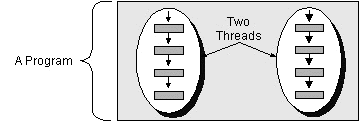
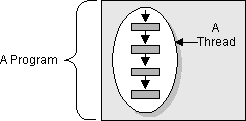
## Capitolul 1

## 1.1 Fire de execuţie

## 

## 1.1.1 Ce este un fir de execuţie ?

Firele de execuţie realizează trecerea de la programarea secvenţială la programarea concurentă. Un program secvenţial reprezintă modelul clasic de program: are un început, o secvenţă de execuţie a instrucţiunilor sale şi un sfârşit. Cu alte cuvinte, la un moment dat programul are un singur punct de execuţie. Un program aflat în execuţie se numeste *proces*. Un sistem de operare monotasking (MS-DOS) execută numai un singur proces la un moment dat, în timp ce un sistem de operare multitasking (UNIX, Windows) poate rula o mulțime de procese în acelaşi timp (concurent), alocând periodic cuante din timpul de lucru al CPU fiecărui proces. Deoarece noţiunea de fir de execuţie nu are sens decât în cadrul unui sistem de operare multitasking. Un fir de execuţie este similar unui proces secvenţial –are un început, o secvenţă de execuţie şi un sfârşit. Diferenţa între un fir de execuţie şi un proces constă în faptul că un fir de execuţie nu poate rula independent, ci trebuie să ruleze în cadrul unui proces.



a)

b)

Figura.1 Fire de execuţie în cadrul unui program

1. Program cu un fir de execuţie;
2. Program cu două fire de execuţie

În Figura. 1 este indicat locul firelor de execuţie în cadrul programului.

***Definiţie :*** Un *fir de execuţie* este o succesiune secvenţială de instrucţiuni care se execută în cadrul unui proces.

Într-un program pot fi definite cîteva fire de execuție, ceea ce înseamnă că în cadrul unui proces se pot executa simultan mai multe fire de execuţie, aceasta permite execuţia concurentă a sarcinilor independente ale acestui program. Un fir de execuţie poate fi asemănat cu o versiune redusă a unui proces, ambele rulând simultan şi independent pe o structură secvenţială de execuţie a instrucţiunilor sale. De asemenea execuţia simultană a firelor de execuţie în cadrul unui proces este similară cu execuţia concurentă a proceselor: sistemul de operare va aloca ciclic cuante din timpul procesorului fiecărui fir de execuţie până la terminarea lor. Din acest motiv firele de execuţie mai sunt numite şi *procese uşoare*. Deosebirea majoră dintre un fir de execuţie şi un proces constă în faptul că firele de execuţie pot rula în cadrul unui proces. O altă deosebire rezultă din faptul că fiecare proces are propria sa memorie (propriul său spaţiu de adrese), iar la crearea unui nou proces (fork) este realizată o copie exactă a procesului părinte : cod + date; la crearea unui fir de execuţie este copiat doar codul procesului părinte; toate firele de execuţie au acces la aceleaşi date, ale procesului original. Astfel, un fir de execuţie poate fi privit şi ca un *context de execuţie* în cadrul unui proces părinte. Firele de execuţie sunt utile în multe privinţe, însă uzual ele sunt folosite pentru executarea unor operaţii consumatoare de timp fără a bloca procesul principal: calcule matematice, așteptarea eliberării unei resurse, acestea realizându-se de obicei pe fundal.

**1.1.2 Concurenţa thread-urilor**

Reieșind din particularitățile hardware, este bine de spus că procesoarele pot executa doar o instrucţiune la un moment dat. Pe o maşină multiprocesor, thread-urile se executa în procesoare diferite în acelaşi timp fizic, chiar dacă procesoarele sunt pe calculatoare diferite conectate într-o reţea. Dar şi pe o maşină cu un singur procesor, thread-urile pot ”împărţi” acelaşi procesor, rulând într-o manieră întreţesută, competiţia pentru timpul CPU crează iluzia că ele se execută simultan. Această iluzie pare reală când 30 de imagini distincte pe secundă captate de ochiul uman sunt percepute într-un flux continuu de imagine.  Comutarea între thread-uri are şi ea un ”preţ”: consumă timp CPU pentru ca acesta să îngheţe starea unui thread şi să dezgheţe starea unui alt thread (schimbare de context). Dacă thread-urile concurente sunt executate pe acelaşi procesor şi toate execută calcule, atunci timpul total de execuţie nu va lua mai mult decât timpul de execuţie al unui program secvenţial care realizează acelaşi lucru. Creşterea vitezei sistemului se realizează prin intersectarea diferitelor faze ale diferitelor thread-urilor. Multe task-uri pot fi segmentate logic în tipuri de faze: faza de calcul şi faza I/O.

Faza de calcul necesită atenţia maxima din partea CPU prin utilizarea diferitelor metode de calcul. Faza de I/O (intrare/ieşire) necesită atenţie maximă din partea perifericelor (imprimantei, hard discului, plăcii de reţea, etc) şi în aceste situaţii procesorul este în general liber, aşteptând ca perifericul să-şi termine sarcina. Creşterea vitezei este obţinută prin intersectarea fazelor. În timp ce un thread se află într-o fază de I/O aşteptând ca o secvenţă de date să fie încărcată de pe hard disk, un thread cu o fază de calcul poate ocupa procesorul şi când ajunge la o fază I/O, celălalt thread (care tocmai a terminat faza I/O proprie) poate începe să utilizeze CPU-ul.

## 

## 1.1.3 Crearea firelor de execuţie

Ca orice alt obiect în Java, un fir de execuţie este o instanţă a unei clase. Firele de execuţie definite de o clasă vor avea acelaşi cod, prin urmare, şi aceeaşi secvenţă de instrucţiuni. Crearea unei clase, care să definească fire de excuţie poate fi făcută prin două modalităţi:

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

Orice clasă ale cărei instanţe vor fi executate într-un fir de execuţie trebuie declarată ca fiind Runnable. Aceasta este o interfaţă care conţine o singură metodă, şi anume metoda ”run”. Asadar, orice clasă ce descrie firele de execuţie va conţine o metodă **run()** în care este implementat codul, ce va fi executat de firul de execuţie. Interfaţa Runnable este concepută ca fiind un protocol comun pentru obiecte, care doresc să execute un cod pe durata existenţei lor (care reprezintă fire de execuţie). Cea mai importantă clasă care implementează interfaţa Runnable este clasa Thread. Clasa Thread implementează un fir de execuţie generic care, implicit, nu face nimic. Cu alte cuvinte, metoda run nu conţine nici un cod. Orice fir de execuţie este o instanţă a clasei Thread sau a unei subclase a acesteia.

### 

### 1.1.3.1 Extinderea clasei Thread

Cea mai simplă metodă de a crea un fir de execuţie, care să realizeze ceva, este extinderea clasei Thread şi supraîncărcarea 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 execuţie

}

}

Prima metodă a clasei este constructorul, care primeşte ca argument un şir ce va reprezenta numele firului de execuţie creat în momentul când constructorul este apelat.

SimpleThread t = new SimpleThread("Java");

//creeaza un fir de execuţie cu numele Java

În cazul în care nu dorim să dam nume firelor de execuţie pe care le creăm, putem renunţa la definirea acestui constructor şi să lasăm doar cu constructorul implicit, fără argumente, care creează un fir de execuţie fără nici un nume. Ulterior acesta poate fi numit utilizînd metoda setName(String).

Se pot defini şi alţi constructori, aceştia fiind utili când vrem să trimitem diverşi parametri firului de execuţie.

A doua metodă este metoda run, "inima" oricărui fir de execuţie în care scriem efectiv codul pe care trebuie să-l execute firul de execuţie. Un fir de execuţie creat nu este automat pornit, lansarea lui în execuţie se realizează prin metoda start, definită de asemenea în clasa Thread.

SimpleThread t = new SimpleThread("Java");

t.start();

//creează şi lansează un fir de execuţie

Să considerăm în continuare un exemplu în care definim un fir de execuţie ce afişează numerele întregi dintr-un interval cu un anumit pas. Firul de execuţie este implementat de clasa Counter.

class Counter extends Thread {

//clasa care defineşte firul de execuţie

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 principală

public static void main(String args[]) {

Counter cnt1, cnt2;

cnt1 = new Counter(0, 10, 2);

//numară de la 0 la 100 cu pasul 5

cnt2 = new Counter(100, 200, 10);

//numară de la 100 la 200 cu pasul 10

cnt1.start();

cnt2.start();

//pornim firele de execuţie

//ele vor fi distruse automat la terminarea lor

}

}

Execuția acestui program in mod secvențial va afişa 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 către contorul cnt1, deci rezultatul afişat pe ecran ar trebui să 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.

În realitate însă, rezultatul obţinut va fi o intercalare de valori produse de cele două fire de execuţie ce rulează simultan. La rulări diferite se pot obţine rezultate diferite deoarece timpul alocat fiecărui fir de execuţie poate să nu fie acelaşi, el fiind controlat de procesor într-o manieră "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…

### 

### 1.1.3.2 Implementarea interfeţei Runnable

Ce facem însă când dorim să creăm o clasă care instanţiază fire de execuţie, iar aceasta are deja o superclasă, ştiind că în Java nu este permisă moştenirea multiplă?

class FirExecuţie extends Părinte, Thread //ilegal !

În acest caz nu mai putem extinde clasa Thread, ci trebuie să implementăm direct în clasa noastră interfaţa Runnable. Clasa Thread implementeaza interfaţa Runnable şi, din acest motiv, la extinderea ei obţinem o implementare implicită a interfeţei. Asadar, interfaţa Runnable permite unei clase să fie active, fără a extinde clasa Thread.

Interfaţa Runnable se găseşte în pachetul java.lang şi este definită astfel:

public interface Runnable {

pulic abstract void run( );

}

Prin urmare, o clasă care instanţiază fire de execuţie prin implementarea interfeţei Runnable trebuie obligatoriu să implementeze metoda run. Formatul general al unei clase care implementează interfaţa 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 execuţie

}

}

Spre deosebire de abordarea anterioară, se pierde tot suportul oferit de clasa Thread, pentu crearea unui fir de execuţie. Simpla instanţiere a unei clase care implementează interfaţa Runnable nu creează nici un fir de execuţie. Din acest motiv crearea firelor de execuţie prin instanţierea unei astfel de clase trebuie făcută explicit. Aceasta se face prin declararea unui obiect de tip Thread, ca variabilă membră a clasei respective. Acest obiect va reprezenta firul de execuţie propriu-zis al cărui cod se găseşte în clasa:

private Thread simpleThread = null;

Următorul pas este instanţierea şi iniţializarea firului de execuţie. Acesta se realizează prin instrucţiunea new, urmată de un apel la constructorul, care primeaște drept argument o instanţă a clasei. După creare, firul de execuţie poate fi lansat printr-un apel la metoda start. (Aceste operaţiuni sunt scrise de obicei în constructorul clasei, pentru a fi executate la iniţializarea unei instanţe, 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, determină crearea unui fir de execuţie, care la lansarea sa va căuta în clasa noastră metoda run şi o va executa. Acest constructor acceptă ca argument orice instanţă a unei clase "Runnable". Asadar, metoda run nu trebuie apelată explicit, cea ce se realizează automat la apelul metodei **start()**.

Apelul explicit al metodei run nu va furniza nici o eroare, însă aceasta va fi executată ca orice altă metodă, deci nu într-un fir de execuţie. Sa rescriem acum exemplul anterior (afişarea numerelor întregi dintr-un interval cu un anumit pas), folosind interfaţa Runnable. Observați că, implementarea interfeţei Runnable permite o flexibilitate sporită în lucrul cu fire de execuţie.

**Varianta 1** (standard)

Crearea firului de execuţie se realizează 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 execuţie (prin constructor)

cnt1 = new Counter(0, 100, 5);

//lansez al doilea fir de execuţie (prin constructor)

cnt2 = new Counter(100, 200, 10);

}

}

**Varianta 2**

Crearea firului de execuţie se realizează 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 execuţie

new Thread( cnt2 ).start();

//lansez al doilea fir de execuţie

}

}

## 

## 1.2. Ciclu de viaţă al unui fir de execuţie

Fiecare fir de execuţie are propriul său ciclu de viaţă: este creat, devine activ prin lansarea sa în execuţie şi, la un moment dat, se sfîrșește. În continuare vom descrie, mai detaliat, stările în care se poate găsi un fir de execuţie. Diagrama ce ilustează generic aceste stări precum și metodele care provoacă tranziția dintr-o stare în alta este ilustrată în figura 2:

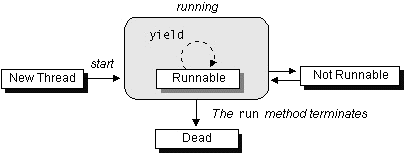


Figura.2 Stările firului de execuţie

Aşadar, un fir de execuţie se poate găsi în una din următoarele patru stări:

#### 1. Starea "New Thread"

Un fir de execuţie se găseşte în această stare imediat după crearea sa, cu alte cuvinte după instanţierea unui obiect din clasa Thread sau dintr-o subclasă a sa.

Thread counterThread = new Thread ( this );

//counterThread se găseşte în starea New Thread

#### 2. Starea "Runnable"

După apelul metodei start un fir de execuţie va trece în starea "Runnable", adică se găseşte în execuţie.

counterThread.start();

//counterThread se găseşte în starea Runnable

Metoda start realizează urmatoarele operaţiuni necesare rulării firului de execuţie: alocă resursele sistem necesare, ordonează firul de execuţie în fir de așteptare la CPU pentru a fi lansat, apelează metoda run a obiectului reprezentat de firul de execuţie.

#### 3. Starea "Not Runnable"

Un fir de execuţie ajunge în această stare în una din urmatoarele situaţii: este "adormit" prin apelul metodei **sleep;** a apelat metoda **wait**, aşteptând ca o anumită condiţie să fie satisfăcută; este blocat într-o operaţie de intrare/ieşire.

#### "Adormirea" unui fir de execuţie

Metoda **sleep** este o metodă statică a clasei Thread care provoacă o pauză în timpul rulării firului curent aflat în execuţie, cu alte cuvinte îl "adoarme" pentru un timp specificat. Lungimea acestei pauze este specificată în milisecunde şi chiar nanosecunde.

public static void sleep( long millis )

throws InterruptedException

public static void sleep( long millis, int nanos )

throws InterruptedException

Întrucât poate provoca excepţii de tipul InterruptedException apelul acestei metode se face într-un bloc de tip try-catch:

try {

Thread.sleep(1000);

//face pauză de o secundă

} catch (InterruptedException e) {

. . .

}.

4. Starea "Dead"

Este starea în care ajunge un fir de execuţie la terminarea sa. Un fir de execuţie nu poate fi oprit din program printr-o anumită metodă, ci trebuie să se termine în mod natural la terminarea metodei run pe care o execută. Terminarea unui fir de execuţie folosind o variabilă 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 apăsarea 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){

}}}}

##### Lucrare de laborator nr. 1

* 1. **Tema lucrării:**Crearea thread-urilor
  2. **Scopul lucrării:** Însuşirea modalităţilor de creare a thread-urlilor în Java;
  3. **Etapele de realizare:**

1. Utilizare clasei Threadpentru crearea unei clase noi;
2. Utilizarea interfeţei Runnable pentru crearea şi lansarea de thread-uri;
3. Prezentarea lucrării.
4. **Exemplu de realizare:**

import java.util.\*;

class Counter1 extends Thread

{

private int from, to, step;

private int[] tablou;

public Counter1(int from, int to, int step, int[] tablou) {

this.from = from;

this.to = to;

this.step = step;

this.tablou = tablou;

}

public void run() {

int s1=0, s2=0, s=0;

int i=from;

while(i!=to){

if(tablou[i]<=50){

s1=i;

i+=step;

do{

if(tablou[i]<=50){

s2=i;

s=s1+s2;

System.out.println(getName()+" " + s1+ " " + s2 +" "+s+ " "+ tablou[s1]+" "+tablou[s2] );

break;

}

i+=step;

}while(true);

// i+=step;

}

i+=step;

}

}

}

public class Main {

public static void main(String args[])

{

Counter1 cnt1, cnt2;

int[] tablou = new int[101];

for(int i=0; i<100; i++){

tablou[i] = (int)(Math.random()\*99);

System.out.print(tablou[i]+" ");

}

System.out.println(" ");

cnt1 = new Counter1(0, 99, 1, tablou);

cnt2 = new Counter1(99, 0, -1, tablou);

cnt1.start();

cnt1.setName("Unu");

cnt2.start();

cnt2.setName("Doi");

}

}

**Rezultatul realuzării:**

65 90 74 92 40 41 53 27 88 34 52 70 37 17 0 75 42 56 90 46 37 11 50 77 81 20 49 19 86 65 36 17 46 31 14 87 34 72 29 93 91 84 1 47 98 22 45 41 57 84 70 44 23 12 47 46 6 4 56 81 2 48 11 42 80 54 51 86 16 67 69 36 59 90 9 84 94 69 41 88 27 57 45 33 84 36 10 60 68 70 54 18 26 75 18 81 68 81 80 22

Doi 99 94 193 22 18

Unu 4 5 9 40 41

Doi 92 91 183 26 18

Unu 7 9 16 27 34

Doi 86 85 171 10 36

Unu 12 13 25 37 17

Doi 83 82 165 33 45

Unu 14 16 30 0 42

Doi 80 78 158 27 41

Unu 19 20 39 46 37

Doi 74 71 145 9 36

Unu 21 22 43 11 50

Doi 68 63 131 16 42

Unu 25 26 51 20 49

Unu 27 30 57 19 36

Unu 31 32 63 17 46

Doi 62 61 123 11 48

Unu 33 34 67 31 14

Doi 60 57 117 2 4

Unu 36 38 74 34 29

Doi 56 55 111 6 46

Unu 42 43 85 1 47

Doi 54 53 107 47 12

Unu 45 46 91 22 45

Doi 52 51 103 23 44

Unu 47 51 98 41 44

Doi 47 46 93 41 45

Unu 52 53 105 23 12

Unu 54 55 109 47 46

Doi 45 43 88 22 47

Unu 56 57 113 6 4

Doi 42 38 80 1 29

Unu 60 61 121 2 48

Doi 36 34 70 34 14

Unu 62 63 125 11 42

Doi 33 32 65 31 46

Unu 68 71 139 16 36

Doi 31 30 61 17 36

Unu 74 78 152 9 41

Doi 27 26 53 19 49

Unu 80 82 162 27 45

Doi 25 22 47 20 50

Doi 21 20 41 11 37

Unu 83 85 168 33 36

Doi 19 16 35 46 42

Unu 86 91 177 10 18

Doi 14 13 27 0 17

Unu 92 94 186 26 18

Doi 12 9 21 37 34

Doi 7 5 12 27 41

BUILD SUCCESSFUL (total time: 1 second)

1. **Sarcinile propuse spre realizare:**

Problema:Scrieţi un program care creează doua fire de execuţie. Ambele fire vor citi datele din acelaşi tablou de date mas[] de tipul int, generat aleatoriu cu dimensiunea 100 şi ce cuprinde valori intre 1 şi 100. Primul fir Th1 va afişa: Condiţie 1 din tabelul 1. Al doilea fir Th2 va afişa: Condiţie 2 din tabelul 1. Apoi după terminarea ambelor fire de execuţie thread-ul principal va afişa informaţia despre studentul care a efectuat lucrarea dată de laborator, literele textului vor apărea pe ecran cu un interval de 100 milisecunde.

Tabelul 1 Condiţiile pentru realizarea sarcinii problemei conform variantelor

|  |  |  |
| --- | --- | --- |
| **Nr/var** | **Condiţia 1** | **Condiţia 2** |
| 1 | Sumelenumerelor pare douăcâtedouăîncepândcăutareaşisumarea de la primul element | Sumelenumerelor pare douăcâtedouăîncepândcăutareaşisumarea de la ultimul element |
| 2 | Suma poziţiilornumerelor pare douăcâtedouăîncepândcău-tareaşisumarea de la primul element | Sumelepoziţiilornumerelor pare douăcâtedouăîncepândcăutareaşisumarea de la ultimul element |
| 3 | Sumelenumerelor pare două cate douăîncepândcăutareaşisumarea de la primul element | Sumelenumerelor pare două cate douăîncepândcăutareaşisumarea de la ultimul element |
| 4 | Sumelepoziţiilornumerelor pare douăcâtedouăîncepândcăutareaşisumarea de la primul element | Sumelepoziţiilornumerelor pare douăcâtedouăîncepândcăutareaşisumarea de la ultimul element |
| 5 | Sumeleproduselornumerelor de pepoziţii pare douăcâtedouăîncepând cu primul element | Sumeleproduselornumerelor de pepoziţii pare douăcâtedouăîncepând cu ultimul element |
| 6 | Sumeleproduselornumerelor de pepoziţiiimparedouăcâtedouăîncepând cu primul element | Sumeleproduselornumerelor de pepoziţiiimparedouăcâtedouăîncepând cu ultimul element |
| 7 | Sumeleproduselornumerelor pare douăcâtedouăîncepând cu primul element | Sumeleproduselornumerelor pare douăcâtedouăîncepând cu ultimul element |
| 8 | Sumeleproduselornumerelorimparedouăcâtedouăîncepând cu primul element | Sumeleproduselornumerelorimparedouăcâtedouăîncepând cu ultimul element |
| 9 | Sumelenumerelor de peprimeletreipoziţii pareîncepând cu primul element | Sumelenumerelor de pepoziţii parecâtetreiîncepând cu ultimul element |
| 10 | Smeleproduselornume-relor de pepoziţiiimparecâtedouăîncepând cu primul element | Sumeleproduselornumerelor de pepoziţiiimparedouă cate douăîncepând cu primul element |

1. **Întrebări de verificare:**
   1. Definiți un fir de execuție?
   2. Numiți modalitățile de creare a firelor de execuție?
   3. Cum poate fi definită și implementate interfața Runnable?
   4. Numiți stările unui fir de execuție?
   5. Pentru ce se folosește metoda setNume(String)?
   6. Explicați noțiunea de procese concurente?