Ministerul Educației și Cercetării al Republicii Moldova Universitate de Stat “Alecu Russo”

Facultatea de Științe Reale, Economice și ale Mediului

# Raport

**“ Fire de execuție”**

Lucrarea de laborator nr. 1

# Platforma JAVA Enterprise II

A efectuat: Cazacu-Condrat Dumitru, Grupa: AW21M

A controlat: dr., conf. univ. Andrian Prisăcaru

**Bălți 2024**

**Introducere**

Acest raport cuprinde descrierea a patru aplicații Java, fiecare utilizând fire de execuție în moduri diferite. Fiecare aplicație ilustrează un concept specific al lucrului cu firele de execuție în Java, de la sincronizare și priorități până la simularea unui cronometru.

**Problema 1: Fire de Execuție Sincronizate**

**Cod Java:**

class Contor {  
 private int numar = 0;  
  
 public synchronized void incrementare() {  
 numar++;  
 System.*out*.println(Thread.*currentThread*().getName() + " - Contor: " + numar);  
 }  
  
 public int getValoare() {  
 return numar;  
 }  
}

class Firul1 extends Thread {  
 private Contor contor;  
  
 public Firul1(Contor contor) {  
 this.contor = contor;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 contor.incrementare();  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

class Firul2 extends Thread {  
 private Contor contor;  
  
 public Firul2(Contor contor) {  
 this.contor = contor;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 contor.incrementare();  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Contor contorComun = new Contor();  
 Firul1 fir1 = new Firul1(contorComun);  
 Firul2 fir2 = new Firul2(contorComun);  
  
 fir1.start();  
 fir2.start();  
  
 try {  
 fir1.join();  
 fir2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Valoarea finală a contorului: " + contorComun.getValoare());  
 }  
}

**Explicație:**

**1. Clasa Contor**

Clasa Contor reprezintă resursa comună pe care cele două fire o vor accesa. Aceasta conține o metodă sincronizată numită incrementare(), care va fi accesată de către firele de execuție. Utilizarea cuvântului cheie synchronized garantează faptul că doar un fir poate accesa metoda la un moment dat, prevenind astfel accesul simultan.

**2. Clasa Firul1 și Firul2**

Cele două fire, Firul1 și Firul2, sunt implementate pentru a accesa contorul comun. Fiecare fir va apela metoda incrementare() a contorului în cadrul unei bucle care se execută de 5 ori. În interiorul metodei run() a fiecărui fir, este implementată o întârziere de 100 ms pentru a permite observarea modificărilor contorului.

**3. Sincronizare**

Datorită utilizării cuvântului cheie synchronized în cadrul metodei incrementare(), sincronizarea este realizată. Acest lucru asigură faptul că, atunci când un fir accesează metoda, celălalt fir va aștepta până când accesul este eliberat. Astfel, se previne actualizarea simultană a contorului de către ambele fire, asigurând o valoare corectă.

**4. Metoda Main**

Metoda main creează instanțele necesare pentru firul 1, firul 2 și contorul comun. Apoi, ambele fire sunt pornite și li se permite să își finalizeze execuția utilizând metoda join(). În acest fel, programul așteaptă finalizarea ambelor fire înainte de a afișa valoarea finală a contorului.

***Prin utilizarea mecanismului de sincronizare din Java, am reușit să prevenim conflictele de acces între două fire de execuție. Sincronizarea este esențială pentru orice program multi-threaded în care mai multe fire accesează o resursă comună, astfel evitând inconsistențele și erorile de concurență.***

**Problema 2: Fire de Execuție Nesincronizate**

**Cod Java:**

class Firul3 extends Thread {  
 private Contor contor;  
  
 public Firul3(Contor contor) {  
 this.contor = contor;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 contor.incrementare();  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

class Firul2 extends Thread {  
 private Contor contor;  
  
 public Firul2(Contor contor) {  
 this.contor = contor;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 contor.incrementare();  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

class Firul1 extends Thread {  
 private Contor contor;  
  
 public Firul1(Contor contor) {  
 this.contor = contor;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 contor.incrementare();  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

class Contor {  
 private int numar = 0;  
  
 public void incrementare() {  
 numar++;  
 System.*out*.println(Thread.*currentThread*().getName() + " - Contor: " + numar);  
 }  
  
 public int getValoare() {  
 return numar;  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Contor contorComun = new Contor();  
 Firul1 fir1 = new Firul1(contorComun);  
 Firul2 fir2 = new Firul2(contorComun);  
 Firul3 fir3 = new Firul3(contorComun);  
  
 fir1.start();  
 fir2.start();  
 fir3.start();  
  
 try {  
 fir1.join();  
 fir2.join();  
 fir3.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Valoarea finală a contorului: " + contorComun.getValoare());  
 }  
}

**Explicație:**

**1. Clasa Contor**

Clasa Contor este utilizată pentru a gestiona o variabilă partajată între cele trei fire de execuție. Spre deosebire de aplicațiile sincronizate, metoda incrementare() nu utilizează cuvântul cheie synchronized, ceea ce permite accesul simultan al mai multor fire asupra acestei metode. Aceasta poate cauza probleme de consistență.

**2. Clasa Firul1, Firul2 și Firul3**

Cele trei fire de execuție (Firul1, Firul2 și Firul3) sunt responsabile pentru apelarea metodei incrementare() a contorului comun. Fiecare fir are o buclă care execută de 5 ori metoda incrementare(), cu o întârziere de 100 ms între iterații. Această întârziere permite observarea efectelor nesincronizării.

**3. Nesincronizare**

Firele de execuție sunt nesincronizate, ceea ce înseamnă că toate pot accesa simultan metoda incrementare(). Acest lucru creează oportunități pentru apariția unor probleme de concurență, cum ar fi incrementarea incorectă a variabilei numar. De exemplu, dacă două fire accesează simultan variabila, valoarea acesteia poate fi incorectă.

**4. Metoda Main**

Metoda main creează instanțele firelor și contorului comun. Cele trei fire sunt pornite și se așteaptă finalizarea acestora folosind metoda join(). Deoarece firele sunt nesincronizate, valoarea finală a contorului poate fi diferită de cea așteptată (în teorie ar trebui să fie 15, însă din cauza concurenței, aceasta poate varia).

***Această aplicație demonstrează riscurile utilizării firelor de execuție nesincronizate în Java. Fără un mecanism de sincronizare adecvat, mai multe fire pot accesa simultan o resursă partajată, ceea ce poate duce la rezultate neprevăzute și comportamente inconsistente. Sincronizarea este esențială pentru evitarea acestor probleme în aplicațiile concurente.***

**Problema 3: Simularea unui Cronometru**

**Cod Java:**

class Cronometru extends Thread {  
 private int secunde = 0;  
 private int minute = 0;  
 private int ore = 0;  
 private boolean inExecutie = true;  
  
 public void oprire() {  
 inExecutie = false;  
 }  
  
 @Override  
 public void run() {  
 while (inExecutie) {  
 try {  
 Thread.*sleep*(1000);  
 secunde++;  
 if (secunde == 60) {  
 secunde = 0;  
 minute++;  
 }  
 if (minute == 60) {  
 minute = 0;  
 ore++;  
 }  
 System.*out*.println(String.*format*("%02d:%02d:%02d", ore, minute, secunde));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

public class Main {  
 public static void main(String[] args) throws InterruptedException {  
 Cronometru cronometru = new Cronometru();  
 cronometru.start();  
  
 Thread.*sleep*(10000);  
 cronometru.oprire();  
 cronometru.join();  
 System.*out*.println("Cronometrul s-a oprit.");  
 }  
}

**Explicație:**

**1. Clasa Cronometru**

Clasa Cronometru extinde clasa Thread și conține logica principală pentru simularea unui cronometru. Aceasta are trei variabile: secunde, minute și ore, care sunt incrementate în funcție de timpul real. Metoda run() gestionează incrementarea acestor variabile la fiecare secundă utilizând metoda Thread.sleep(1000), care pune firul în pauză timp de 1 secundă.

**2. Metoda Oprire**

Metoda oprire() este utilizată pentru a întrerupe execuția cronometru. Aceasta setează variabila inExecutie la false, ceea ce oprește bucla din interiorul metodei run(). În acest fel, cronometru se oprește controlat, fără a bloca aplicația.

**3. Metoda Run**

Metoda run() conține bucla principală care rulează cronometru. Aceasta verifică constant dacă firul trebuie să continue execuția și, la fiecare 1 secundă, crește valoarea secundei. Când secunda atinge 60, se resetează la 0 și minutul este incrementat. Similar, când minutul atinge 60, se resetează și ora este incrementată.

**4. Metoda Main**

În metoda main() din clasa principală, un obiect al clasei Cronometru este creat și firul de execuție este pornit cu start(). Programul simulează cronometrarea timpului pentru 10 secunde, după care cronometru este oprit cu oprire(), iar execuția firului se finalizează.

***Acest program Java ilustrează utilizarea firelor de execuție pentru simularea unui cronometru. Firele permit rularea unor sarcini în paralel cu alte procese ale aplicației, ceea ce face posibilă actualizarea cronometru în timp real. Metoda Thread.sleep() este utilizată pentru a crea pauze precise între actualizări, iar prin gestionarea controlată a opririi firului, se asigură o oprire sigură a cronometru.***

**Problema 4: Fire de Execuție cu Priorități Diferite**

**Cod Java:**

public class Main {  
 public static void main(String[] args) {  
 FirCuPrioritate fir1 = new FirCuPrioritate("Fir 1 - Prioritate Maximă");  
 FirCuPrioritate fir2 = new FirCuPrioritate("Fir 2 - Prioritate Minimă");  
  
 fir1.setPriority(Thread.*MAX\_PRIORITY*);  
 fir2.setPriority(Thread.*MIN\_PRIORITY*);  
  
 fir1.start();  
 fir2.start();  
  
 try {  
 fir1.join();  
 fir2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Execuția firelor s-a încheiat.");  
 }  
}

class FirCuPrioritate extends Thread {  
 private String numeFir;  
  
 public FirCuPrioritate(String numeFir) {  
 this.numeFir = numeFir;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 1; i <= 5; i++) {  
 System.*out*.println(numeFir + " - Iterație: " + i);  
 try {  
 Thread.*sleep*(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println(numeFir + " a terminat execuția.");  
 }  
}

**Explicație:**

**1. Clasa FirCuPrioritate**

Clasa FirCuPrioritate extinde clasa Thread și reprezintă un fir de execuție. Fiecare fir primește un nume și execută o sarcină simplă în buclă. În metoda run(), firul trece prin 5 iterații în care afișează un mesaj cu numărul iterației curente, apoi se oprește pentru 500 ms.

**2. Setarea Priorităților**

În metoda main(), două obiecte FirCuPrioritate sunt create. Un fir primește prioritate maximă utilizând metoda setPriority(Thread.MAX\_PRIORITY), iar celălalt fir primește prioritate minimă cu setPriority(Thread.MIN\_PRIORITY).

**3. Execuția Firelor**

Firele de execuție sunt pornite cu metoda start(), iar execuția lor este sincronizată folosind join() pentru a aștepta terminarea fiecărui fir. Firele execută sarcini simultan, iar prioritățile lor influențează ordinea în care sunt programate de JVM.

***Acest program demonstrează modul în care Java gestionează prioritățile firelor de execuție. Deși prioritățile pot influența ordinea în care firele sunt executate, nu există o garanție absolută că firul cu prioritate mai mare va rula primul. Sistemul de operare și JVM decid ordinea finală de execuție în funcție de disponibilitatea resurselor și alte condiții.***

**Concluzie**

***Aceste patru aplicații Java demonstrează aspectele esențiale ale lucrului cu firele de execuție. De la sincronizare și nesincronizare la priorități și simularea unui cronometru, firele de execuție permit rularea paralelă și eficientă a diferitelor sarcini într-o aplicație, oferind un control detaliat asupra execuției acestora.***