` a clasei `Selfish` ar trebui rescrisa astfel:

```
        public void run() {
                int i = 0;
                while (i < 100000) {
                        i ++;
                        if (i % 100 == 0)
                                System.out.println(getName()+" a ajuns la "+i);
                        yield(); //cedez temporar procesorul
                }
```

Prin metoda `yield` un fir de executie nu cedeaza procesorul decât firelor de executie care au aceeasi prioritate cu a sa si nu celor cu prioritati mai mici.

Sincronizarea firelor de executie

Pâna acum am vazut cum putem crea fire de executie independente si asincrone, cu alte cuvinte care nu depind în nici un fel de executia sau de rezultatele altor fire de executie. Exista însa numeroase situatii când fire de executie separate, dar care ruleaza concurent, trebuie sa comunice între ele pentru a accesa diferite resurse comune sau pentru a-si transmite dinamic rezultatele "muncii" lor. Cel mai elocvent scenariu în care firele de executie trebuie sa se comunice între ele este cunoscut sub numele de problema *producatorului/consumatorului*, în care producatorul genereaza un flux de date care este preluat si prelucrat de catre consumator.
Sa consideram de exemplu o aplicatie Java în care un fir de executie (producatorul) scrie date într-un fisier în timp ce alt fir de executie (consumatorul) citeste date din acelasi fisier pentru a le prelucra. Sau, sa presupunem ca producatorul genereaza niste numere si le plaseaza, pe rând, într-un buffer iar consumatorul citeste numerele din acel buffer pentru a le interpreta. In ambele cazuri avem de-a face cu fire de executie concurente care folosesc o resursa comuna : un fisier, respectiv un vector si, din acest motiv, ele trebuie sincronizate într-o maniera care sa permita decurgerea normala a activitatii lor.

Scenariul producator / consumator

Pentru a întelege mai bine modalitatea de sincronizare a doua fire de executie sa implementam efectiv o problema de tip producator/consumator.
Sa consideram urmatoarea situatie:

- Producstorul genereaza numerele întregi de la 1 la 10, fiecare la un interval neregulat cuprins între 0 si 100 de milisecunde. Pe masura ce le genereaza încearca sa le plaseze într-o zona de memorie (o variabila întreaga) de unde sa fie citite de catre consumator.
- Consumatorul va prelua, pe rând, numerele generate de catre producator si va afisa valoarea lor pe ecran.

Pentru a fi accesibila ambelor fire de executie, vom încapsula variabila ce va contine numerele generate într-un obiect descris de clasa `Buffer` si care va avea doua metode **`put`** (pentru punerea unui numar în buffer) si **`get`** (pentru obtinerea numarului din buffer).
Fara a folosi nici un mecanism de sincronizare clasa Buffer arata astfel:

```
class Buffer {
        private int number = -1;

        public int get() {
         return number;
        }

        public void put(int number) {
                this.number = number;
        }
}
```

Vom implementa acum clasele Producator si Consumator care vor descrie cele doua fire de executie. Ambele vor avea o referinta comuna la un obiect de tip Buffer prin intermediul caruia îsi comunica valorile.

```
class Producator extends Thread {
        private Buffer buffer;
        public Producator(Buffer b) {
                buffer = b;
        }
        public void run() {
                for (int i = 0; i < 10; i++) {
                        buffer.put(i);
                        System.out.println("Producatorul a pus:\t" + i);
                        try {
                                sleep((int)(Math.random() * 100));
                        } catch (InterruptedException e) { }
                }
        }
}

class Consumator extends Thread {
        private Buffer buffer;
        public Consumator(Buffer b) {
                buffer = b;
        }
        public void run() {
                int value = 0;
                for (int i = 0; i < 10; i++) {
                        value = buffer.get();
                        System.out.println("Consumatorul a primit:\t" + value);
                }
        }
```

```
}
//Clasa principala
public class TestSincronizare1 {
        public static void main(String[] args) {
                Buffer b = new Buffer();
                Producator p1 = new Producator(b);
                Consumator c1 = new Consumator(b);
                p1.start();
                c1.start();
        }
}
```

Dupa cum ne asteptam rezultatul rularii acestui program nu va rezolva fi nici pe departe problema propusa de noi, motivul fiind lipsa oricarei sincronizari între cele doua fire de executie. Mai precis, rezultatul va fi ceva de forma:

```
Consumatorul a primit: -1
Consumatorul a primit: -1
Producatorul a pus:    0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Producatorul a pus:    1
Producatorul a pus:    2
Producatorul a pus:    3
Producatorul a pus:    4
Producatorul a pus:    5
Producatorul a pus:    6
Producatorul a pus:    7
Producatorul a pus:    8
Producatorul a pus:    9
```

Ambele fire de executie acceseaza resursa comuna, adica obiectul de tip Buffer, într-o maniera haotica si acest lucru se întâmpla din dou\ motive :

- consumatorul nu asteapta înainte de a citi ca producatorul sa genereze un numar si va prelua de mai multe ori acelasi numar.
- producatorul nu asteapta consumatorul sa preia numarul generat înainte de a produce un altul, în felul acesta consumatorul va "rata" cu siguranta unele numere (în cazul nostru aproape pe toate).

Problema care se ridica în acest moment este : cine trebuie sa se ocupe de sincronizarea celor doua fire de executie : clasele Producator si Consumator sau resursa comuna Buffer ?
Raspunsul este: resursa comuna Buffer, deoarece ea trebuie sa permita sau nu accesul la continutul sau si nu firele de executie care o folosesc. In felul acesta efortul sincronizarii este transferat de la producator/consumator la un nivel mai jos, cel al resursei critice.
Activitatile producatorului si consumatorului trebuie sincronizate la nivelul resursei comune în doua privinte:

1. Cele doua fire de executie nu trebuie sa acceseze simultan buffer-ul ; acest lucru se realizeaza prin blocarea obiectului Buffer atunci când este accesat de un fir de executie, astfel încât nici nu alt fir de executie sa nu-l mai poate accesa. (vezi "Blocarea unui obiect").
2. Cele doua fire de executie trebuie sa se coordoneze, adica producatorul trebuie sa gaseasca o modalitate de a "spune" consumatorului ca a plasat o valoare în buffer, iar consumatorul trebuie sa comunice producatorului ca a preluat aceasta valoare, pentru ca acesta sa poata genera o alta.

Pentru a realiza aceasta comunicare, clasa Thread pune la dispozitie metodele **`wait, notify, notifyAll`**. (vezi "Metodele wait, notify, notifyAll").

Folosind sincronizarea clasa Buffer va arata astfel:

```
class Buffer {
        private int number = -1;
        private boolean available = false;
        public synchronized int get() {
                while (!available) {
                        try {
                                wait();
                                //asteapta producatorul sa puna o valoare
                        } catch (InterruptedException e) { }
                }
                available = false;
                notifyAll();
                return number;
        }
        public synchronized void put(int number) {
                while (available) {
                        try {
                                wait();
                                //asteapta consumatorul sa preia valoarea
                        } catch (InterruptedException e) { }
                }
                this.number = number;
                available = true;
                notifyAll();
        }
}
```

Rezultatul obtinut va fi cel scontat:

```
Producatorul a pus:    0
Consumatorul a primit: 0
Producatorul a pus:    1
Consumatorul a primit: 1
. . .
Producatorul a pus:    9
Consumatorul a primit: 9
```

Blocarea unui obiect (cuvântul cheie synchronized)

Definitie

Un segment de cod ce gestioneaza o resursa comuna mai multor de fire de executie separate si concurente se numeste *sectiune critica*. In Java o sectiune critica poate fi un bloc de instructiuni sau o metoda.

Controlul accesului într-o sectiune critica se face prin cuvântul cheie **`synchronized`**. Platforma Java asociaza un monitor fiecarui obiect al unui program ce contine sectiuni critice care necesita sincronizare. Acest monitor va indica daca resursa critica este accesata de vreun fir de executie sau este libera, cu alte cuvinte "monitorizeaza" o resursa critica. In cazul în care este accesata, va "pune un lacat" pe aceasta, astfel încât sa împiedice accesul altor fire de executie la ea. In momentul când resursa este eliberata "lacatul" va fi eliminat pentru a permite accesul altor fire de executie.
In exemplul tip producator/consumator de mai sus, sectiunile critice sunt metodele `put` si `get` iar resursa citica comuna este obiectul `buffer`. Consumatorul nu trebuie sa acceseze buffer-ul când producatorul tocmai pune o valoare în el, iar producatorul nu trebuie sa modifice valoarea din buffer în momentul când aceasta este citita de catre consumator.

```
        public synchronized int get() {
```

```
        ...
    }
    public synchronized void put(int number) {
        ...
    }
```

Sa observam ca ambele metode au fost declarate cu modificatorul synchronized. Cu toate acestea sistemul asociaza un monitor unei instante a clasei Buffer si nu unei metode anume. In momentul în este apelata o metoda sincrona firul de executie care a facut apelul va bloca obiectul a carei metoda o acceseaza , ceea ce înseamna ca celelalte fire de executie nu vor mai putea accesa resursele critice, adica nu vor putea apela nici o metoda sincrona din acel obiect. Acesta este un lucru logic, deoarece mai multe sectiuni critice (metode sincrone) ale unui obiect gestioneaza de fapt o singura resursa critica.
In exemplul nostru, atunci când producatorul apeleaza metoda put pentru a scrie un numar, va bloca tot obiectul de tip Buffer, astfel ca firul de executie consumator nu va avea acces la cealalta metoda sincrona get, si reciproc.

```
    public synchronized void put(int number) {
        // buffer blocat de producator
        ...
        // buffer deblocat de producator
    }
    public synchronized int get() {
        // buffer blocat de consumator
        ...
        // buffer deblocat de consumator
    }
```

Metodele wait, notify si notifyAll

Obiectul de tip Buffer din exemplul are o variabila membra privata numita `number`, în care este memorat numarul pe care îl comunica producatorul si din care îl preia consumatorul. De asemenea, mai are o variabila privata logica `available` care ne da starea buffer-ului: daca are valoarea true înseamna ca producatorul a pus o valoare în buffer si consumatorul nu a preluat-o înca; daca este false, consumatorul a preluat valoarea din buffer dar producatorul nu a pus deocamdata alta la loc.
Deci, la prima vedere metodele clasei Buffer ar trebui sa arate astfel:

```
    public synchronized int get() {
        if (available) {
                available = false;
                return number;
        }
    }
    public synchronized int put(int number) {
        if (!available) {
             available = true;
           this.number = number;
        }
    }
```

Implementate ca mai sus cele doua metode nu vor functiona corect Acest lucru se întâmpla deoarece firele de executie, desi îsi sincronizeaza accesul la buffer, nu se "asteapta" unul pe celalalt. Situatiile în care metodele get si put nu fac nimic vor duce la "ratarea" unor numere de catre consumator. Asadar, cele doua fire de executie trebuie sa se astepte unul pe celalalt.

```
    public synchronized int get() {
        while (!available) {
                //nimic - astept ca variabila sa devina true
        }
        available = false;
        return number;
    }
```

```
        public synchronized int put(int number) {
                while (available) {
                        //nimic - astept ca variabila sa devina false
                }
            available = true;
            this.number = number;
        }
```

Varianta de mai sus, desi pare corecta, nu este. Aceasta deoarece implementarea metodelor este "selfish" - cele doua metode îsi asteapta in mod egoist conditia de terminare. Ca urmare, corectitudinea functionarii va depinde de sistemul de operare, ceea ce trprezinta o greseala de programare.

Punerea corecta a unui fir de executie în asteptare se realizeaza cu metoda **`wait`** a clasei `Thread`, care are trei forme:

```
                void wait( )
                void wait( long timeout )
                void wait( long timeout, long nanos )
```

Dupa apelul metodei `wait`, firul de executie curent elibereaza monitorul asociat obiectului respectiv si asteapta ca una din urmatoarele conditii sa fie îndeplinita:

- un alt fir de executie informeaza pe cei care "asteapta" la un anumit monitor sa se trezeasca; acest lucru se realizeaza printr-un apel al metodei **`notifyAll`** sau **`notify`**.
- perioada de astepatare specificata a expirat.

Metoda `wait` poate produce exceptii de tipul `InterruptedException`, atunci când firul de executie care asteapta (este deci în starea Not Runnable) este întrerupt din asteptare si trecut fortat în starea Runnable, desi conditia asteptata nu era înca îndeplinita.
Metoda `notifyAll` informeaza toate firele de executie care sunt în asteptare la monitorul obiectului curent îndeplinirea conditiei pe care o asteptatu. Metoda `notify` informeaza doar un singur fir de executie.

Iata variantele corecte ale metodelor `get` si `put`:

```
                public synchronized int get() {
                        while (!available) {
                                try {
                                        wait();
                                        //asteapta producatorul sa puna o valoare
                                } catch (InterruptedException e) { }
                        }
                        available = false;
                        notifyAll();
                        return number;
                }
        public synchronized void put(int number) {
                        while (available) {
                                try {
                                        wait();
                                        //asteapta consumatorul sa preia valoarea
                                } catch (InterruptedException e) { }
                        }
                        this.number = number;
                        available = true;
                        notifyAll();
                }
        }
```

Gruparea firelor de executie

Curs 8

Gruparea firelor de executie pune la dispozitie un mecanism pentru manipularea acestora ca un tot si nu individual. De exemplu, putem sa pornim sau sa suspendam toate firele dintr-un grup cu un singur apel de metoda. Gruparea firelor de executie se realizeaza prin intermediul clasei **`ThreadGroup`**.

Fiecare fir de executie Java este mebmru al unui grup, indiferent daca specificam explicit acest lucru. Afilierea unui fir de executie la un anumit grup se realizeaza la crearea sa si devine permanenta, în sensul ca nu vom putea muta un fir de executie dintr-un grup în altul, dupa ce acesta a fost creat. In cazul în care cream un fir de executie fara a specifica în constructor din ce grup face parte, el va fi plasat automat în acelasi grup cu firul de executie care l-a creat. La pornirea unui program Java se creeaza automat un obiect de tip `ThreadGroup` cu numele **`main`**, care va reprezenta grupul tuturor firelor de executie create direct din program si care nu au fost atasate explicit altui grup. Cu alte cuvinte, putem sa ignoram complet plasarea firelor de executie în grupuri si sa lasam sistemul sa se ocupe cu aceasta, adunându-le pe toate în grupul `main`.

Exista situatii când programul creeaza multe fire de executie iar gruparea lor poate usura substantial manevrarea lor. Crearea unui fir de executie si plasarea lui într-un grup (altul decât cel implicit) se realizeaa prin urmatorii constructori ai clasei `Thread`:

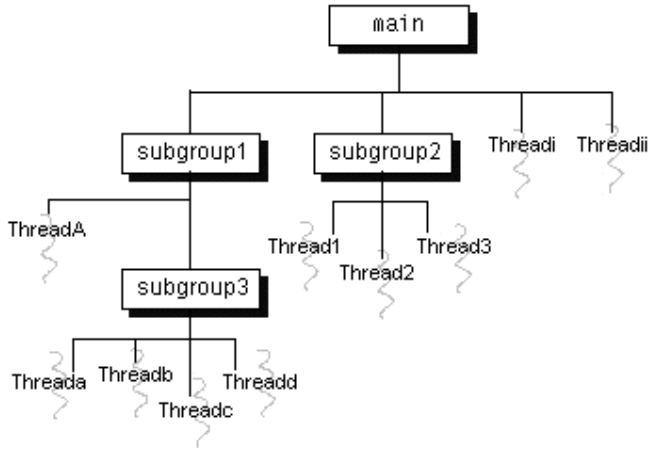
```
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target, String name)
```

Fiecare din acesti costructori creeaza un fir de executie, îl initializeaza si îl plaseaza într-un grup specificat ca argument. In exemplul urmator vor fi create doua grupuri, primul cu doua fire de executie iar al doile cu trei:

```
ThreadGroup grup1 = new ThreadGroup("Producatori");
Thread p1 = new Thread(grup, "Producator 1");
Thread p2 = new Thread(grup, "Producator 2");

ThreadGroup grup2 = new ThreadGroup("Consumatori");
Thread c1 = new Thread(grup, "Consumator 1");
Thread c2 = new Thread(grup, "Consumator 2");
Thread c3 = new Thread(grup, "Consumator 3");
```

Pentru a afla carui grup apartine un anumit fir de executie putem folosi metoda `getThreadGroup` a clasei `Thread`. Un grup poate avea ca parinte un alt grup, ceea ce înseamna ca firele de executie pot fi plasate într-o ierarhie de grupuri, în care radacina este grupul implicit `main`, ca în figura de mai jos:



Exemplu: listarea firelor de executie active

```
public class EnumerateTest {
        public void listCurrentThreads() {
                ThreadGroup currentGroup = Thread.currentThread().getThreadGroup();

                //aflu numarul firelor de executie active
                int numThreads = currentGroup.activeCount();
```

```
                //pun intr-un vector referinte la firele de exec. active
                Thread[] listOfThreads = new Thread[numThreads];
                currentGroup.enumerate(listOfThreads);

                //le afisez pe ecran
                for (int i = 0; i < numThreads; i++)
                        System.out.println("Thread #" + i + " = " +
                                listOfThreads[i].getName());
        }
}
```

Comunicarea prin fluxuri de tip "pipe"

O modalitate deosebit de utila prin care doua fire de executie pot comunica este realizata prin intermediul *canalelor de comunicatii (pipes)*. Acestea sunt implementate prin fluxuri descrise de clasele

`PipedReader, PipedWriter` - pentru caractere, respectiv
`PipedOutputStream, PipedInputStream` - pentru octeti

Constructorii acestor clase sunt :

```
public PipedReader( )
public PipedReader( PipedWriter pw ) throws IOException
public PipedWriter( )
public PipedWriter( PipedReader pr ) throws IOException
```

In cazul în care este folosit constructorul fara argument conectarea unui flux de intrare cu un flux de iesire se face prin metoda **`connect`**:

```
public void connect( PipedWriter pw ) throws IOException
public void connect( PipedReader pr ) throws IOException,
```

Intrucât fluxurile care sunt conectate printr-un pipe trebuie sa execute simultan operatii de scriere/citire folosirea lor se va face în cadrul unor fire de executie.
Functionarea obicetelor care instantiaza `PipedWriter` si `PipedReader` este asemanatoare cu a canalelor UNIX (pipes). Fiecare capat al unui canal este utilizat dintr-un fir de executie separat. La un capat al pipeline-ului se scriu caractere, la celalalt se citesc. La citire, daca nu sunt date disponibile firul de executie se va bloca. Se observa ca acesta este un comportament tipic producator-consumator, firele de executie comunicând printr-un canal.
Realizarea conexiunii se face astfel:

```
                PipedWriter pw1 = new PipedWriter();
                PipedReader pr1 = new PipedReader(pw1);
        sau
                PipedReader pr2 = new PipedReader();
                PipedWriter pw2 = new PipedWriter(pr2);
        sau
                PipedReader pr = new PipedReader();
                PipedWriter pw = new PipedWirter();
                pr.connect(pw) //echivalent cu
                pw.connect(pr);
```

Scrierea si citirea pe/de pe canale se realizeaza prin metodele uzuale `read` si `write` în toate formele lor.

Sa reconsideram acum exemplul producator/consumator folosind canale de comunicatie.
Producatorul trimite datele printr-un flux de iesire de tip `DataOutputStream` catre consumator care le primeste printr-un flux de intrare de tip `DataInputStream`. Aceste doua fluxuri sunt interconectate prin intermediul unor fluxuri de tip "pipe".

```
import java.io.*;

//clasa principala
public class TestPipes {
        public static void main(String[] args) throws IOException {
```

```
                PipedOutputStream pipeOut = new PipedOutputStream();
                PipedInputStream pipeIn = new PipedInputStream(pipeOut);

                DataOutputStream out = new DataOutputStream( pipeOut);
                DataInputStream in = new DataInputStream( pipeIn );

                Producator p1 = new Producator(out);
                Consumator c1 = new Consumator(in);

                p1.start();
                c1.start();
        }
}

class Producator extends Thread {
        private DataOutputStream out;

        public Producator(DataOutputStream out) {
                this.out = out;
        }

        public void run() {
                for (int i = 0; i < 10; i++) {
                        try { out.writeInt(i); }
                        catch (IOException e) {}
                        System.out.println("Producatorul a pus:\t" + i);
                        try {
                                sleep((int)(Math.random() * 100));
                        } catch (InterruptedException e) { }
                }
        }
}

class Consumator extends Thread {
        private DataInputStream in;

        public Consumator(DataInputStream in) {
                this.in = in;
        }
        public void run() {
                int value = 0;
                for (int i = 0; i < 10; i++) {
                        try { value = in.readInt(); }
                        catch (IOException e) {}
                        System.out.println("Consumatorul a primit:\t" + value);
                }
        }
}
```