FIR DE EXECUȚIE

1. Introducere

În contextul programării concurente, un **fir de execuție** (*thread*) reprezintă cea mai mică unitate de execuție care poate fi planificată independent de către sistemul de operare. În Java, firele de execuție sunt elemente fundamentale pentru realizarea aplicațiilor care necesită paralelism, responsivitate crescută sau procesări simultane.

Limbajul Java oferă un suport robust pentru gestionarea firelor, integrând atât mecanisme de creare, cât și de sincronizare a acestora, facilitând astfel implementarea programării concurente într-un mod sigur și controlat.

2. Modelul de execuție în Java

2.1 Thread-ul principal

La pornirea unei aplicații Java, JVM creează automat un fir de execuție principal numit **main thread**, responsabil de apelarea metodei public static void main(String[] args). Toate firele adiționale sunt create pornind de la acesta.

2.2 Multithreading

Conceptul de **multithreading** se referă la abilitatea unui program de a executa mai multe fire simultan. Acest model oferă următoarele beneficii:

- **Creșterea performanței** prin exploatarea procesoarelor multi-core.
- **Îmbunătățirea responsivității** aplicațiilor (ex.: UI, servere).
- **Împărțirea sarcinilor complexe** în sub-sarcini mai mici.

3. Crearea firelor de execuție în Java

Java oferă două modalități principale de definire și lansare a firelor:

3.1 Extinderea clasei Thread

Pentru a crea un fir, se poate extinde clasa Thread și se poate suprascrie metoda run():

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Firul se execută.");
    }
}
```

Inițializarea și pornirea firului se realizează prin:

```
MyThread t = new MyThread();
t.start();
```

3.2 Implementarea interfeței Runnable

O abordare recomandată, mai flexibilă, constă în implementarea interfeței Runnable:

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Executarea firului prin Runnable.");
    }
}
```

Lansarea:

```
Thread t = new Thread(new MyRunnable());
t.start();
```

3.3 Diferența dintre run() și start()

- Apelarea **run()** execută metoda în același fir, fără a crea un fir nou.
- Apelarea **start()** creează un fir de execuție separat și, intern, apelează run().

Această distincție este esențială în programarea concurentă.

4. Stările unui fir de execuție

Conform specificației Java, un fir poate parcurge următoarele stări:

1. **New** – firul a fost creat, dar nu a fost pornit.
2. **Runnable** – firul este eligibil pentru execuție; poate fi în execuție efectivă sau în așteptare.
3. **Blocked** – firul este blocat, de obicei așteaptă accesul la o resursă sincronizată.
4. **Waiting** – firul așteaptă un eveniment extern (ex: wait(), join()).
5. **Timed Waiting** – firul așteaptă un timp determinat (ex: sleep()).
6. **Terminated** – execuția firului s-a încheiat.

FIR DE EXECUȚIE

Tabel – Metode importante pentru gestionarea firelor în Java

Metodă	Descriere	Clasa / Tip	Utilizare tipică
start()	Creează un nou fir de execuție și pornește metoda run() în acel fir.	Thread	Lansarea unui thread nou.
run()	Codul care va fi executat de fir; se suprascrie de către programator.	Thread / Runnable	Definirea comportamentului unui thread.
sleep(long millis)	Suspendă execuția firului pentru un interval de timp dat.	Thread	Pauze controlate, temporizare.
join()	Firul curent așteaptă finalizarea firului asupra căruia se aplică.	Thread	Sincronizarea execuției dintre fire.
join(long millis)	Așteaptă terminarea firului un timp limitat.	Thread	Sincronizare cu limită de timp.
yield()	Sugerează planificatorului să cedeze procesorul altui fir de același nivel de prioritate.	Thread	Ajustarea fină a execuției.
interrupt()	Semnalează unui fir că trebuie să înceteze execuția (nu îl oprește forțat).	Thread	Oprire controlată a firelor.
isInterrupted()	Verifică dacă firul a primit un semnal de întrerupere.	Thread	Gestionarea întreruperilor.
interrupted()	Verifică și resetează starea de întrerupere a firului curent.	Thread	Monitorizarea întreruperilor.
setPriority(int priority)	Setează prioritatea firului în planificator.	Thread	Stabilirea ordinii de execuție (rareori recomandat).
getPriority()	Returnează prioritatea curentă a firului.	Thread	Debugging și analiză.
setDaemon(boolean value)	Setează firul ca daemon (se închide odată cu programul).	Thread	Fire de fundal (ex.: garbage collector).
isDaemon()	Verifică dacă un fir este daemon.	Thread	Analiză și debugging.

FIR DE EXECUȚIE

1. Să se scrie un program Java care folosește două fire de execuție pentru a afișa în mod concurent numerele de la 1 la 20.
– Primul fir trebuie să afișeze **numerele pare**.
– Al doilea fir trebuie să afișeze **numerele impare**.

Programul trebuie să folosească fie clasa Thread, fie interfața Runnable. Se va observa comportamentul nedeterminist al firelor (ordonarea afișărilor poate varia la fiecare execuție).

```
package executie;
class FirPar extends Thread {
 @Override
 public void run() {
     // Parcurgem numerele de la 1 la 20
     for (int i = 1; i <= 20; i++) {
         // Dacă numărul este par, îl afișăm
         if (i % 2 == 0) {
             System.out.println("Par: " + i);
         } }}}
class FirImpar extends Thread {
 @Override
 public void run() {
     // Parcurgem numerele de la 1 la 20
     for (int i = 1; i <= 20; i++) {
         // Dacă numărul este impar, îl afișăm
         if (i % 2 != 0) {
             System.out.println("Impar: " + i);
         } }}}
public class Problema1 {
 public static void main(String[] args) {
     // Creăm cele două fire
     FirPar firPar = new FirPar();
     FirImpar firImpar = new FirImpar();
     // Pornim firele (se vor executa "în paralel")
     firPar.start();
     firImpar.start(); }}
```

2. Să se realizeze un program Java care simulează un cronometru simplu folosind un fir de execuție. Firul trebuie să afișeze în consolă numărul de secunde trecute de la pornire.

Cerințe:

1. firul va afișa: **"Secunda: X"**, unde X crește de la 1 la 10;
2. între afișări, firul trebuie să se suspende 1 secundă folosind metoda sleep();
3. programul trebuie să ruleze independent de firul principal.

FIR DE EXECUȚIE

```java
package executie;
class FirCronometru extends Thread {
 @Override
 public void run() {
     // Numărăm de la 1 la 10
     for (int secunda = 1; secunda <= 10; secunda++) {
         System.out.println("Secunda: " + secunda);

         try {
             // Punem firul "la somn" 1000 milisecunde = 1 secundă
             Thread.sleep(1000);
         } catch (InterruptedException e) {
             // Dacă firul a fost întrerupt în timpul sleep-ului
             System.out.println("Firul de cronometru a fost întrerupt.");
         }
     }
     System.out.println("Cronometrul s-a terminat.");
 }
}

public class Problema2 {
 public static void main(String[] args) {
     // Creăm firul de cronometru
     FirCronometru cronometru = new FirCronometru();

     // Pornim firul
     cronometru.start();

     // Firul main poate continua să facă alte lucruri aici, dacă vrem
     System.out.println("Firul principal (main) a pornit cronometrul.");
 }
}
```

3. Să se scrie un program Java care calculează suma elementelor unui vector de numere întregi folosind două fire de execuție.

Cerințe:

1. vectorul se împarte în **două jumătăți**;
2. primul fir calculează suma primei jumătăți;
3. al doilea fir calculează suma celei de-a doua jumătăți;
4. firul principal trebuie să folosească metoda join() pentru a aștepta finalizarea celor două fire;
5. la final se va afișa suma totală calculată.

FIR DE EXECUȚIE

```java
package executie;
class FirSuma extends Thread {
 private int[] vector;
 private int start;   // index de început (inclusiv)
 private int end;     // index de sfârșit (exclusiv)
 private int sumaPartiala;
 public FirSuma(int[] vector, int start, int end) {
     this.vector = vector;
     this.start = start;
     this.end = end;
     this.sumaPartiala = 0;
 }
 @Override
 public void run() {
     // Calculăm suma elementelor între start (inclusiv) și end (exclusiv)
     for (int i = start; i < end; i++) {
         sumaPartiala += vector[i];
     }
 }

 public int getSumaPartiala() {
     return sumaPartiala;
 }
}
public class Problema3 {
 public static void main(String[] args) {
     // Definim un vector de exemplu
     int[] v = {1, 2, 3, 4, 5, 6, 7, 8};
     int mijloc = v.length / 2; // Împărțim vectorul în două jumătăți
     // Primul fir: calculează suma primei jumătăți (de la 0 la mijloc-1)
     FirSuma fir1 = new FirSuma(v, 0, mijloc);
     // Al doilea fir: calculează suma celei de-a doua jumătăți (de la mijloc la v.length-1)
     FirSuma fir2 = new FirSuma(v, mijloc, v.length);
```

```java
     // Pornim firele
     fir1.start();
     fir2.start();
     try {
         // Așteptăm ca ambele fire să termine (join)
         fir1.join();
         fir2.join();
     } catch (InterruptedException e) {
         System.out.println("Un fir a fost întrerupt.");
     }

     // Adunăm cele două sume parțiale
     int sumaTotala = fir1.getSumaPartiala() + fir2.getSumaPartiala();

     System.out.println("Suma totală a elementelor din vector este: " + sumaTotala);
 }
}

```