IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

Ioana Leustean

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

https://docs.oracle.com/javase/specs/jls/se23/jls23.pdf

Overview (Java SE 23 & JDK 23) (oracle.com)

➢ Thread

Orice fir de executie (thread) este un obiect al clasei Thread

- Platform thread

  " A platform thread is implemented as a thin wrapper around an operating system (OS) thread.[…] They are suitable for running all types of tasks but may be a limited resource. "

- Virtual thread

  "Virtual threads are suitable for executing tasks that spend most of the time blocked, often waiting for I/O operations to complete. Virtual threads are not intended for long running CPU intensive operations."

➢ API-ul pentru concurenta: java.util.concurrent

Thread (Java SE 22 & JDK 22) (oracle.com)

➢ Thread (platform thread)

Orice fir de executie (thread) este un obiect al clasei Thread

➢ Atributele unui thread

- **ID** - unic pentru fiecare thread
  - este accesat cu getId, nu poate fi modificat

- **Name** - este un String
  - este accesat cu : getName, setName

- **Priority** - un numar intre 1 si 10
  - este accesata cu: getPriority, setPriority
  - in principiu thread-urile cu prioritate mai mare sunt executate primele
  - setarea prioritatii nu ofera garantii in privinta executiei

- **Status** - este accesat cu: getState
  - nu poate fi modificat direct (e.g. nu exista setState)

Thread (Java SE 22 & JDK 22) (oracle.com)

➢ Enum Thread.State

public static enum Thread.State
extends Enum<Thread.State>

Starile posibile ale unui thread:

- NEW: create dar care nu si-a inceput executia
- RUNNABLE: in executie
- BLOCKED: blocat de lacatul unui monitor
- WAITING: asteapta ca un alt thread sa execute o actiune
- TIMED_WAITING: asteapta un alt thread, dar numai un timp limitat
- TERMINATED: thread-ul si-a terminat executia

➢ **Ciclul de viata al unui thread**

- exemplu – HowToDoInJava
- exemplu – javatpoint.com

Thread.State (Java SE 22 & JDK 22) (oracle.com)

➢ **Crearea obiectelor de tip Thread:**

- Metoda directa
  - ca subclasa a clasei Thread
  - implementarea interfetei Runnable

- Metoda abstracta
  - folosind clasa Executors

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

public class Thread
extends Object
implements Runnable
```

- Definirea unui thread ca subclasa a clasei Thread

```
public class HelloThread extends Thread {

   public void run() {
      System.out.println("Hello from a thread!");
   }

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

definim o subclasa a clasei Thread
cu propria metoda run()

definim thread-ul ca un obiect din noua clasa
si il pornim folosind metoda start()

- metoda start() porneste thread-ul creand un fir de exeutie separat
- metoda run() contine ceea ce executa thread-ul dupa apelul metodei start()

- Definirea unui thread ca subclasa anonima a clasei Thread

```
public class HelloThreadAn  {

    public static void main(String args[]) {

            Thread t = new Thread() { public void run() {
                                    System.out.println("Hello from a thread!");
                                            }};
             t.start();
   }}
```

- Definirea unui thread prin implementarea interfetei Runnable

Interfata Runnable contine o singura metoda: run()

```
public class HelloRunnable implements Runnable {

   public void run() {
      System.out.println("Hello from a thread!");
   }

   public static void main(String args[]) {
       Thread t = new Thread (new HelloRunnable());
        t.start();
   }}
```

apelul metodei start() creaza un fir de executie separat, in care este apelata metoda run() a obiectului respectiv

➢ Clasa Thread

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

- Metodele ale instantelor:
  - **run()**
  - **start()**
  - **join()**
  - **join(long millisecunde)**
  - **interrupt()**
  - **boolean isAlive()**

- Metode statice
  (se aplica thread-ului current):
  - **yield()**
  - **sleep(long milisecunde)**
  - **currentThread()**

➢ Clasa Thread

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

Metodele ale instantelor:

- **start()**
  porneste thread-ul intr-un fir de executie separate si invoca **run()**
- **run()**
  este suprascrisa sau apelata din Runnable
- **join(), join(long millisecunde)**
  este invocata de thread-ul curent pe un al doilea thread;
  thread-ul current este blocat pana cand al doilea thread isi termina executia sau
  pana cand expira timpul (milisecunde)
- **interrupt()**
  este folosit in situatia in care un thread cere altui thread t sa isi intrerupa executia;
  intrerupe executia thread-ului t daca acesta este in stare de asteptare, in caz contrar seteaza un flag
- **boolean isAlive()**
  intoarce true atata timp cat thread-ul nu si-a incetat executia

➢ Clasa Thread

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

Metode statice (se aplica thread-ului current):

- **yield()**
  thread-ul cedeaza randul altui thread care are aceeasi prioritate

- **sleep(long milisecunde)**
  thread-ul este suspendat pentru numarul de milisecunde precizat

- **currentThread()**
  intoarce o referinta la thread-ul care invoca metoda

➢ Multi-threading in Java

- O aplicatie Java porneste (prin apelul functiei main) un thread, numit thread-ul principal. Acesta poate porni alte thread-uri.

- Fiecare thread are o prioritate, care poate fi setata de programator. Thread-urile cu prioritate mai mare se executa primele, iar cele cu prioritati egala se executa in ordine FIFO.

- Fiecare thread are propria stiva dar poate accesa si date partajate.

- Exista thread-uri utilizator si thread-uri demon (thread-uri cu prioritate mica, care au rolul de a servi thread-urile utilizator, de exemplu garbadge collector thread; acestea sunt pornite in mare parte de JVM).

- JVM continua sa execute thread-uri pana cand:
  - este apelata metoda *exit* a clasei Runtime
  - toate thread-urile utilizator si-au terminat executia (normal sau printr-o exceptie)

- Executia este nedeterminista

```
public class HelloThread implements Runnable {
      public void run() {
      for (int x = 0; x < 5; x = x + 1)
      System.out.println("Hello from the new thread!");
   }

   public static void main(String args[]) {
       Thread t = new Thread (new HelloThread());
        t.start();
        for (int x = 0; x < 5; x = x + 1)
        System.out.println("Hello from the main thread!");
   }}
```

```
C:\myjava>java HelloThread
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!

C:\myjava>java HelloThread
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the main thread!
```

- join() si InterruptedException

```
public class HelloThread implements Runnable {
        public void run() {
        for (int x = 0; x < 1000; x = x + 1)
        System.out.println("Hello from the new thread!");
    }

    public static void main(String args[]) throws InterruptedException {
         Thread t = new Thread (new HelloThread());
          t.start();
          for (int x = 0; x < 1000; x = x + 1)
          System.out.println("Hello from the main thread!");
          t.join();
    }}
```

Thread-ul current (in exemplu thread-ul principal) intra in stare de asteptare pana cand thread-ul t isi va termina executia; daca threadul t este intrerupt va arunca o exceptie.

- Transmiterea unui parametru catre un thread

```
public static void main(String args[]) {
    Scanner myInput = new Scanner( System.in );
    int n;

    System.out.print( "Enter n " );
    n=myInput.nextInt();

    Thread t = new Thread (new HelloThread(n));
     t.start();

     for (int x = 0; x < n; x = x + 1)
     System.out.println("Hello from the main thread");
}}
```

- Transmiterea unui parametru catre un thread

```
import java.util.Scanner;
public class HelloThread implements Runnable {
   private int n;
   public HelloThread(int n){this.n=n;}
   public void run() {
      for (int x = 0; x < n; x = x + 1)
      System.out.println("Hello from " + Thread.currentThread().getId()+"!");
   }

   public static void main(String args[]) {
      Scanner myInput = new Scanner( System.in );
      int n;
      System.out.print( "Enter n " );
      n=myInput.nextInt();
      Thread t = new Thread (new HelloThread(n));
       t.start();
        for (int x = 0; x < n; x = x + 1)
       System.out.println("Hello from the main thread!");  }}
```

parametrul este transmis constructorului

➢ sleep(ms) si InterruptedException

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

```
public class SleepMessages {

   public static void main(String args[]) throws InterruptedException {

      String importantInfo[] = { “This", “is “, “important"};

      for (int i = 0;  i < importantInfo.length; i++) {

                  //Pause for 4 seconds
                Thread.sleep(4000);

                  System.out.println(importantInfo[i]);
               }
 }}
```

opreste executia threadului **curent** pentru ms milisecunde si arunca exceptie daca threadul este intrerupt

➢ sleep(ms) cu tratarea InterruptedException

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

```
public class MessageLoop implements Runnable {

    public void run() {
        String importantInfo[] = {“This”, “is”, “important”};
        try {
            for (int i = 0;  i < importantInfo.length;i++) {
                Thread.sleep(4000);
                 threadMessage(importantInfo[i]);
            }
        } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
        }}}
```

- Thread-ul secundar este creat prin implementarea interfetei Runnable

➢ sleep(ms) cu tratarea InterruptedException

- Thread-ul principal creaza un al doilea thread si asteapt ca acesta sa isi termine executia.

```
public static  void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
         System.out.format("%s: %s%n", threadName, message);    }

 public static void main(String args[])   throws InterruptedException {

         threadMessage("Starting MessageLoop thread");
         Thread t = new Thread(new MessageLoop());
         t.start();
         threadMessage("Waiting for MessageLoop thread to finish");

         t.join();
        threadMessage("Finally!");}
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java MessageLoop
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
Thread-0: This
Thread-0: is
Thread-0: important
main: Finally!
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java MessageLoop
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
Thread-0: This
Thread-0: is
Thread-0: important
main: Finally!
```

```
public static void main(String args[])   throws InterruptedException {

      threadMessage("Starting MessageLoop thread");
      Thread t = new Thread(new MessageLoop());
      t.start();
      threadMessage("Waiting for MessageLoop thread to finish");

      t.interrupted();

      t.join();
     threadMessage("Finally!");}
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java MessageLoop
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
Thread-0: I wasn't done!
main: Finally!
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
public class SharedCounter {
  static int c = 0;

  public static void main(String[] args)  {
    Thread myThread = new Thread(() -> { for (int x = 0; x < 5000; ++x) c++; });
    myThread.start();

    for (int x = 0; x < 5000; ++x) c--;
          System.out.println("c = " + c);
  }}
```

```
>java SharedCounter.java
c=-5000
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
public class SharedCounter {
 static int c = 0;

 public static void main(String[] args) throws InterruptedException {
  Thread myThread = new Thread(() -> {  for (int x = 0; x < 5000; ++x) c++;  });
  myThread.start();

  Thread.sleep(3000);

 for (int x = 0; x < 5000; ++x) c--;

 System.out.println("c = " + c);
 }}
```

durata executiei?

```
>java SharedCounter.java
c=0
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
public class SharedCounter {
 static int c = 0;

 public static void main(String[] args) throws InterruptedException {
  Thread myThread = new Thread(() -> {
   for (int x = 0; x < 5000; ++x) c++;
  });
  myThread.start();

 for (int x = 0; x < 5000; ++x) c--;

 myThread.join();

System.out.println("c = " + c);
 }}
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
public class SharedCounter {
 static int c = 0;

 public static void main(String[] args) throws InterruptedException {
  Thread myThread = new Thread(() -> { for (int x = 0; x < 5000; ++x) c++; });
  myThread.start();

 for (int x = 0; x < 5000; ++x) c--;

 myThread.join();

System.out.println("c = " + c);
 }}
```

accesul la contor nu este sincronizat!

```
>java SharedCounter.java
c=0
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference  implements Runnable {
static Integer counter = 0;

        public void run () {
           for (int i = 0; i < 5; i++) {
              performTask();
           }}

private void performTask () {
        int temp = counter;
        counter++;
        System.out.println(Thread.currentThread()
                              .getName() + " - before: "+temp+" after:" + counter);}
public static void main (String[] args) {.. }}
```

➢ Comunicarea intre thread-uri – data race

- doua thread-uri care incrementeaza acelasi contor

```
public static void main (String[] args) {
    Thread thread1 = new Thread(new Interference());
    Thread thread2 = new Thread(new Interference());
    thread1.start(); thread2.start();
    thread1.join(); thread2.join();  }
```

```
Thread-1 - before: 1 after:2
Thread-0 - before: 0 after:1
Thread-1 - before: 2 after:3
Thread-0 - before: 3 after:4
Thread-1 - before: 4 after:5
Thread-0 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-0 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-0 - before: 9 after:10
```

```
Thread-0 - before: 0 after:2
Thread-1 - before: 1 after:2
Thread-0 - before: 2 after:3
Thread-0 - before: 4 after:5
Thread-1 - before: 3 after:4
Thread-0 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-0 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-1 - before: 9 after:10
```

data race

➢ Comunicarea intre thread-uri – data race

- doua thread-uri care incrementeaza acelasi contor

```
public static void main (String[] args) {
    Thread thread1 = new Thread(new Interference());
    Thread thread2 = new Thread(new Interference());
    thread1.start(); thread2.start();
    thread1.join(); thread2.join();  }
```

```
Thread-0 - before: 0 after:2
Thread-0 - before: 2 after:3
Thread-0 - before: 3 after:4
Thread-0 - before: 4 after:5
Thread-0 - before: 5 after:6
Thread-1 - before: 1 after:6
Thread-1 - before: 6 after:7
Thread-1 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-1 - before: 9 after:10
```

data race

```
Thread-0 - before: 0 after:2
Thread-1 - before: 1 after:2
Thread-0 - before: 2 after:3
Thread-0 - before: 4 after:5
Thread-1 - before: 3 after:4
Thread-0 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-0 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-1 - before: 9 after:10
```

data race

➢ Mecanismul de sincronizarea thread-urilor

- Fiecare obiect are un lacat intern (*intrinsic lock*, *monitor lock*).

- Un thread care acceseaza un obiect trebuie sa:
  - detina (**aquire**) lacatul intern,
  - acceseaza/modifica datele obiectului,
  - elibereaza (**release**) lacatul obiectului.

- In timpul in care un thread detine lacatul intern al unui obiect, orice alt thread care doreste sa detina (**acquire**) lacatul este blocat.

- Sincronizarea thread-urilor se face cu:

  - Metode sincronizate

```
private synchronized void syncMethod () {
   //codul metodei
 }
```

Cand un thread apeleaza o metoda sincronizata el trebuie sa detina lacatul obiectului caruia ii apartine metoda, executa metoda apoi elibereaza lacatul.

acquire, execute, **release**

Pentru metodele statice, lactul este al obiectului *Class* asociat clasei respective.

➢Sincronizarea thread-urilor

- Metode sincronizate

```
private synchronized void syncMethod () {
   //codul metodei
 }
```

- Instructiuni (blocuri) sincronizate

```
synchronized (object reference){
   // instructiuni
 }
```

se specifica obiectul care detine lacatul

O metoda sincronizata poate fi scrisa ca bloc sincronizat:

```
private void syncMethod () {
     synchronized (this){
       //codul metodei
  }}
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference {

   public static void main (String[] args) throws InterruptedException  {
      Counter c = new Counter();
      Thread thread1 = new Thread(new CounterThread(c));
      Thread thread2 = new Thread(new CounterThread(c));

      thread1.start(); thread2.start();
      thread1.join(); thread2.join();
   }
}
```

```
class CounterThread implements Runnable {
      Counter counter;
      CounterThread (Countercounter){this.counter=counter;}
       public void run(){}
}

Class Counter{…}
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
class CounterThread implements Runnable {
    Counter counter;
    CounterThread (Counter counter) {this.counter=counter;}

      public void run () {
         for (int i = 0; i < 5; i++) {
            counter.performTask();
         }
      }
}
```

```
class Counter{
    private int counter = 0;
    public  void performTask () {}
 }
```

➢ Comunicarea intre thread-uri

- doua thread-uri care incrementeaza acelasi contor

```
class CounterThread implements Runnable {
   Counter counter;
   CounterThread (Counter counter) {this.counter=counter;}
      public void run () {}
}
```

```
class Counter{
private int counter = 0;
public  void performTask () {
      int temp = counter;
      counter++;
      System.out.println(Thread.currentThread()
                        .getName() + " - before: "+temp+" after:" + counter);
  }}
```

➢ Comunicarea intre thread-uri – data race

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference {
    public static void main (String[] args) throws InterruptedException  {
        Counter c = new Counter();
        Thread thread1 = new Thread(new CounterThread(c));
        Thread thread2 = new Thread(new CounterThread(c));
        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
    }
}
```

```
Thread-1 - before: 1 after:2
Thread-0 - before: 0 after:1
Thread-1 - before: 2 after:3
Thread-0 - before: 3 after:4
Thread-1 - before: 4 after:5
Thread-0 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-0 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-0 - before: 9 after:10
```

```
Thread-0 - before: 0 after:2
Thread-1 - before: 1 after:2
Thread-0 - before: 2 after:3
Thread-0 - before: 4 after:5
Thread-1 - before: 3 after:4
Thread-0 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-0 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-1 - before: 9 after:10
```

data race

- Metode sincronizate

```
class CounterThread implements Runnable {
   SCounter scounter;
   CounterThread (SCounter scounter) {this.scounter=scounter;}
      public void run () {}
}
```

```
class SCounter{
private int scounter = 0;
public  synchronized void performTask () {
      int temp = scounter;
      scounter++;
      System.out.println(Thread.currentThread()
                        .getName() + " - before: "+temp+" after:" + scounter);
  }}
```

➢ Metode sincronizate

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference {
   public static void main (String[] args) throws InterruptedException  {
      SCounter sc = new SCounter();
      Thread thread1 = new Thread(new CounterThread(sc));
      Thread thread2 = new Thread(new CounterThread(sc));
      thread1.start(); thread2.start();
      thread1.join(); thread2.join();
   }
}
```

lacatul este pe sc

```
Thread-1 - before: 1 after:2
Thread-1 - before: 2 after:3
Thread-0 - before: 0 after:1
Thread-1 - before: 3 after:4
Thread-0 - before: 4 after:5
Thread-1 - before: 5 after:6
Thread-0 - before: 6 after:7
Thread-1 - before: 7 after:8
Thread-0 - before: 8 after:9
Thread-0 - before: 9 after:10
```

- Metode statice sincronizate

```
class CounterThread implements Runnable {
   SCounter scounter;
   CounterThread (SCounter scounter) {this.scounter=scounter;}
      public void run () {}
}
```

```
class SCounter{
private static int scounter = 0;
public static synchronized void performTask () {
      int temp = scounter;
      scounter++;
      System.out.println(Thread.currentThread()
                        .getName() + " - before: "+temp+" after:" + scounter);
  }}
```

lacatul este pe obiectul din clasa Class asociat cu SCounter

- Blocuri sincronizate

```
class CounterThread implements Runnable {
   SCounter scounter;
   CounterThread (SCounter scounter) {this.scounter=scounter;}
      public void run () {}
}
```

```
class SCounter{
private  int scounter = 0;
private Object counter_lock = new Object();

public  void performTask () {
    synchronized (counter_lock){
      int temp = scounter;
      scounter++;
      System.out.println(Thread.currentThread()
                        .getName() + " - before: "+temp+" after:" + scounter);
  }}}
```

lacatul este pe counter_lock

➢ Mecanismul de sincronizarea thread-urilor prin lacatul intern

- Lacatul este pe obiect.
- Accesul la toate metodele sincronizate este blocat .
- Accesul la metodele nesincronizate nu este blocat.
- Numai un singur thread poate detine lacatul obiectului la un moment dat.
- Un thread detine lacatul intern al unui obiect daca:
  - executa o metoda sincronizata a obiectului,
  - executa un bloc sincronizat de obiect ,
  - daca obiectul este Class, thread-ul executa o metoda static sincronizata .
- Un thread poate face aquire pe un lacat pe care deja il detine (reentrant synchronization):

```
public class reentrantEx {
    public synchronized void met1{}
    public synchronized void met2{  this.met1() ;}
}
```

➢ Modelul de memorie JAVA

- Fiecare thread are propria stiva de executie, heap-ul este comun pentru toate thread-urile.
- Erorile de consistenta a memoriei apar atunci cand thread-uri diferite au vad in mod inconsistent datele comune.
- Accesul la memoria comuna este reglementat de relatia ***happens-before*** care stabileste cand modificarile facute de un thread sunt vizibile altui thread:

  daca actiunea X este in relatie *happens-before* cu actiunea Y atunci
  exita garantia ca thread-ul care executa Y va vedea rezultatele actiunii X

- In absenta relatiei *happens-before* actiunile pot fi reordonate (compiler optimization).

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

https://docs.oracle.com/javase/specs/jls/se23/html/jls-17.html#jls-17.4

➢ Happens-before

daca actiunea X este in relatie *happens-before* cu actiunea Y atunci
exita garantia ca thread-ul care executa Y va vedea rezultatele actiunii X

- Relatia happens-before este o relatie de ordine partial pe toate actiunile unui program.
- Relatia happens-before este tranzitiva.

Reguli care definesc happens-before

Thread unic: in cadrul aceluiasi thread, relatia *happens-before* este stabilita de ordinea actiunilor in program.
Monitor: orice actiune unlock pe un lacat este in relatia *happens-before* cu orice actiune lock ulterioara pe acelasi lacat.
Variabile volatile: scrierea unei variabile volatile este in relatia *happens-before* cu orice citire ulterioara a variabilei.
Thread.start(): actiunea *thread1.start()* este in relatia *happens-before* cu orice actiune din *thread1*
actiunea de pornire a unui thread este in relatia *happens-before* cu orice alta actiune din thread-ul respective
Thread.join(): orice actiune din *thread1* este in relatia *happens-before* cu orice actiune ulterioara lui *thread1.join()*

https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/happens-before.html

Exista reguli care defines relatia happens-before pentru clasele din java.util.concurrent:
https://docs.oracle.com/javase/specs/jls/se23/html/jls-17.html#jls-17.4

➢ Vizibilitate si Atomicitate

Interactiune dintre thread-uri trebuie sa asigure:

- *Excludere mutuala* - numai un thread executa o sectiune critica
  (o parte in care accesul la resurse trebuie sincronizat)
- *Vizibilitate* – modificarile datelor partajate facute de un thread sunt vizibile celorlalte thread-uri

Metodele sincronizate (si lacatele) asigura ambele proprietati, dar au cost computational mai ridicat.

Metode mai simple:

- variabilele **atomice**: operatiile sunt implementate prin instructiuni compare-and-swap
- variabilele **volatile** : asigura vizibilitatea dar nu si atomicitatea

➢ Variabile atomice

sunt implementate folosind instructiuni compare-and-swap care sunt mai rapide

public class AtomicInteger extends Number

Metode:
get(), set(),
incrementAndGet()
addAndGet(int d)
compareAndSet(int old, int new)

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounter {
 private static  AtomicInteger counter = new AtomicInteger();

public static void main(String[] args) throws InterruptedException{
Thread t1 = new Thread(new Runnable() {
                    public void run() {for (int i = 0; i < 1000; i++) counter.incrementAndGet();
});

Thread t2 = new Thread(new Runnable() {
                   public void run() {for (int i = 0; i < 1000; i++) counter.incrementAndGet();}
});

t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(counter.get());
```

- Variabile volatile NU asigura atomicitatea

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounter {
 private static volatile int   counter = 0;

public static void main(String[] args) throws InterruptedException{
Thread t1 = new Thread(new Runnable() {
                    public void run() {for (int i = 0; i < 1000; i++) counter++;  });

Thread t2 = new Thread(new Runnable() {
                    public void run() {for (int i = 0; i < 1000; i++) counter++;}
});

t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(counter);
```

```
C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
2000

C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
2000

C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
2000

C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
1979
```

1979

➢ Variabilele **volatile**

sunt folosite atunci cand exista un thread care le actualizeaza si (eventual) mai multe care le citesc, situatia tipica fiind variabila de control a unui ciclu

```
public class VolatileEx {
static volatile  boolean stop=false;

public static void main(String[] args) throws InterruptedException{
Thread t1 = new Thread(new Runnable() {
                    public void run() {int count =0;
                                        while (!stop) {count++; System.out.println(count); }
                                        System.out.println(count); });

Thread t2 = new Thread(new Runnable() {
                    public void run() {try{Thread.sleep(10);} catch (InterruptedException e) {}
                                        stop=true;}});
t1.start(); t2.start();
 t1.join(); t2.join();
System.out.println("STOP"); }
```

cu **volatile**

```
30606937
STOP
```

fara volatile
nu afiseaza nimic

Pe săptămâna viitoare!