

TEHNOLOGII JAVA

PENTRU DEZVOLTAREA APLICAȚIILOR

LUCRARE DE LABORATOR 17

Fire de execuție în Java

I. SCOPUL LUCRĂRII

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

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

II. NOȚIUNI TEORETICE

1. Crearea firelor de execuție

“Multithreading” înseamnă capacitatea unui program de a executa mai multe secvențe de cod în același timp. O astfel de secvență de cod se numește fir de execuție sau **thread**. Limbajul Java suportă multithreading prin clase disponibile în pachetul **java.lang**. În acest pachet există 2 clase **Thread** și **ThreadGroup**, și interfața **Runnable**. Clasa **Thread** și 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 implementează 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()** moștenită din clasa **Thread**
- se instanțiază un obiect thread folosind **new**
- se pornește thread-ul instanțiat, prin apelul metodei **start()** moștenită din clasa **Thread**. Apelul acestei metode face ca mașina 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 și pornirea unui nou fir de execuție. Din funcția **`start()`** se va ieși imediat. Firul de execuție corespunzător metodei **`main()`** își 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 definește 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 și îl pune în coada de așteptare.

Iată un exemplu simplu care demonstrează cum se lucrează cu prioritățile firelor de execuție. Metoda **getName()** (moștenită 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 folosește fire de execuție poate avea comportări diferite la execuții diferite pentru aceleași 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 dorește a fi implementată moștenește o altă clasă (Java nu permite moștenirea multiplă). Interfața **Runnable** descrie o singură metodă **run()**.

Se urmează etapele:

- se creează o clasă care implementează interfața **Runnable**
- se implementează 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 pornește 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 așteptare), **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 pornesște mașina virtuală Java, există un singur fir de execuție care nu este de tip Daemon și care apelează metoda `main()`. JVM rămâne pornită cât există activ un thread care să nu fie de tipul Daemon.

2. Excludere mutuală și 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 același 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 aceeași 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 părăsește 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++;
    }
}

```

3. Metodele wait() și 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 și consum diferă? În acest scop se utilizează metodele **wait()**, **notify()** și **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()** pentru același 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 așteptare. 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 afișează pe ecran și apoi se introduce într-o coadă care de asemenea se afișează pe ecran. Șeful ia comanda din coadă în sistemul primul venit, primul servit. Chelnerul acceptă o comandă, o afișează și 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(col);
        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("Coadă",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);
        }
    }
}

```

III. MODUL DE LUCRU

1. Se editează codul sursă al programului Java folosind un editor de text disponibil(de ex., se poate utiliza Notepad).
2. Se salvează fișierul cu extensia **.java**. Fișierul trebuie salvat la următoarea locație:
c:\JBulider7\jdk1.3.1\bin
3. Compilarea aplicației Java se va face din linia de comandă. Se poate proceda astfel. Se deschide o fereastră MS-Dos: Start ->Run, se tipărește **command** în căsuța de text și se apasă butonul OK. Printr-o schimbare de directoare și subdirectoare se trece la locația:

c:\JBulider7\jdk1.3.1\bin. Sau, se lansează WindowsCommander. Se trece la locația **c:\JBulider7\jdk1.3.1\bin.** Se deschide o fereastră MS-Dos: Commander ->Run Dos.

4. Pentru compilare, se tipărește la prompter **javac nume_fișier_sursă.java** și se apasă Enter. De ex., dacă fișierul se numește Test.java, se va scrie **javac Test.java**. În cazul în care programul conține erori acestea vor fi semnalate și afișate în fereastra MS-Dos, după care va apare iar prompter-ul. Dacă programul nu conține erori și compilarea se face cu succes, atunci va apare numai prompter-ul.
5. Pentru rularea unei aplicații Java stand-alone, se lansează interpretorul Java. Se tipărește la prompter următoarea comandă **java nume_clasă_care_conține_main** și se apasă Enter. De ex., dacă clasa din program care conține metoda *main()* se numește Test, se va scrie **java Test**.
6. Dacă programul Java este un applet, se editează fișierul **.html**. Se salvează în același director cu fișierul **.class** rezultat în urma compilării cu succes a fișierului sursă java. Apoi pentru rulare se poate utiliza **appletviewer nume.html**.
Alternativ, după compilarea aplicației Java, fișierul **.class** împreună cu fișierul **.html** pot fi mutate în orice alt director (nu trebuie neapărat să fie în **c:\JBulider7\jdk1.3.1\bin**). Se încarcă fișierul **.html** într-un browser Web (ex., Internet Explorer).

IV. TEMĂ

1. Se vor parcurge toate exemplele prezentate în platforma de laborator testându-se practic.
2. Scrieți un program Java pentru simularea unui cronometru (utilizând fire de execuție). Acesta are două butoane, unul pentru start și continuarea secvenței de timp și 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”.