TEHNICI AVANSATE DE PROGRAMARE
LUCRARE DE LABORATOR 8

Fire de execuție în Java

I. SCOPUL LUCRĂRII

Lucrarea de față are rolul de a prezenta şi familiariza studentul cu modul de construire şi utilizare a firelor de execuție în Java.

La sfârşitul acestei lucrări, studentul va avea posibilitatea să scrie programe Java în care să utilizeze noțiunile învățate.

II. NOȚIUNI TEORETICE

Crearea firelor de execuție

"Multithreading" înseamnă capacitatea unui program de a executa mai multe secvențe de cod în acelaşi timp. O astfel de secvență de cod se numeşte fir de execuție sau **thread**. Limbajul Java suportă multithreading prin clase disponibile în pachetul **java.lang**. În acest pachet există 2 clase **Thread** şi **ThreadGroup,** şi interfața **Runnable**. Clasa **Thread** şi interfața **Runnable** oferă suport pentru lucrul cu thread-uri ca entități separate, iar clasa **ThreadGroup** pentru crearea unor grupuri de thread-uri în vederea tratării acestora într-un mod unitar.

Există 2 metode pentru crearea unui fir de execuție: se creează o clasă derivată din clasa **Thread**, sau se creează o clasă care implementeză interfața **Runnable**.

Crearea unui fir de execuție prin extinderea clasei Thread

Se urmează etapele:
- se creează o clasă derivată din clasa **Thread**
- se suprascrie metoda **public void run()** moştenită din clasa **Thread**
- se instanțiază un obiect thread folosind **new**
- se porneşte thread-ul instanțiat, prin apelul metodei **start()** moştenită din clasa Thread. Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program necesar unui thread după care să apeleze metoda **run()**.

Exemplu:

```
public class Fir
{
   public static void main(String args[])
  {
     FirdeExecutie fir=new FirdeExecutie();
     fir.start();
     System.out.println("Revenim la main");
  }
}

class FirdeExecutie extends Thread
```

```
{
   public void run()
   {
      for(int i=0;i<10;i++)
         System.out.println("Pasul "+i);
      System.out.println("Run s-a terminat");
   }
}
```

Observație:

Metoda main() are propriul fir de execuție. Prin apelul **start()** se cere JVM crearea şi pornirea unui nou fir de execuție. Din funcția **start()** se va ieşi imediat. Firul de execuție corespunzător metodei **main()** îşi va continua execuția independent de noul fir de execuție creat.

Să considerăm un alt exemplu simplu în care se vor crea două fire de execuție ce rulează concomitent. **Metoda sleep()** cere oprirea rulării firului de execuție curent pentru un interval de timp specificat.

```
public class Fir1
{
   public static void main(String args[])
  {
     FirdeExecutie fir1=new FirdeExecutie();
     FirdeExecutie fir2=new FirdeExecutie();
     fir1.start();
     fir2.start();
     System.out.println("Revenim la main");
  }
}

class FirdeExecutie extends Thread
{
   public void run()
   {
      for(int i=0;i<10;i++)
      {
         System.out.println("Pasul "+i);
         try{
             sleep(500);//oprirea pt. 0,5 secunde a firului de executie
            }
         catch(InterruptedException e)  {System.err.println("Eroare");}
      }
      System.out.println("Run s-a terminat");
   }
}
```

Java defineşte 3 constante pentru selectarea priorităților firelor de execuție:

```
public final static int MAX_PRIORITY; // 10
public final static int MIN_PRIORITY; // 1
public final static int NORM_PRIORITY; // 5
```

O metodă importantă în contextul utilizării priorităților este

```
public static native void yield()
```

care scoate procesul curent din execuție şi îl pune în coada de aşteptare.

Iată un exemplu simplu care demonstrează cum se lucrează cu prioritățile firelor de execuție. Metoda **getName()** (moştenită din clasa **Thread**) returnează numele procesului curent.

```
public class Fir2
{
```

```
   public static void main(String args[])
  {
     FirdeExecutie fir1=new FirdeExecutie("Fir 1");
     FirdeExecutie fir2=new FirdeExecutie("Fir 2");
     FirdeExecutie fir3=new FirdeExecutie("Fir 3");
     fir1.setPriority(Thread.MIN_PRIORITY);
     fir2.setPriority(Thread.MAX_PRIORITY);
     fir3.setPriority(7);
     fir1.start();
     fir2.start();
     fir3.start();
     System.out.println("Revenim la main");
  }
}
class FirdeExecutie extends Thread
{
   public FirdeExecutie(String s)
   {
     super(s);
   }
   public void run()
   {
      String numeFir=getName();
      for(int i=0;i<5;i++)
      {
        //if(numeFir.compareTo("Fir 3")==0) yield();
        System.out.println(numeFir+ " este la pasul "+i);
        try{
              sleep(500);
            }
         catch(InterruptedException e)  {System.err.println("Eroare");}
      }
      System.out.println(numeFir+ " s-a terminat");
   }
}
```

Observație:

Implementările Java depind de platformă. Un program Java care foloseşte fire de execuție poate avea comportări diferite la execuții diferite pentru aceleaşi date de intrare. Platformele Windows folosesc cuante de timp (firele de execuție sunt administrate într-o manieră Round-Robin).

Crearea unui fir de execuție folosind interfața Runnable

Este o modalitate extrem de utilă atunci când clasa de tip **Thread** care se doreşte a fi implementată moştenește o altă clasă (Java nu permite moştenirea multiplă). Interfața **Runnable** descrie o singură metodă **run()**.

Se urmează etapele:

- se creează o clasă care implementează interfața **Runnable**
- se implemetează metoda **run()** din interfață
- se instanțiază un obiect al clasei folosind **new**
- se creează un obiect din clasa **Thread** folosind un constructor care are ca parametru un obiect de tip **Runnable** (un obiect al clasei ce implementează interfața)
- se porneşte thread-ul creat la pasul anterior prin apelul metodei **start()**.

Exemplu:

```
public class Fir3
{
   public static void main(String args[])
  {
     FirdeExecutie fir=new FirdeExecutie();
     Thread thread=new Thread(fir);
     thread.start();
     System.out.println("Revenim la main");
  }
}

class A
{
  public void afis()
 {
   System.out.println("Este un exemplu simplu");
 }
}

class FirdeExecutie extends A implements Runnable
{
   public void run()
   {
      for(int i=0;i<5;i++)
         System.out.println("Pasul "+i);
      afis();
      System.out.println("Run s-a terminat");
   }
}
```

Observații utile:

Un fir de execuție se poate afla la un moment dat în una din următoarele stări: **running** (rulează), **waiting** (adormire, blocare, suspendare), **ready** (gata de execuție, prezent în coada de aşteptare), **dead** (terminat). Fiecare fir de execuție are o prioritate de execuție. În general thread-ul cu prioritatea cea mai mare este cel care va accesa primul resursele sistem.

O altă clasă aparte de thread-uri sunt cele **Daemon** care sunt thread-uri de serviciu (aflate în serviciul altor fire de execuție). Când se pornesşte maşina virtuală Java, există un singur fir de execuție care nu este de tip Daemon şi care apelează metoda main(). JVM rămâne pornită cât există activ un thread care să nu fie de tipul Daemon.

Excludere mutuală şi sincronizare

Presupunem că două fire de execuție incrementează valoarea unui întreg partajat. Pot apare probleme în sensul că cele 2 fire de execuție incrementează în acelaşi timp întregul, sărindu-se astfel peste valori ale acestuia. Problema se rezolvă dacă cel mult un fir de execuție are acces la acea dată la un moment dat. Java implementează excluderea mutuală prin specificarea metodelor care partajează variabile ca fiind **synchronized**. Aceste metode trebuie să fie din aceeaşi clasă. Orice fir de execuție care încearcă să acceseze o metodă **synchronized** a unui obiect atîta timp cât metoda este utilizată de un alt fir de execuție, este blocat până când primul fir de execuție părăseşte metoda. Iată un exemplu.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
```

```
public class DoiContori extends Applet implements ActionListener
{
private Button start;
private int contor=0;
public void init()
{
start=new Button("Start");
add(start);
start.addActionListener(this);
}
public void actionPerformed(ActionEvent event)
{
 if(event.getSource()==start)
  {
     contor++;
     Graphics g=getGraphics();
     NumarPartajat numar=new NumarPartajat(g,contor);
     Contor1 contor1=new Contor1(numar);
     Contor2 contor2=new Contor2(numar);
     contor1.start();
     contor2.start();
  }
}
}

class Contor1 extends Thread
{
 private NumarPartajat nr;
 public Contor1(NumarPartajat nr)
 {
   this.nr=nr;
 }
 public void run()
 {
  for(int i=1;i<=10;i++)
       nr.increment();
 }
}

class Contor2 extends Thread
{
 private NumarPartajat nr;
 public Contor2(NumarPartajat nr)
 {
   this.nr=nr;
 }
 public void run()
 {
  for(int i=1;i<=10;i++)
       nr.increment();
 }
}

class NumarPartajat
{
 private int n=0;
 private Graphics g;
 private int x=0;
 private int contor;
```

```
public NumarPartajat(Graphics g,int contor)
{
  this.g=g;
  this.contor=contor;
}
public synchronized void increment()
{
 n=n+1;
 g.drawString(n+", ",x*20,30+15*contor);
 x++;
}
}
```

Metodele wait() şi notify()

Tot o problemă de excludere mutuală este exemplul producător-consumator. Producătorul furnizează date pe care consumatorul le utilizează mai departe. Ce se întâmplă dacă vitezele de producere şi consum diferă? În acest scop se utilizează metodele **wait(), notify()** şi **notifyAll()** ale clasei **Object**. Apelul lui **wait()** va trece obiectul apelat în starea Blocked. El rămâne blocat până la apelul unei metode **notify()** sau **notifyAll()** pentu acelaşi obiect. Condiția necesară pentru a se apela una dintre aceste metode este ca apelul lor să se facă în interiorul metodelor **synchronized**. Pentru metoda **wait()** se poate specifica o durată maximă de aşteptare. În acest caz, firul de execuție rămâne blocat până când timpul expiră sau un alt fir de execuție apelează **notify()**.

Să considerăm următorul exemplu. Un program Java pentru simularea unei cafenele. O comandă se realizează prin apăsarea unui buton (burger, fries, cola). Comanda se afişează pe ecran şi apoi se introduce într-o coadă care de asemenea se afişează pe ecran. Şeful ia comnada din coadă în sistemul primul venit, primul servit. Chelnerul acceptă o comandă, o afişează şi o inserează în coadă.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Cafe extends Applet implements ActionListener
{
private Button burger, fries, cola, cooked;
private Order order, complete;

public void init()
{
  Graphics g=getGraphics();
  burger=new Button("Burger");
  add(burger);
  burger.addActionListener(this);
  fries=new Button("Fries");
  add(fries);
  fries.addActionListener(this);
  cola=new Button("Cola");
  add(cola);
  cola.addActionListener(this);
  cooked=new Button("Cooked");
  add(cooked);
  cooked.addActionListener(this);
  order=new Order();
  Queue queue=new Queue(g);
  complete=new Order();
```

```
  Waiter waiter=new Waiter(g,order,queue);
  waiter.start();
  Chef chef=new Chef(g,complete, queue);
  chef.start();
}

public void actionPerformed(ActionEvent event)
{
  if(event.getSource()==burger)
        order.notifyEvent("burger");
  if(event.getSource()==fries)
        order.notifyEvent("fries");
  if(event.getSource()==cola)
        order.notifyEvent("cola");
  if(event.getSource()==cooked)
        complete.notifyEvent("cooked");
}
}

class Waiter extends Thread
{
 private Order order;
 private Graphics g;
 private Queue queue;

public Waiter(Graphics g, Order order,Queue queue)
{
  this.g=g;
  this.order=order;
  this.queue=queue;
}
public void run()
{
  g.drawString("O noua comanda",10,50);
  while(true)
  {
     String nouaComanda=order.waitForEvent();
     g.clearRect(10,50,50,25);
     g.drawString(nouaComanda,10,70);
     try{
           Thread.sleep(5000);
     }
     catch(InterruptedException e){
           System.out.println("Exceptie waiter!");
     }
     if (! queue.isFull())
          queue.enter(nouaComanda);
  }
}
}

class Order
{
 private String order="";

 public synchronized void notifyEvent(String nouaComanda)
 {
   order=nouaComanda;
   notify();
```

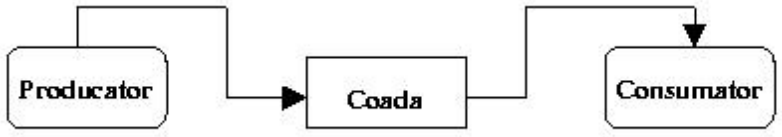
```
 }
 public synchronized String waitForEvent()
 {
   while (order.equals(""))
      try{
           wait();
      }
      catch(InterruptedException e) {
          System.out.println("Exceptie order!");
      }
   String nouaComanda=order;
   order="";
   return nouaComanda;
 }
}

class Queue
{
  private Graphics g;
  private String[] queue=new String[20];
  private int count=0;

  public Queue(Graphics g)
  {
    this.g=g;
  }
  public synchronized void enter(String item)
  {
    queue[count]=item;
    count++;
    display();
    notify();
  }
  public synchronized String remove()
  {
    while(count==0)
       try{
           wait();
       }
       catch(InterruptedException e) {
          System.out.println("Exceptie queue!");
       }
    String item=queue[0];
    count--;
    for(int c=0;c<count;c++)
       queue[c]=queue[c+1];
    display();
    return item;
  }
  public synchronized boolean isFull()
  {
    return count==queue.length;
  }
  private void display()
  {
    g.drawString("Coada",120,50);
    g.clearRect(120,50,50,220);
    for(int c=0;c<count;c++)
       g.drawString(queue[c],120,70+c*20);
```

```
  }
}

class Chef extends Thread
{
 private Graphics g;
 private Order complete;
 private Queue queue;

 public Chef(Graphics g, Order complete, Queue queue)
 {
   this.g=g;
   this.complete=complete;
   this.queue=queue;
 }
 public void run()
 {
  g.drawString("Cooking",200,50);
  while(true)
  {
     String order=queue.remove();
     g.clearRect(200,55,50,25);
     g.drawString(order,200,70);
     g.drawString(order,200,70);
     String cookedInfo=complete.waitForEvent();
     g.clearRect(200,55,50,25);
  }
 }
}
```

Exemple:

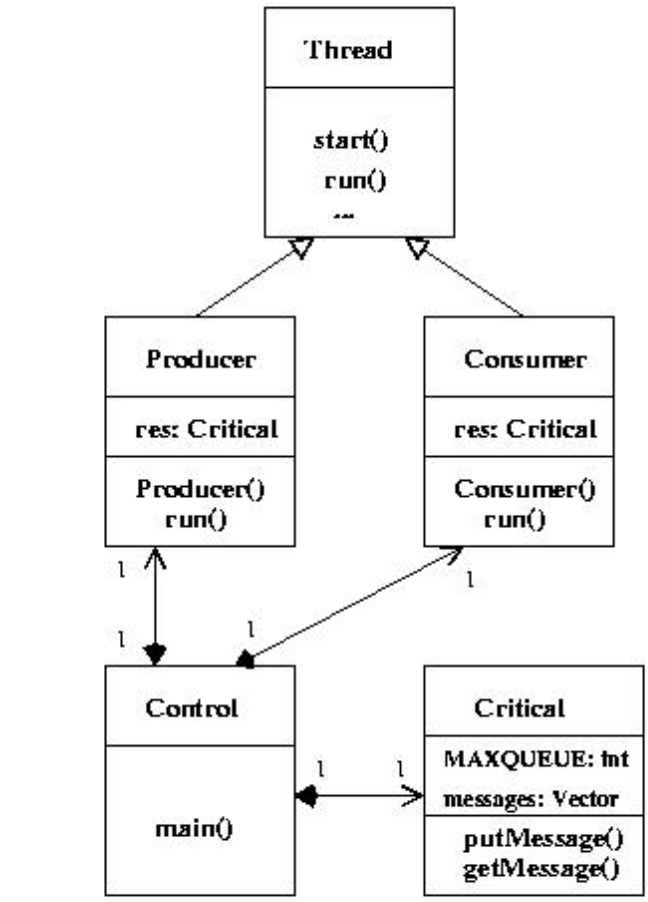
1. **Problema producatorului si a consumatorului**

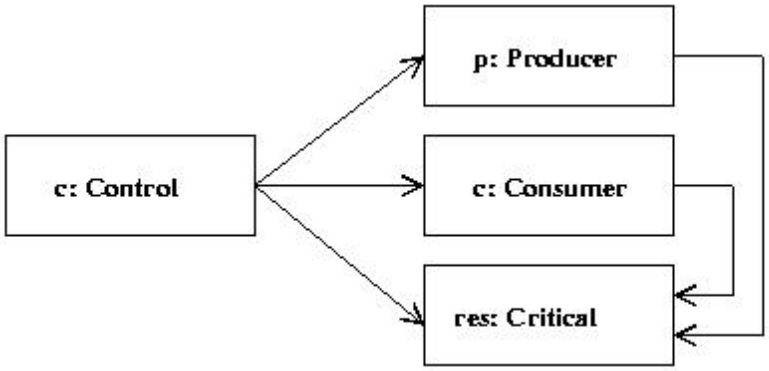
Specificatia problemei:



Producatorul produce mesaje care contin data si ora emiterii mesajului. Produsele sunt stocate intr-o coada. Consumatorul "consuma" mesajele din coada afisandu-le la terminal. Coada reprezinta o resursa critica, la care accesul trebuie sincronizat. Producatorul va bloca accesul la coada in momentul depunerii "produsului" in coada, si va elibera accesul dupa ce depunerea s-a terminat. Daca coada se umple, producatorul intra in asteptare. Daca coada se goleste, atunci consumatorul va intra in asteptare.

Descrierea claselor:





Sursa Java:

```
import java.util.*;
class Critical{
   static final int MAXQUEUE = 5;
   private Vector messages = new Vector();
   public synchronized void putMessage() throws InterruptedException{
       while( messages.size() == MAXQUEUE )
          wait();
       String message = new String( new java.util.Date().toString() );
       System.out.println("Producator: "+message);
       messages.addElement( message );
       notify();
   }
   public synchronized String getMessage() throws InterruptedException{
       notify();
       while( messages.size() == 0 )
          wait();
       String message = (String) messages.firstElement();
       messages.removeElement( message );
       return message;
   }
}
class Producer extends Thread{
   private Critical res;
   public Producer( String name, Critical res )
   {
       super(name);
       this.res = res;
   }
   public void run()
   {
       try{
          while( true ) {
             res.putMessage();
             sleep( 1000 );
          }
       }
       catch( InterruptedException e ){}
   }
}
class Consumer extends Thread{
   private Critical res;
   public Consumer( String name, Critical res )
   {
       super(name);
       this.res = res;
   }
   public void run()
   {
```

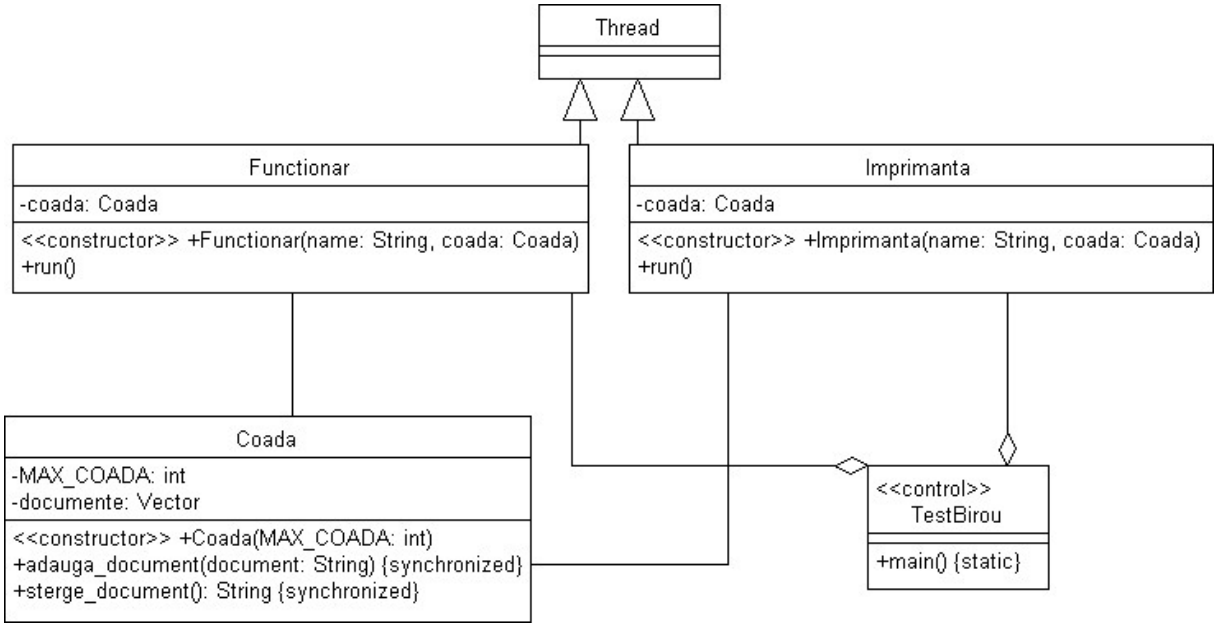
```
        try{
           while( true ) {
               String message = res.getMessage();
               System.out.println("Consumator: "+message);
               sleep(2000);
           }
        }
        catch( InterruptedException e ) {}
    }
}
public class Control6{
    public static void main( String args[])
    {
        Critical res = new Critical();
        Producer p = new Producer( "Fir producator",res);
        Consumer c = new Consumer( "Fir consumator",res);
        p.start();
        c.start();
    }
}
```

2.

Specificatie problema:

Intr-un birou sunt 8 functionari care din cand in cand tiparesc la imprimanta documente, nu toti eleboreaza documentele in acelasi ritm. Fiindca au o singura imprimanta in birou, poate tipari doar o singura persoana la un moment dat. Sa se simuleze functionarea biroului.

Descrierea claselor



Sursa Java:

Coada.java

```
package Birou;
import java.util.*;
public class Coada{
```

```
   private Vector documente= new Vector();
   private int MAX_COADA;

   public Coada( int MAX_COADA ){
       this.MAX_COADA=MAX_COADA;
   }
   public synchronized void adauga_document( String document )
     throws InterruptedException{
       while( documente.size() == MAX_COADA )
          wait();
       documente.addElement( document );
       notifyAll();
   }
   public synchronized String sterge_document()
     throws InterruptedException{
       notifyAll();
       while( documente.size() == 0 )
          wait();
       String document = ( String )documente.elementAt(0);
       documente.removeElementAt( 0 );
       return document ;
   }
}
```

Functionar.java

```
package Birou;
public class Functionar extends Thread{
   private Coada coada;
   private int nr_document;
   public Functionar ( String name, Coada coada ){
       setName( name );
       this.coada = coada;
       nr_document = 0;
   }
   public void run(){
    try{
       while( true ){
        nr_document++;
        coada.adauga_document( getName()+"__"+Integer.toString(nr_document));
        sleep( (int) (Math.random() * 2000) );
       }
    }
    catch( InterruptedException e ){
    }
   }
}
```

Imprimanta.java

```
package Birou;
public class Imprimanta extends Thread{
```

```
   private Coada coada;
   public Imprimanta ( String name, Coada coada ){
       setName( name );
       this.coada = coada;
   }
   public void run(){
    try{
       while( true ){
        System.out.println( "TIPARIRE: "+coada.sterge_document());
        sleep( (int) (Math.random() * 500) );
       }
    }
    catch( InterruptedException e ){
    }
   }
}
```

TestBirou.java

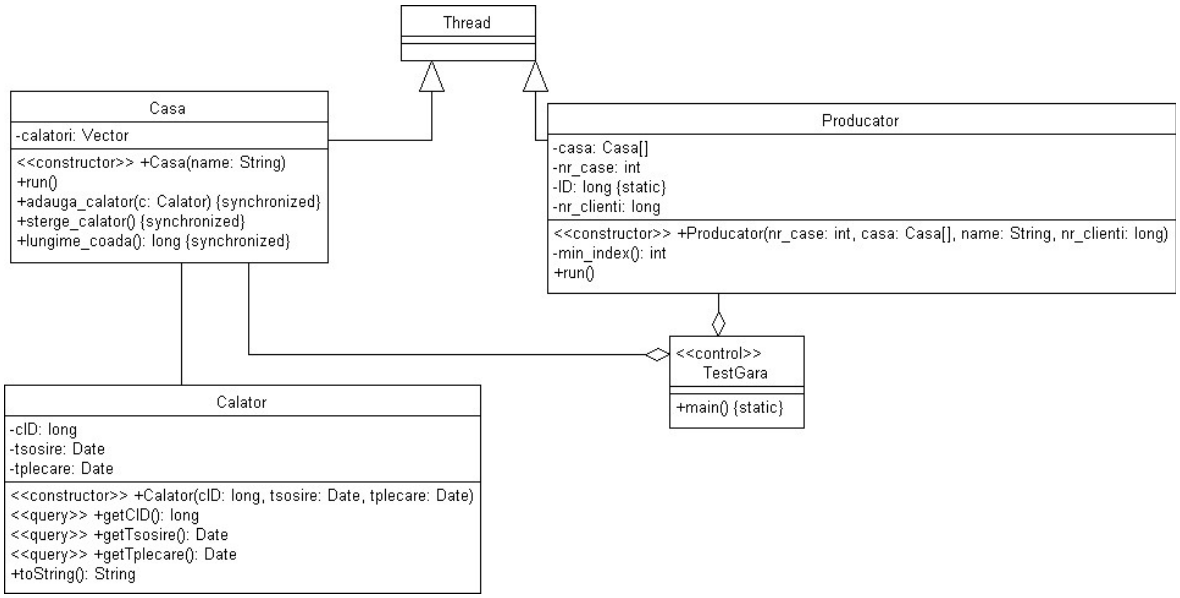
```
import Birou.*;
public class TestBirou{
   public static void main( String args[] ){
       int nr_functionari =4;
       Coada c = new Coada( 10 );
       Functionar f[] = new Functionar[5];
       for( int i=0;i<nr_functionari; i++ ){
           f[ i ] = new Functionar( Integer.toString(i+1), c);
           f[ i ].start();
       }
       Imprimanta i = new Imprimanta( "Imprimanta" , c );
       i.start();
   }
}
```

3.

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 vand biletele in acelasi ritm. Simulati functionarea caselor de bilete.

Descrierea claselor



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();
    }
}
```

III. MODUL DE LUCRU

- Clasic:

1. Se editează codul sursă al programului Java folosind un editor de text disponibil (de ex., se poate utiliza Notepad).
2. Se salvează fişierul cu extensia **.java**.
3. Compilarea aplicației Java se va face din linia de comandă:
   javac nume_fişier_sursă.java
   În cazul în care programul conține erori acestea vor fi semnalate şi afişate.
4. Pentru rularea aplicației Java, se lansează interpretorul Java:
   java nume_clasă_care_conține_main

- Se foloseşte utilitarul disponibil în laborator J2SDK Net Beans IDE.

IV. TEMĂ

1. Se vor parcurge toate exemplele prezentate în platforma de laborator testându-se practic şi explicând rezultatele obținute.
2. Scrieți un program Java pentru simularea unui cronometru (utilizând fire de execuție). Acesta are două butoane, unul pentru start şi continuarea secvenței de timp şi altul pentru revenirea la poziția inițială.
3. Scrieți un program Java care generează două fire de execuție pentru parcurgerea unui String de la cele două capete. Folosiți doi pointeri a căror valoarea se incrementează, respectiv se decrementează într-o funcție din memoria comună (synchronized). În memoria comună se află String-ul. Firele de execuție se întâlnesc la „mijloc”.