Fire de executie

https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Ce sunt firele de executie?

O aplicatie Java ruleaza in interiorul unui proces al sistemului de operare. Procesul consta in segmente de cod si segmente de date mapate intr-un spatiu virtual de adresare. Fiecare proces detine un numar de resurse alocate de catre sistemul de operare, cum ar fi fisiere deschise, zone de memorie alocate dinamic sau fire de executie. Resursele alocate procesului sunt eliberate la terminarea executiei procesului.

Un fir de executie este unitatea de executie a unui proces. Fiecarui fir de executie i se asociaza o secventa de instructiuni, un set de registri CPU si o stiva. Procesul nu executa instructiuni, este un spatiu de adresare comun pentru unul sau mai multe fire de executie. Firele de executie sunt cele care executa instructiunile.

Firele de executie seamana cu procesele, pot fi la fel planificate pentru executie. Principala diferenta este ca firul se executa in spatiul de adresare al procesului caruia apartine si poate modifica valori care sunt vazute si de celelalte fire care apartin aceluiasi proces. Din aceasta cauza apare necesitatea ca firele sa comunice intre ele, adica trebuie sincronizat accesul la datele utilizate in comun. Sincronizarea asigura siguranta datelor, adica prin sincronizare se previn situatile ca un fir sa modifice o variabila care este tocmai utilazata de catre un alt fir de executie.

Fir de executie = Secventa de instructiuni + un set de registi CPU + o stiva

Java fiind un limbaj interpretat, procesul detine codul interpretorului, iar codul binar Java (bytecode) este tratat ca o zona de date de catre interpretor. Deci firele de excutie sunt create de fapt de catre interpretorul Java. La lansarea in executie a unei aplicatii Java este creat si automat un prim fir de executie, numit firul principal. Acesta poate sa creeze alte fire de executie, care la randul lor pot crea alte fire, si asa mai departe.

De ce avem nevoie de fire de executie?

Aplicatiile care utilizeaza mai multe fire de executie pot executa in paralel mai multe sarcini. De exemplu o aplicatie care realizeaza o animatie intr-o fereastra, iar intr-o alta fereastra afiseaza rezultatul unei interogari de baza de date.
Un program de navigare realizeaza cel putin doua lucruri in paralel: aduce date din retea si paralel afiseaza aceste date. Firele de executie faciliteaza ca programarea sa fie mai apropiata de gandirea umana. Omul de obicei se ocupa de mai multe lucruri in acelasi timp. Un profesor in timp ce preda trebuie sa fie atent si la studentii care il asculta.

Firele de executie Java ofera o serie de facilitati programatorilor, dar exista si o serie de diferente fata de alte pachete pentru fire de executie. Comparand cu aceste pachete putem considera ca ne ofera mai putin, dar au si un avantaj, simplitatea. Exista si un standard POSIX P10033.4a un pachet de fire de excutie, care poate fi implementat in orice sistem de operare.

Orice pachet Java (de la Sun) este "Thread safe", adica este asigurat ca metodele continute in pachet pot fi apelate simultan de mai multe fire de executie.

Utilizarea firelor de executie in Java

1. Creare si lansare

In Java avem doua posibilitati de a crea un fir de executie:

- Prin definirea unei clase care mosteneste de la clasa predefinita Thread (deriva din clasa Thread)
- Prin implementarea interfetei Runnable

Clasa ***Thread*** este definita in pachetul java.lang avand o serie de metode. Principala metoda este metoda ***run()***. Aceasta trebuie sa contina toate activitatile pe care firul trebuie sa le execute.

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

Clasa Thread este o clasa care are ca si superclasa clasa Object si implementeaza interfata Runnable. Interfata Runnable declara o singura metoda run() si clasa Thread implementeaza aceasta. Masina virtuala Java permite unei aplicatii sa ruleze mai multe fire de executie in paralel. Fiecare fir de executie are o prioritate si fiecare fir poate fi marcat ca si fir demon. In momentul crearii unui fir de executie se seteaza si proprietatile acestea, adica firul nou creat va avea aceasi prioritate ca si firul parinte si va fi de tip demon numai daca firul parinte este demon.
Cand se interpreteaza codul octeti al unei aplicatii, masina virtuala Java creaza un prim fir de executie care ruleaza metoda main. Masina virtuala Java continua sa execute acest fir sau alte fire noi create pana cand toate firele de executie nedemon sunt distruse.

Thread
`start()`
`run()`
`sleep()`
`interrupt()`
`join()`
`getName()`
`setName()`
`destroy()`
`isinterrupted()`

Metoda *start()* este utilizata pentru lansarea in executie a unui fir nou creat. Metoda start() se utilizeaza o singura data in ciclul de viata al unui fir . Firul se distruge cu metoda destroy*()*, si aceasta putand fi apelat o singura data. Metoda *sleep()* se apeleaza cu un argument care reprezinta timpul in milisecunde in care firul de executie asteapta(adoarme). *sleep()* este o metoda statica se apeleaza prefixat cu numele clasei. In caz ca firul este intrerupt se genereaza exceptia *InterruptedException*. Deci apelul metodei trebuie facut intr-un context de tratarea acestei exceptii.

```
try{
    Thread.sleep( 1000 )
}
catch ( InterruptedException e ){
    //Codul  de tratarea a exceptiei
}
```

Firul poate fi scos din starea de adormire prin apelul metodei *interrupt()*. Metoda se mai apeleaza pentru a

intrerupe executia unui fir care este blocat in asteptarea unei operatii lungi de intrare/iesire. Metoda *join()* se utilizeza pentru legarea firelor de executie. Prin *getName()* si *setName()* putem obtine sau sa setam numele firului.

Diagrama de clase:

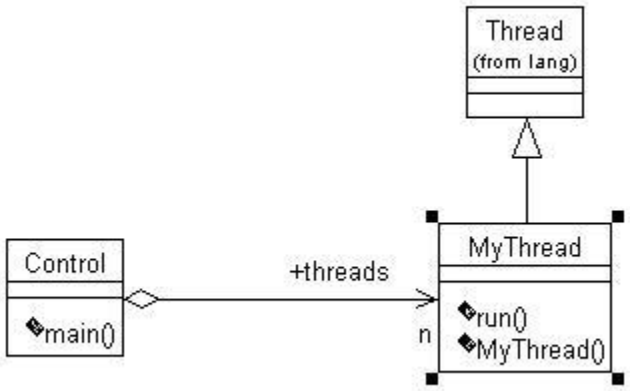


Diagrama de colaborare:

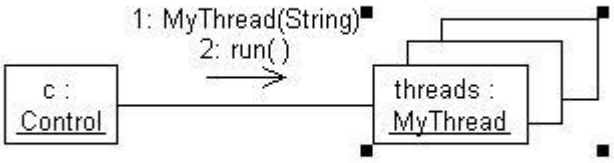


Figura precedenta prezinta proiectul unei aplicatii, care creaza doua fire de executie si le starteaza, neasteptand terminarea acestora. Firele vor afisa numele lor pana cand sunt oprite. Firele pot fi oprite cu combinatia de taste Ctrl-C.

Crearea unui obiect din clasa MyThread nu inseamna si inceperea executiei firului creat. Firul poate fi lansat in executie prin apelul metodei ***start()***. Metoda ***start()*** apeleaza metoda ***run()***, care executa instructiunile proprii firului.

Sursa Java pentru exemplul precedent:

```
//Source file: MyThread.java

public class MyThread extends Thread
{
   public MyThread()
   {
   }

   public void run()
   {
    while( true )
      try{
           System.out.println(" Se ruleaza "+getName());
           sleep( 1000 );
      }
      catch( InterruptedException e ){}
   }
```

```
   public MyThread(String name)
   {
    super( name );
   }
}

//Source file: Control.java

public class Control
{
   public MyThread threads[];

   public Control()
   {
    threads = new MyThread[ 2 ];
    threads[ 0 ] = new MyThread( "Fir 1");
    threads[ 1 ] = new MyThread( "Fir 2");
    threads[ 0 ].start();
    threads[ 1 ].start();
   }

   public static void main(String[] args)
   {
    Control c = new Control();
   }
}
```

A doua modalitate de crearea unui fir de executie se utilizeaza cand clasa noua creata trebuie sa mosteneasca de la doua clase. Deoarece Java nu permite mostenire multipla, se incearca simularea acestuia prin interfete. Interfata este o declaratie care contine o serie de metode abstracte care trebuie implementate de catre clasa care implementeaza interfata.
Interfata Runnable este:

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

In acest caz, clasa noua nu mosteneste metodele clasei Thread (start(), stop() etc.) deci crearea firului se va face prin crearea unui obiect de tip Thread care primeste ca parametru un obiect de tip Runnable. (Clasa Thread are si un asemenea constructor).

Diagrama de clase:

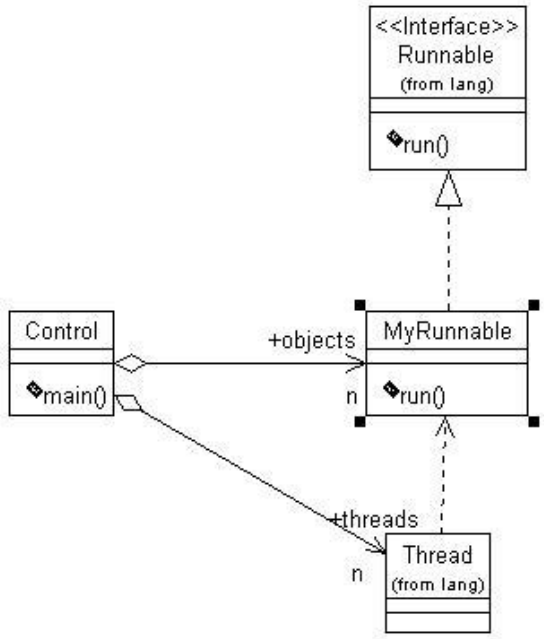
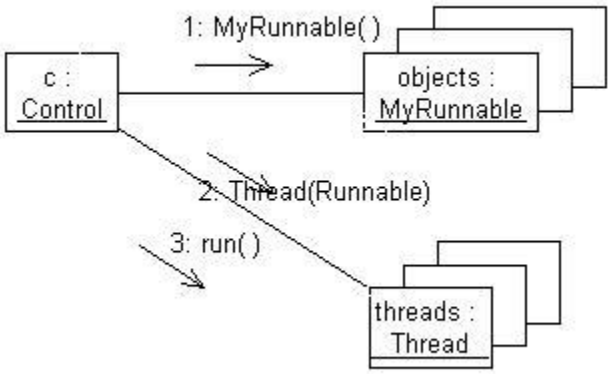


Diagrama de colaborare:



Sursa Java:

//Source file: Control.java

```
public class Control
{
   public MyRunnable objects[];
   public Thread threads[];

   public Control()
   {
    objects = new MyRunnable[ 2 ];
    objects[ 0 ] = new MyRunnable();
    objects[ 1 ] = new MyRunnable();
    threads = new Thread[ 2 ];
    threads[ 0 ] = new Thread(objects[ 0 ]);
    threads[ 1 ] = new Thread(objects[ 1 ]);
    threads[ 0 ].start( );
    threads[ 1 ].start( );
   }
```

```
   public static void main(String[] args)
   {
    Control c = new Control();
   }
}
```

```
//Source file: MyRunnable.java

public class MyRunnable implements Runnable
{
   public MyRunnable()
   {
   }

   public void run()
   {
    while( true )
    {
      System.out.println( "Se ruleaza "+(Thread.currentThread()).getName() );
      try{
       Thread.sleep( 1000 );
      }
      catch( InterruptedException e ){}
     }
   }
}
```

2. Legarea firelor de executie

Legarea firelor de executie permite unui fir sa astepte ca un alt fir sa-si termine executia. De exemplu un fir trebuie sa utilizeze date care sunt calculate de un alt fir de executie, atunci in momentul cand are nevoie de aceste date isi intrerupe executia asteptand firul care ii va furniza acestea. Pentru legarea firelor se utilizeaza metoda join(). Metoda are doua forma de apel:

- `join( int timp )` parametrul reprezinta timpul de asteptare (in milisecunde), adica cat se asteapta pentru terminarea firului,
- `join()` se asteapta terminarea firului indiferent cat trebuie asteptat

Exemplul urmator creaza doua fire de excutie. Unul dintre fire afiseaza un text si asteapta 500 de milisecunde de cinci ori. Celalalt fir isi afiseaza numele dupa care asteapta pe celalalt fir ca acesta sa-si termine executia.

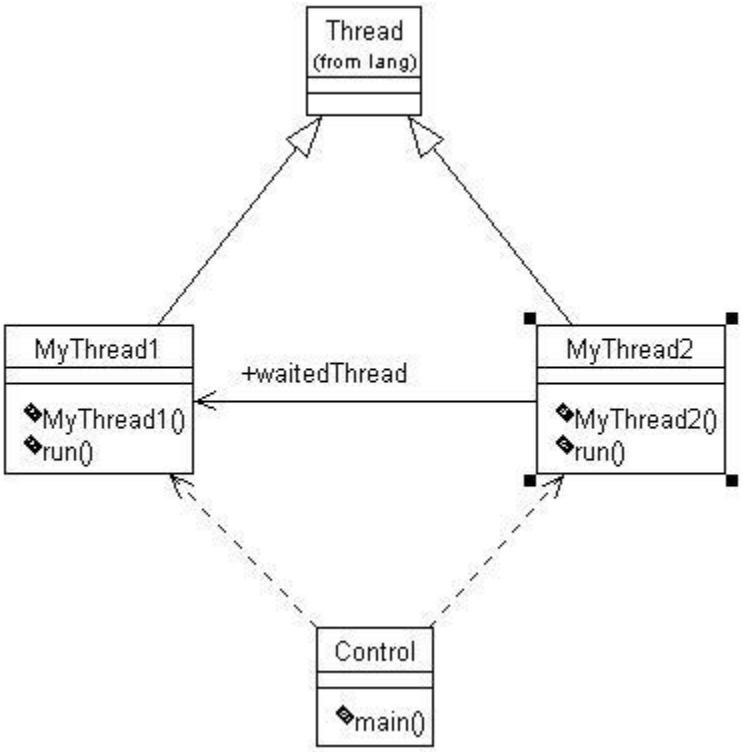
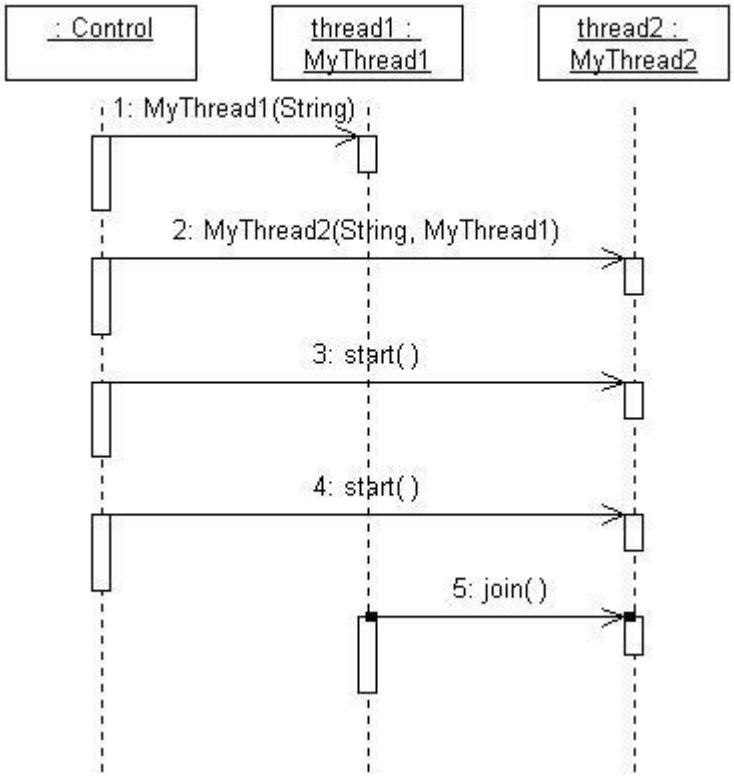


Diagrama de secventiere:



Sursa Java:

//Source file: MyThread1.java

```
public class MyThread1 extends Thread
{
```

```
   public MyThread1(String name)
   {
    super( name);
   }

   public void run()
   {
     System.out.println(getName() + "is running");
        for(int i=0;i<5;i++){
         try{
                     sleep( 500 )  ;
            }
            catch( InterruptedException e ){}
            System.out.println(getName()+" writes "+Integer.toString(i));
       }
   }
}
```

//Source file: MyThread2.java

```
public class MyThread2 extends Thread
{
   public MyThread1 waitedThread;


   public MyThread2(String name, MyThread1 waitedThread)
   {
    super( name );
    this.waitedThread = waitedThread;
   }

   public void run()
   {
     System.out.println(getName()+" waits for Thread "+waitedThread.getName());
     try{
                 waitedThread.join();
     }
     catch( InterruptedException e ){}
     System.out.println(waitedThread.getName()+" has been finished");
     System.out.println(getName()+" has been finished");
   }
}
```

//Source file: Control.java

```
public class Control
{

   public static void main(java.lang.String args[])
   {
     MyThread1 thread1 = new MyThread1("First Thread");
     MyThread2 thread2 = new MyThread2("Second Thread",thread1);
     thread1.start();
     thread2.start();
   }
}
```

3. Fire de executie demoni

Caracteristici:

- Sunt fire speciale similare cu procesele demoni.
- Realizeaza anumite activitati in fundal (background)
- Se distrug automat la terminarea celorlaltor fire de executie.
- Au prioritate de executie redusa, fiind planificate la CPU cand acesta nu ruleaza alte fire

Colectarea gunoiul in Java se relueza pe un fir demon. Firul poate fi transformat in fir demon prin apelul metodei *setDaemon()*.

Exemplul urmator ar trebui sa ne arate ca firele demoni sunt planificate pentru executie mai rar decat cele normale.

```
  class  MyThread extends Thread{
      public MyThread( String name, boolean is_Daemon ){
          super(name);
          setDaemon(is_Daemon);
      }
      public void run(){
        for(int i=0;i<5;i++){
              try{
                  sleep( 500 )  ;
              }
              catch( InterruptedException e ){}
              System.out.println(getName() + "se ruleaza");
        }
      }

}

public class Control{
    public static void main( String args[] )
    {
        MyThread t1,t2;
        t1 = new MyThread("Normal", false );
        t2 = new MyThread("Demon ", true  );
        t1.start();
        t2.start();
    }
}
```

4. Grupe de fire de executie

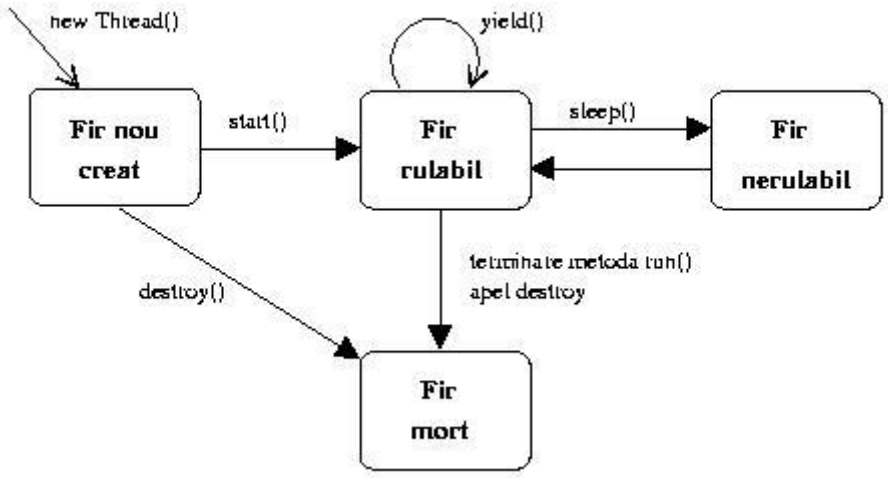
La crearea unui fir nou de executie, firul nou creat putem sa-l adaugam la un grup de fire de executie existent, sau putem lasa in seama interpretorului Java la care grup sa-l adauge.
In exemplul demonstrativ vom utiliza metoda `yield()` pentru a ceda unitatea centrala altui fir de executie (astfel firul care cedeaza trece in coada de asteptarea a firelor gata de executie).

```
class MyThread extends Thread{
     MyThread( String nume ){
         super( nume );
     }
     public void run(){
         while( true ){
             if (isInterrupted() )
```

```
                    break;
                yield();
            }
        }
    }


public class Control{
    static public void main( String args[] ){
        MyThread a, b, c;
        ThreadGroup group;
        // Obtinerea unei referinte la grup
        group = Thread.currentThread().getThreadGroup();
        System.out.println(" Nume grup: "+group.getName());
        System.out.println(" Nr. fire active: "+group.activeCount());
        a = new MyThread("Fir 1 ");
        b = new MyThread("Fir 2 ");
        c = new MyThread("Fir 3 ");
        a.start(); b.start(); c.start();
        System.out.println(" Nr. fire active: "+group.activeCount());
        a.interrupt();
        b.interrupt();
        c.interrupt();
    }
}
```

5. Starile unui fir de executie



Fir "**rulabil**" (Runnable): se ajunge in aceasta stare prin apelul metodei *start()*. Firul trece in stare de "**nerulabil**" (Not Runnable) prin apelul metodei *sleep()* , *wait()* sau daca firul se blocheaza intr-o operatie de citire. Dupa terminarea asteptarii firul revine in starea "rulabil".
Firul ajunge in starea "**mort**", dupa ce se termina executia metodei *run()* sau daca se apeleaza metoda *destroy().*

Firele sunt planificate pentru executie la fel ca si procesele, utilizand un algoritm adecvat pentru planificarea lor. Daca masina virtuala se ruleaza pe un sistem de oprare care lucreaza cu fire de executie, atunci s-ar putea ca fiecare fir de executie creat de catre masina virtuala Java sa fie executat pe un fir nativ.

6. Prioritatea firelor de executie

Masina virtuala Java utilizeaza prioritatile firelor in planificarea firelor pentru executie. In Java sunt 10 niveluri de prioritate. Avem si trei constante in clasa Thread ce pot fi utilizate:

Thread.MIN_PRIORITY avand valoarea 1
Thread.MAX_PRIORITY avand valoarea 10
Thread.NORM_PRIORITY avand valoarea 5

Prioritatea firului se seteaza cu metoda setPriority()

7. Sincronizarea firelor de executie.

Fiecare fir de executie are o viata proprie si nu este interesat de ceea ce fac celelalte fire de executie. Daca calculatorul este dotat cu mai multe procesoare, atunci diferitele fire de executie ale aceluiasi proces pot fi planificate pentru executie pe diferite procesoare. Pentru cel care proiecteaza aplicatia, acest lucru este nesemnificativ, deoarece toate aceste lucruri cad in sarcina masinii virtuale si nu in sarcina programatorului. Totusi apare o problema in cazul aplicatiilor care se ruleaza pe mai multe fire si anume accesul resurselor comune. Ca sa nu apara probleme, accesul la resursele utilizate in comun trebuie sincronizat. Sincronizarea se bazeaza pe conceptul de **monitor** introdus de catre C.A. Hoare. Un monitor este de fapt un lacat atasat unei resurse pentru a preveni utilizarea resursei in paralel.

Un fir de executie ocupa un monitor daca apeleaza o metoda sincronizata. Daca un fir ocupa un monitor, un alt fir care incearca ocuparea aceluiasi monitor, este blocat si asteapta pana la eliberarea monitorului.
Oricare obiect, care contine unul sau mai multe metode sincronizate, are si un monitor atasat. Metodele sincronizate se definesc in modul urmator:

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

Orice clasa Java are un monitor atasat. Acest monitor se deosebeste de monitoarele atasate obiectelor, se utilizeaza cand se apeleaza metodele statice sincronizate ale clasei.

```
public static synchronized void my_static_method() {...}
```

wait() si notify()

Cu ajutorul cuvantului-cheie `synchronized` putem serializa executia anumitor metode. Metodele `wait()` si `notify()` ale clasei Object extind aceasta capabilitate. Utilizand `wait()` si `notify()`, un fir de executie poate elibera monitorul ocupat, asteptand un semnal pentru reocuparea acestuia. Metodele pot fi utilizate doar in metode sincronizate sau in blocuri de instructiuni sincronizate. Executand `wait()` intr-un bloc sincronizat, firul elibereaza monitorul si adoarme. De obicei firul recurge la aceasta posibilitate cand trebuie sa astepte aparitia unui eveniment intr-o alta parte a aplicatiei. Mai tarziu, cand apare evenimentul, firul, in care a aparut evenimentul, apeleaza `notify()` pentru a trezi firul adormit. Firul trezit ocupa din nou monitorul si isi continua activitatea din punctul de intrerupere.

```
class MyClass{
    public synchronized void method1(){}
    public synchronized void method2(){}
    public static synchronized void method3(){}
}
...
MyClass object1 = new MyClass(); // Se ataseaza un monitor obiect pentru sincronizarea
accesului la metodele method1() si method2() pentru firele de executie al obiectului object1
```

```
MyClass object2 = new MyClass(); // Se ataseaza un monitor obiect pentru sincronizarea
accesului la metodele method1() si method2() pentru firele de executie al obiectului object2
```

Pentru exemplul precedent se vor crea trei monitoare. Un monitor de clasa, care este responsabil pentru sincronizarea accesului la metoda statica (metoda clasei) method3() si doua monitoare obiect, cate unul pentru fiecare obiect declarat pentru sincronizarea accesului la method1(0 si method2()

Metoda *notify()* anunta intotdeauna un singur fir de executie care asteapta pentru a accesa monitorul. Exista si o alta metoda *notifyAll()* care se utilizeaza atunci cand mai multe fire de executie concureaza pentru acelasi monitor. In acest caz fiecare fir este instiintat si ele vor concura pentru obtinerea monitorului. Metodele *wait(), notify()* si *notifyAll()* se utilizeaza doar in metode sincronizate, altfel mediul de executi arunca exceptia *IllegalMonitorStateException*. Exemplul urmator se compileaza fara erori, dar se arunca aceasta exceptie pe parcursul executiei.

```
public class test{
 public test(){
     try{
          wait();
     }
     catch( InterruptedException e ){}
 }

 public static void main( String args[] ){
  test t = new test();
 }
}
```

8. Problema producatorului si a consumatorului

Specificatia problemei

Intr-un birou lucreaza mai multi functionari, fiecare elaborand documente pe cate un calculator legate intr-o retea. La aceeasi retea sunt legate si cateva imprimante, fiecare avand acces la toate imprimantele conectate. Sa se simuleze functionarea biroului stiind ca nu toti functionarii elaboreaza documentele in acelasi ritm si functionarii selecteaza aleator imprimantele pe care vor tipari documentele.

Rezolvarea problemei

Perspectiva conceptuala

Se vede clar ca problema se reduce la celebra problema a producatorului si a consumatorului, in acest caz avand mai multi
producatori si mai multi consumatori. Fiecare functionar poate fi privit ca un producator si fiecare imprimanta ca un consumator.
Fiecare imprimanta are atasat o coada de asteptare (buffer) unde sunt gestionate documentele inainte de a fi tiparite.
Trecem la identificarea obiectelor (claselor). Principalele clase vor fi urmatoarele:

- Producer (Functionar)
- Consumer (Imprimanta)
- Buffer (Coada de asteptare a imprimantei)

Pentru a simula paralelismul vom lucra cu fire de executie, adica fiecare functionar si imprimanta va fi implementat ca un fir de
executie. Evident apare si problema sincronizarii la cozile de asteptare atasate imprimantelor.

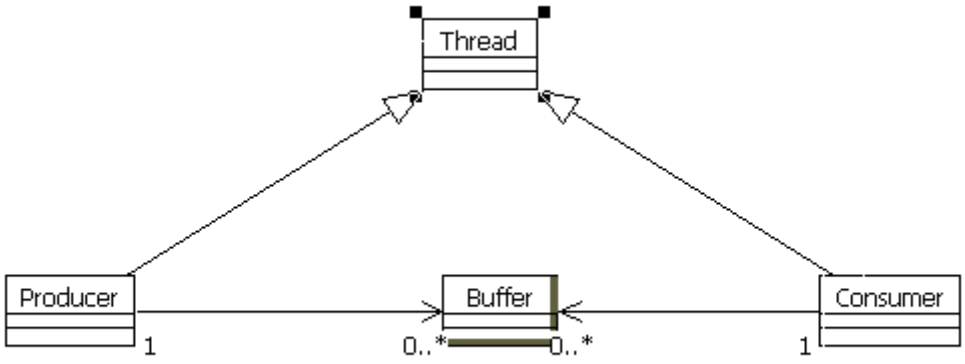
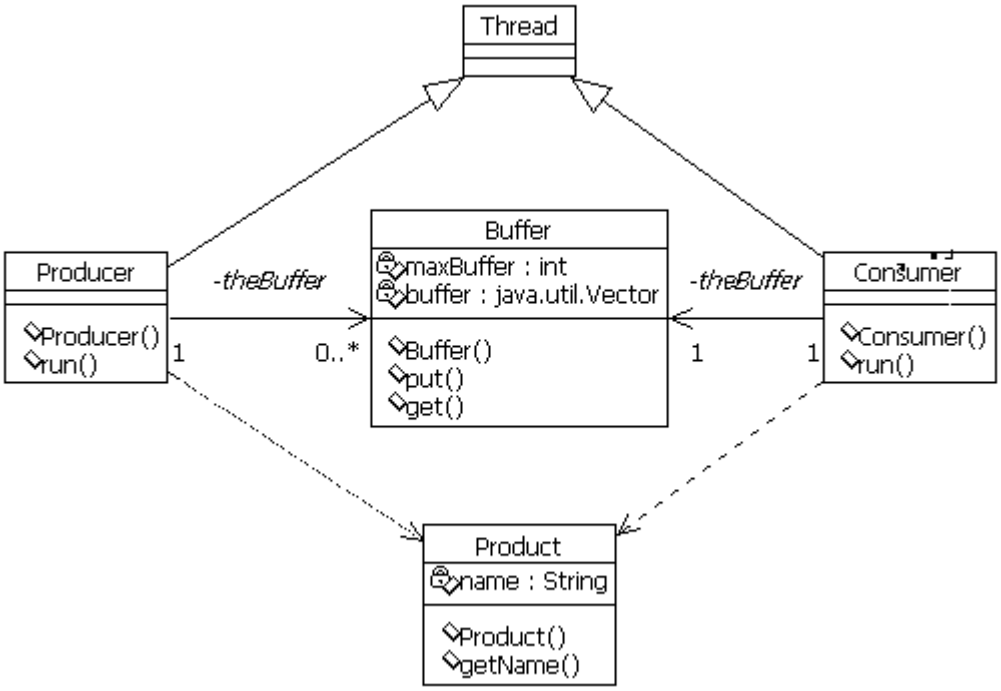


Figura de mai sus este diagrama de clasa din perspectiva conceptuala.
Clasele Producer si Consumer sunt derivate din clasa Thread (fir de executie).
Intre Producer si Buffer avem o asociere unidirectionala, numai Producer cunoaste Buffer si multiplicitatea asocierii este 1:n,
adica un obiect producator cunoaste n cozi de asteptare.
Intre Consumer si Buffer exista tot o asociere unidirectionala, imprimanta cunascand coada de asteptare, dar aici multiplicitatea
asocierii este 1:1 insemnand ca o imprimanta are atasata o singura coada de asteptare.

Perspectiva specificatiei



Continuam cu descrierea acestor clase, adica a atributelor si metodelor principale ale acestora.

Clasa Buffer

Atribute: Aceasta clasa trebuie sa aiba un atribut dimensiune prin care specificam dimensiunea maxima a unui buffer si inca un
alt atribut de tip colectie pentru a permite stocarea produselor.

Metode: Avem nevoie de doua metode, una care adauga un produs la sfarsitul cozii de asteptare si inca una care scoate
primul produs din coada de asteptare.

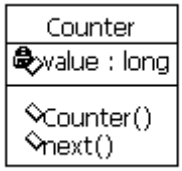
Clasa Producer

In cazul acestei clase nu avem nevoie de atribute, doar de cele care se nasc in urma asocierilor. Clasa va avea un constructor
care initializeaza numele firului de executie (atribut mostenit de la clasa parinte Thread) si atributul theBuffer creat in urma
asocierii cu clasa Buffer. Acest atribut este unul de tip array. Ca sa putem initializa acest atribut, trebuie sa avem create in
prealabil obiectele de tip Buffer.

Clasa Consumer

Clasa va avea un constructor care initializeaza numele firului de executie (atribut mostenit de la clasa parinte Thread) si atributul
theBuffer creat in urma asocierii cu clasa Buffer. Acest atribut este un atribut simplu, dar ca sa-l putem initializa, trebuie sa avem
create in prealabil obiectele de tip Buffer si atunci fiecarui obiect Buffer ii corespunde un obiect de tip Consumer.

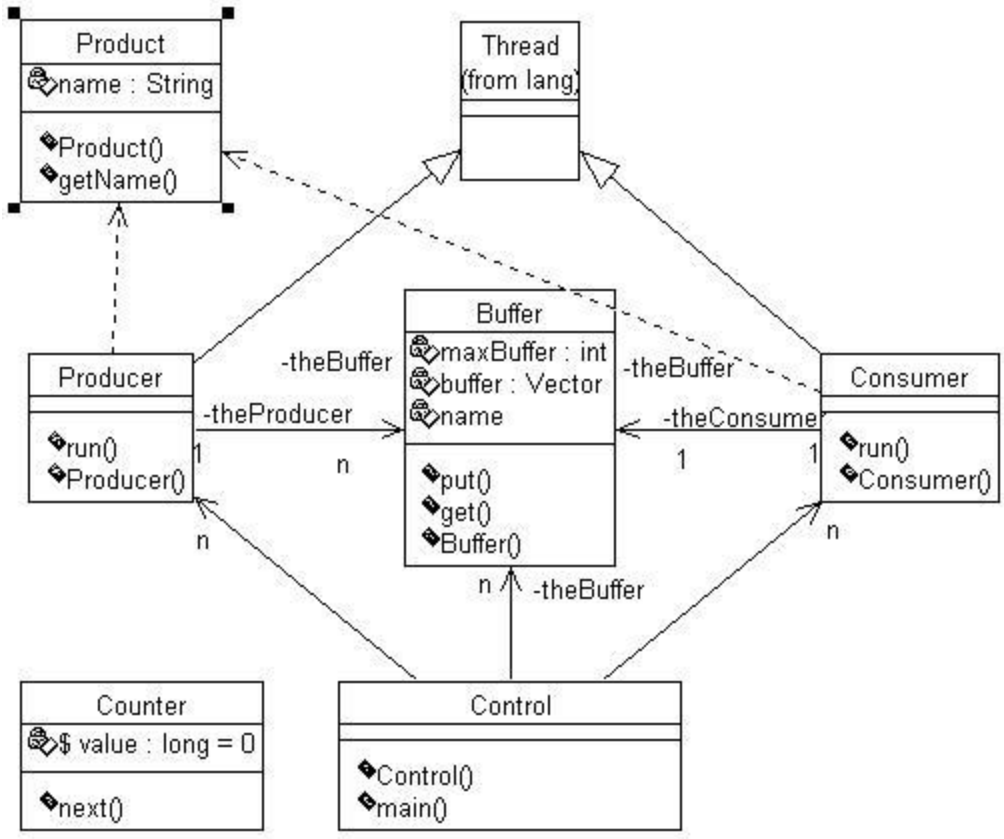
Introducem si o clasa **Product** pentru identificarea documentelor produse. Fiecare document va avea un nume, deci clasa
trebuie sa aiba un atribut tip sir de caractere reprezentand numele produsului. Acest atribut va fi initializat cu constructorul clasei
si adaugam o metoda de tip query pentru returnarea acestui atribut.
Pentru numerotarea documentelor vom introduce si o clasa **Counter**. La pornirea aplicatiei se creaza o singura instanta din
aceasta clasa si toti producatorii o vor folosi. Clasa va avea un atribut value de tip intreg lung si o operatie next, pentru
returnarea urmatoarei valori a numaratorului.



Perspectiva de implementare

La diagrama din sectiunea precedenta vom adauga inca o clasa Control. Aceasta clasa va avea cate o asociere cu calsele
Producer, Consumer si Buffer. Multiplicitatea acestor asocieri se vede pe diagrama alaturata. Constructorul clasei face
instantierile obiectelor Buffer, Producer, Consumer. Fiecare asociere are multiplicitatea 1:n, un singur obiect de tip control si

mai multi bufferi, producatori si consumatori. Trebuie sa vaem grija ca numarul bufferelor sa corespunda cu numarul
imprimantelor.



Codul final:

```
//Source file: Control.java


public class Control
{
   private Producer theProducer[];
   private Consumer theConsumer[];
   private Buffer theBuffer[];

   public Control()
   {
   }

   public Control(int np, int nc)
   {
        int i;

      theProducer = new Producer[ np ];
      theBuffer   = new Buffer  [ nc ];
      theConsumer = new Consumer[ nc ];

      for( i=0; i<nc; i++ )
       theBuffer[ i ] = new Buffer( 3 );

      for( i=0; i<np; i++ ){
       theProducer[ i ] = new Producer( theBuffer , "Producer. "+Integer.toString( i+1 ));
```

```
       theProducer[ i ].start();
      }
      for( i=0; i<nc; i++ ){
       theConsumer[ i ] = new Consumer( theBuffer[ i ], "Consumer. "+ Integer.toString( i+1
));
       theConsumer[ i ].start();
       }
   }

   public static void main(String[] args)
   {
    Control c = new Control(4,3);
   }
}


//Source file: Producer.java


public class Producer extends Thread
{
   private Buffer theBuffer[];


   public void run()
   {
      try{
             while( true ) {
               Product p = new Product( Long.toString(Counter.next()) );

             // Choose a buffer
              int nr = ( int )Math.round( Math.random() * theBuffer.length );
              if( nr >= theBuffer.length ) nr = theBuffer.length -nr;
              System.out.println( getName() +": "+p.getName()+"...."+Integer.toString( nr
));
              theBuffer[ nr ].put( p );
              sleep( Math.round( Math.random()*1000 ));
             }
         }
         catch( InterruptedException e ){}
   }


   public Producer(Buffer[] buffer, String name)
   {
     super( name );
     theBuffer = buffer;
   }
}

//Source file: C:\Manyi\TempRose\Consumer.java


public class Consumer extends Thread
{
   private Buffer theBuffer;

   public Consumer()
   {
   }
```

```
   public void run()
   {
     try{
             while( true ) {
                  Product p= theBuffer.get();
                  System.out.println( getName()+" "+p.getName());
                  sleep(2000);
             }
         }
         catch( InterruptedException e ) {}

   }



   public Consumer(Buffer buffer, String name)
   {
    super( name );
    theBuffer = buffer;
   }
}

//Source file: C:\Manyi\TempRose\Buffer.java

import java.util.Vector;

public class Buffer
{
   private int maxBuffer;
   private Vector buffer = new Vector();

   public Buffer()
   {
   }

   public synchronized void put(Product p) throws InterruptedException
   {
       while( buffer.size() == maxBuffer )
             wait();
       buffer.addElement( p );
       notify();
   }



   public synchronized Product get() throws InterruptedException
   {

         while( buffer.size() == 0 )
             wait();

         Product p  = (Product) buffer.firstElement();
         buffer.removeElement( p );

         notify();
         return p;

   }

   public Buffer(int maxBuffer)
   {
    this.maxBuffer = maxBuffer;
```

```
   }
}

//Source file: Counter.java


public class Counter
{
   private static long value = 0;

   public Counter()
   {
   }

   public static long next()
   {
    return value++;
   }
}

//Source file: Product.java

public class Product
{
   private String name;

   public Product()
   {
   }

   public Product(String name)
   {
    this.name = name;
   }

   public String getName()
   {
    return name;
   }
}
```

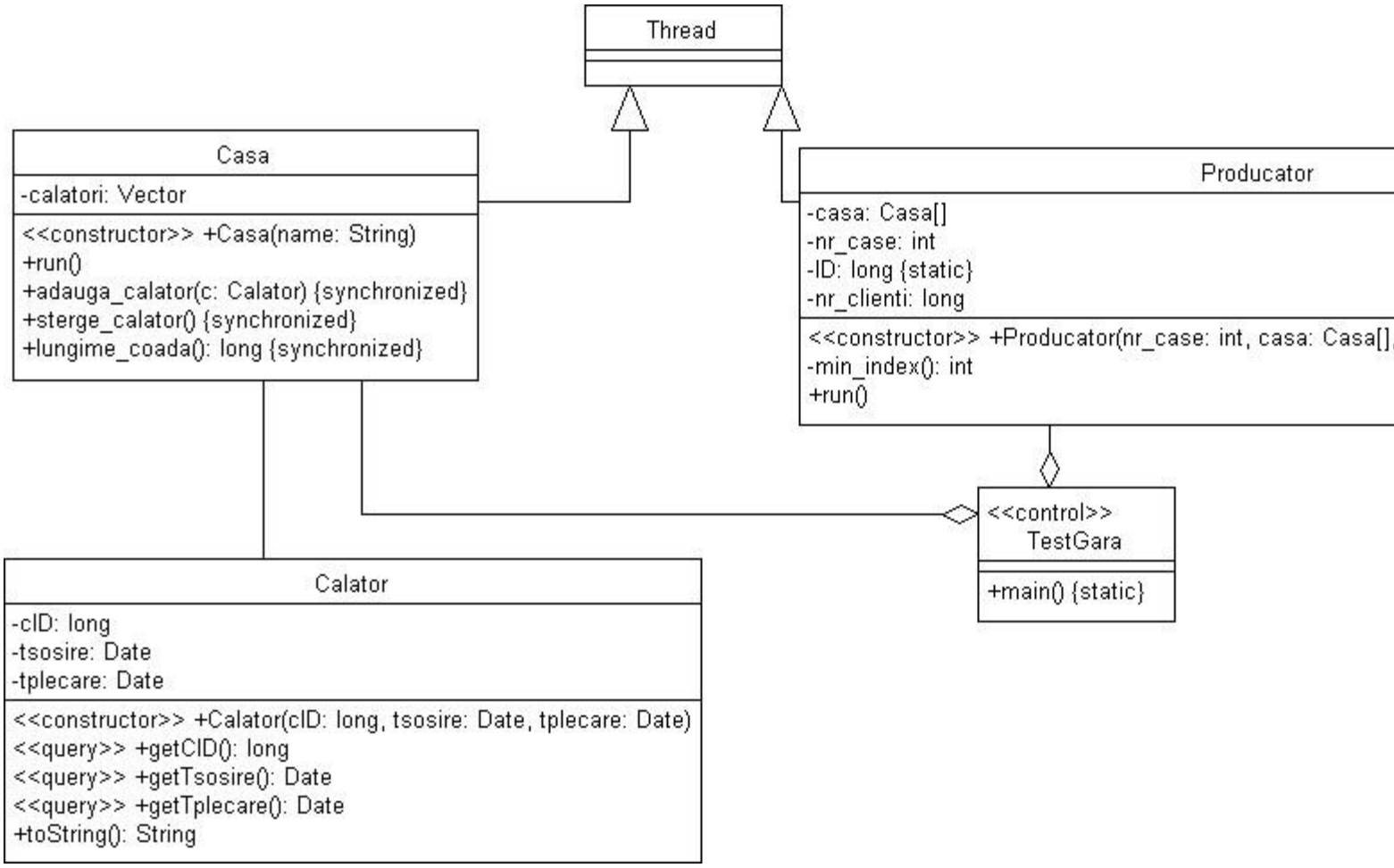
9. Exemple

9.1. Specificatie problema:

Intr-o gara sunt 3 case de bilete. Calatorii se aseaza la coada la una dintre cozi (de ex: la care e mai scurta), dar casieritele nu elibereaza biletele in acelasi ritm. Simulati functionarea caselor de bilete.

Diagrama de clase

Sursa Java:



Calator.java

```
package Gara;
import java.util.*;

public class Calator{

    private long cID;
    private Date tsosire;
    private Date tplecare;

    public Calator( long cID, Date tsosire, Date tplecare ){
         this.cID = cID;
         this.tsosire  = tsosire;
         this.tplecare = tplecare;
    }

    public long getCID(){
         return cID;
    }

    public Date getTsosire(){
         return tsosire;
    }

    public Date getTplecare(){
         return tplecare;
    }
```

```
    public String toString(){
         return ( Long.toString(cID)+" "+tsosire.toString()+" "+tplecare.toString());
    }

}
```

Casa.java

```
package Gara;

import java.util.*;

public class Casa extends Thread{
    private Vector calatori=new Vector();
        public Casa( String name ){
        setName( name );
    }

    public void run(){
        try{
         while( true ){
              sterge_calator();
              sleep( (int) (Math.random()*4000) );
         }
        }
        catch( InterruptedException e ){
             System.out.println("Intrerupere");
             System.out.println( e.toString());
         }
    }

    public synchronized void adauga_calator( Calator c ) throws InterruptedException
    {
         calatori.addElement(c);
         notifyAll();
    }

    public synchronized void sterge_calator() throws InterruptedException
    {
         while( calatori.size() == 0 )
             wait();
         Calator c = ( Calator )calatori.elementAt(0);
         calatori.removeElementAt(0);
         System.out.println(Long.toString( c.getCID())+" a fost deservit de casa
"+getName());
         notifyAll();
    }

    public synchronized long lungime_coada() throws InterruptedException{
         notifyAll();
         long size = calatori.size();
         return size;
    }

}
```

Producator.java

```
package Gara;
import java.util.*;
```

```
public class Producator extends Thread{


    private Casa casa[];
    private int nr_case;
    static long ID =0;
    private int nr_clienti;//Numar calatori care trebuie deserviti de catre casele de bilete

    public Producator( int nr_case, Casa  casa[], String name, int nr_clienti ){
         setName( name );
         this.nr_case = nr_case;
         this.casa = new Casa[ nr_case ];
         this.nr_clienti = nr_clienti;
         for( int i=0; i<nr_case; i++){
             this.casa[ i ] =casa[ i ] ;
         }
    }

    private int min_index (){
      int index = 0;
      try
      {
         long min  = casa[0].lungime_coada();
         for( int i=1; i<nr_case; i++){
             long lung = casa[ i ].lungime_coada();
             if ( lung < min ){
                      min = lung;
                      index = i;
             }
         }
      }
      catch( InterruptedException e ){
           System.out.println( e.toString());
       }
      return index;
    }


    public void run(){
      try
      {
         int i=0;
         while( i<nr_clienti ){
             i++;
             Calator c = new Calator( ++ID, new Date(), new Date() );
             int m = min_index();
             System.out.println("Calator :" +Long.toString( ID )+" adaugat la CASA
"+     Integer.toString(m));

             casa[ m ].adauga_calator( c );
             sleep( (int) (Math.random()*1000) );
         }
      }
      catch( InterruptedException e ){
          System.out.println( e.toString());
      }
    }
}
```

TestGara.java

```
import Gara.*;
public class TestGara{
    public static void main( String args[] ){
     int i;
     int nr = 4;
     Casa c[] = new Casa[ nr ];
     for( i=0; i<nr; i++){
         c[ i ] = new Casa("Casa "+Integer.toString( i ));
         c[ i ].start();
     }
     Producator p = new Producator( nr , c, "Producator",30);
         p.start();

    }
}
```