MHP
A PORSCHE COMPANY

METODE AVANSATE DE PROGRAMARE

CONCURENȚA ÎN JAVA

SERGIU LIMBOI, ANDREA IORDACHE, TUDOR COLCERIU

26.11.2025

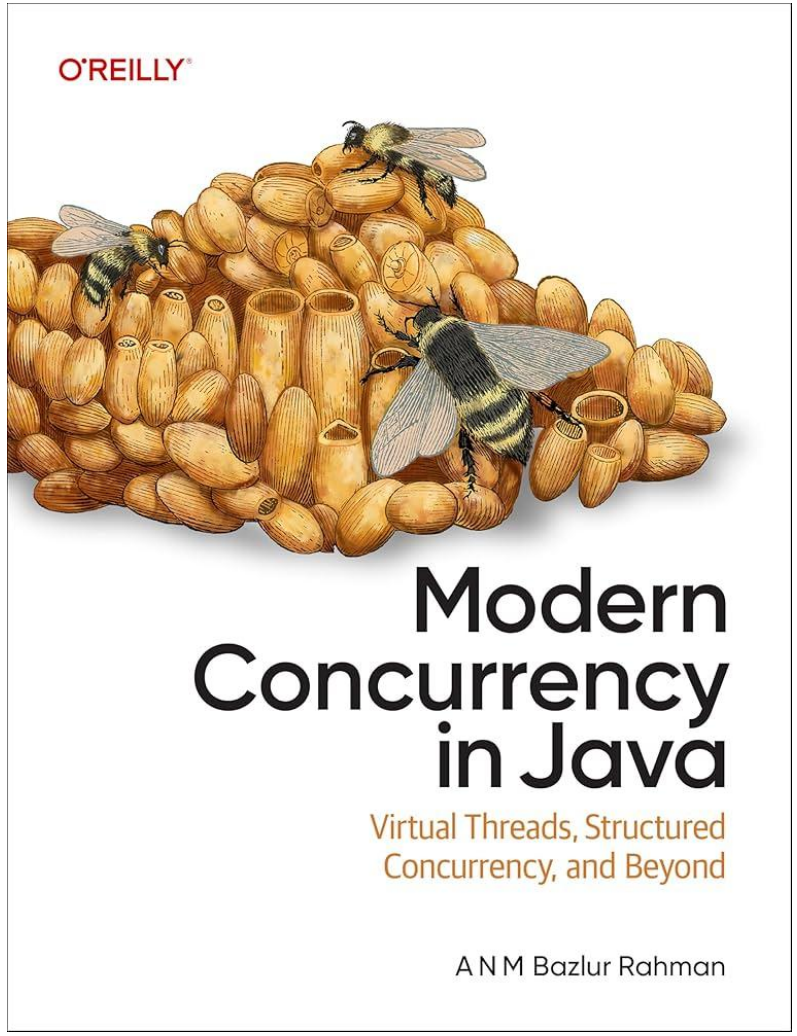
AGENDA

- Concurență și fire de execuție
- Clasele programării multithreading în Java
- Crearea firelor de execuție
- Stările și ciclul de viață al unui thread
- Sincronizarea thread-urilor
- Pachetul java.util.concurrent
- Thread Pools
- Clase de sincronizare
- Colecții thread-safe
- Parallel stream

Concurență și fire de execuție

Programarea concurentă și paralelă

Rolul concurenței



Simplificarea programării şi o mai bună structurare a codului

O aplicație poate să execute mai multe taskuri (unitati logice) *deodată*

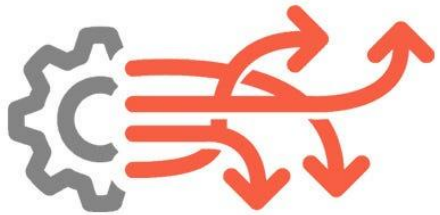
Eficienţă în utilizarea resurselor sistemului (modelul de comunicare multithread înlocuieşte soluţiile clasice de comunicare între procese)

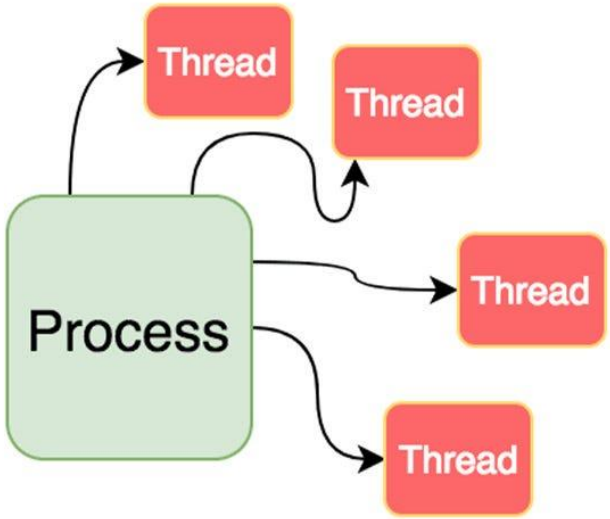
Eficiență în execuția operaţiilor consumatoare de timp - pentru a nu bloca procesul principal.

Îmbunătăţirea interacţiunii la nivel de aplicaţie prin proiectarea de interfeţe performante, responsive (*Interfata cu utilizatorul sa nu"înghețe" la execuția unor taskuri care durează mult (citirea datelor, descărcarea unui fisier) –responsive GUI.)*

Procese și fire de execuție

- În programarea concurentă există două *unități de execuție* de bază: **procese** (processes) și **fire de execuție** (threads)
  - Ambele concepte implică execuţia în “parallel” a unor secvenţe de cod
  - Procesele sunt entităţi independente ce se execută independent şi sunt gestionate de către nucleul sistemului de operare.
  - Firele de execuţie sunt secvenţe ale unui program (proces) ce se execută *aparent* în paralel în cadrul unui singur proces.





Fire de execuție – Threads

Un fir de execuție este o succesiune secvențială de instrucțiuni care se execută *aparent* în paralel în cadrul unui proces (un mecanism care poate executa task-urile asincron).

Sunt denumite "procese usoare" (lightweight processes).

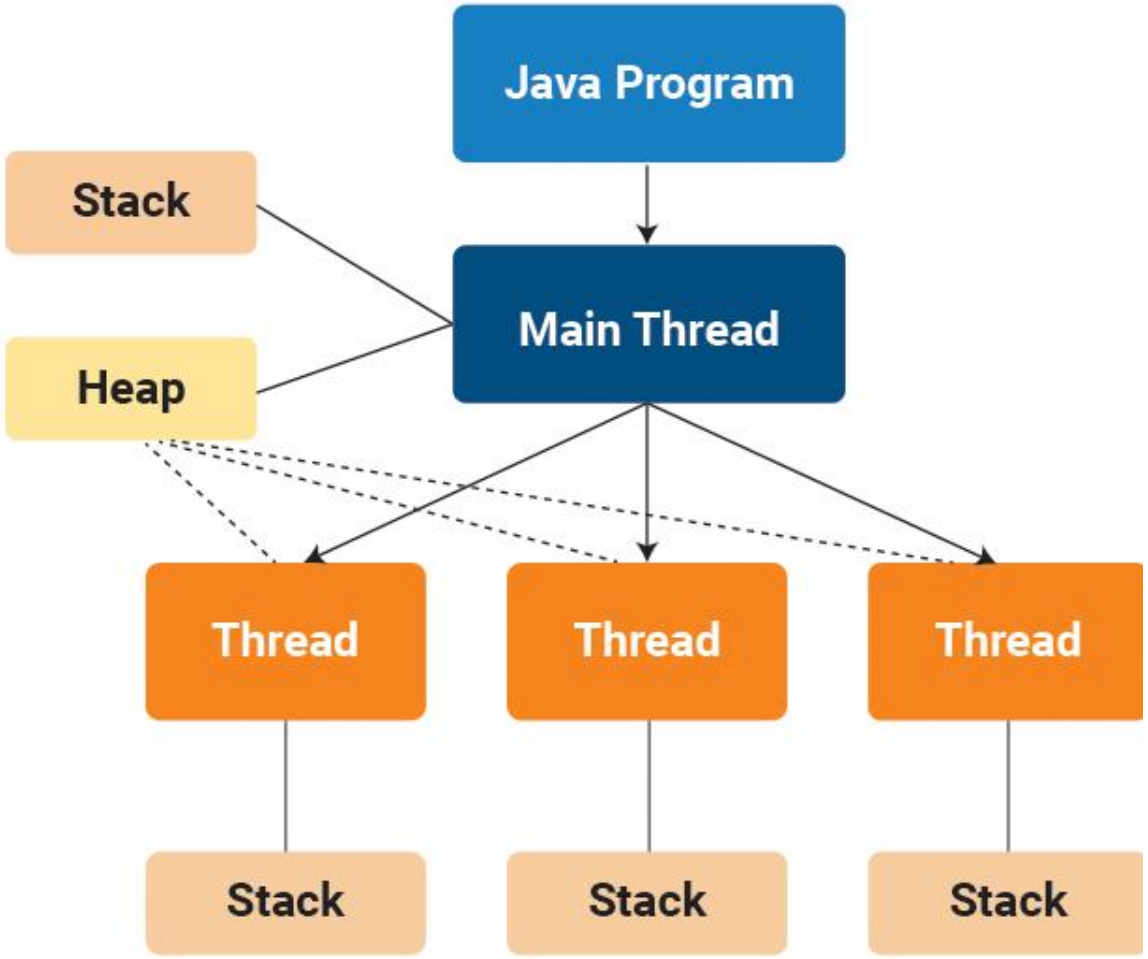
Crearea unui fir de execuție necesită mai puține resurse decât crearea unui proces.

Un fir de execuție există în cadrul unui proces - fiecare proces are cel puțin un fir de execuție

Firele de execuție partajează resursele procesului, incluzând datele, câștigând eficiență în felul acesta, dar comunicarea între acestea devine problematică.

- Toate firele de execuție văd același heap. Java alocă toate obiectele în heap, deci toate firele au acces la toate obiectele.
- Fiecare fir are propria sa stivă. Variabilele locale si parametrii unei metode se alocă în stivă, deci fiecare fir are propriile valori pentru variabilele locale și pentru parametrii metodelor pe care le execută.
- O operație, funcție sau structură de date este thread-safe dacă poate fi folosită în siguranță de mai multe thread-uri în același timp, fără ca datele să fie corupte și fără comportament imprevizibil.

Costul programării multithreading



- Programul devine supraîncărcat (overheaded) crescând foarte mult complexitatea acestuia.
- Costul:
  - creării și distrugerii firelor de execuție.
  - planificării firelor de execuție, a încărcării acestora, stocarea statusurilor dupa fiecare cuantă de timp.
- Costul dacă toate threadurile sunt în acelasi process- acest aspect adaugă o complexitate sporită codului pentru a ne asigura că accesul la zona de date nu ruinează aplicația.
- Depanarea unui program ce rulează pe mai multe fire de execuție este foarte grea; *reproducerea acelorași rezultate planificate este dificilă.*

MHP
A PORSCHE COMPANY

Clasele programării multithreading în Java

Clasa Object

- Metodele clasei Object, *wait(), notify() şi notifyAll(),* sunt utilizate în programarea multithread, având următoarea semnificaţie:
  - *wait()* - pune obiectul în aşteptare până la apariţia unui eveniment (notificare) cu sau fără indicarea duratei maxime de aşteptare
  - *notify()* - permite anunţarea altor obiecte de apariţia unui eveniment
  - *notifyAll()* - implementează notificarea mai multor obiecte la apariţia unor evenimente

Clasa Thread (java.lang.Thread)

- Principalele câmpuri şi metode ale clasei Thread sunt:

- *void start()* - lansează în execuţie noul thread, moment în care execuţia programului este controlată de cel puţin două threaduri: threadul curent ce execută metoda start şi noul thread ale cărui instrucţiuni sunt definite în metoda run()
- *void run()* – defineşte corpul threadului nou creat, întreaga activitate a threadului va fi descrisă prin suprascrierea acestei metode
- *static void sleep()* – pune în aşteptare threadul curent pentru un anumit interval de timp (msecs)
- *void join ()* - se aşteaptă ca obiectul thread ce apelează această metodă să se termine
- *suspend()* - suspendare temporară a threadului (*resume()* este metoda dual ă ce relansează un thread suspendat (implementările JDK ulterioare versiunii 1.2 au renunţat la utilizarea lor)

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

Clasa Thread

- *yield()* - realizează cedarea controlului de la obiectul thread, planificatorului JVM pentru a permite unui alt thread să ruleze
- *void interrupt()* – trimite o întrerupere obiectului thread ce o invocă (setează un flag de întrerupere a threadului activ).
- *static boolean interrupted()* - testează dacă threadul curent a fost întrerupt, resetează starea interrupted a threadului current
- *boolean isInterrupted ()* - testează dacă un thread a fost întrerupt fără a modifica starea threadului
- *boolean isAlive()* - permite identificarea stării obiectului thread
- *void setDaemon(boolean on)* - apelată imediat înainte de start permite definirea threadului ca daemon. Un thread este numit daemon, dacă metoda lui run conţine un ciclu infinit, astfel încat acesta nu se va termina la terminarea threadului părinte.
- *getPriority()* - returnează prioritatea threadului curent
- *setPriority(newPriority)* -permite atribuirea pentru threadul curent a unei priorităţi dintr-un interval.

Metodele stop(), suspend() şi resume(), definite în versiuni anterioare au fost eliminate deoarece în cazul unei proiectări defectuoase a codului pot provoca blocarea acestuia .

Crearea firelor de execuție

Crearea firelor de execuție în Java

MHP
A PORSCHE COMPANY

În orice program Java, aflat în execuție, JVM creează automat un obiect fir de execuție, având rolul de a apela metoda *main*.

Firele de execuție pot fi create și explicit de către programator:

- Implement the Runnable Interface (preferred)
- Extend the Thread class

Creare Thread- extindem clasa Thread

- **Important:**
  - run() = ce face thread-ul.
  - start() = pornește un **nou** thread și va apela run() în acel thread.

- De ce nu e ideal în practică:
  - Java nu permite moștenire multiplă → dacă extinzi Thread, nu mai poți extinde altă clasă.

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Rulez în thread: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // PORNEȘTE thread-ul (NU run())
    }
}
```

Creare Thread- implementare Runnable (varianta clasică)

- Clasic

```
class MyTask implements Runnable {
    @Override
    public void run() {
        System.out.println("Task în: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        Runnable task = new MyTask();
        Thread t = new Thread(task, "Worker-1");
        t.start();
    }
}
```

- Cu lambda

```
public class Main {
    public static void main(String[] args) {
        Runnable task = () -> {
            System.out.println("Hello din " + Thread.currentThread().getName());
        };

        Thread t = new Thread(task, "Worker-1");
        t.start();
    }
}
```

Creare Thread- Executors/ThreadPool (best practice în 2025)

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        // Pool fix de 4 thread-uri
        ExecutorService executor = Executors.newFixedThreadPool(4);

        for (int i = 0; i < 10; i++) {
            int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId +
                        " in " + Thread.currentThread().getName());
            });
        }

        executor.shutdown(); // nu mai acceptă task-uri noi
        executor.awaitTermination(1, TimeUnit.MINUTES); // așteaptă să termine
    }
}
```

- În loc să creezi zeci/sute de new Thread(...), folosești un **pool** de thread-uri gestionat de JVM.

MHP
A PORSCHE COMPANY

Stările și ciclul de viață al unui thread

Stările unui thread

- NEW -Thread-ul a fost creat, dar **nu a început încă** execuția.

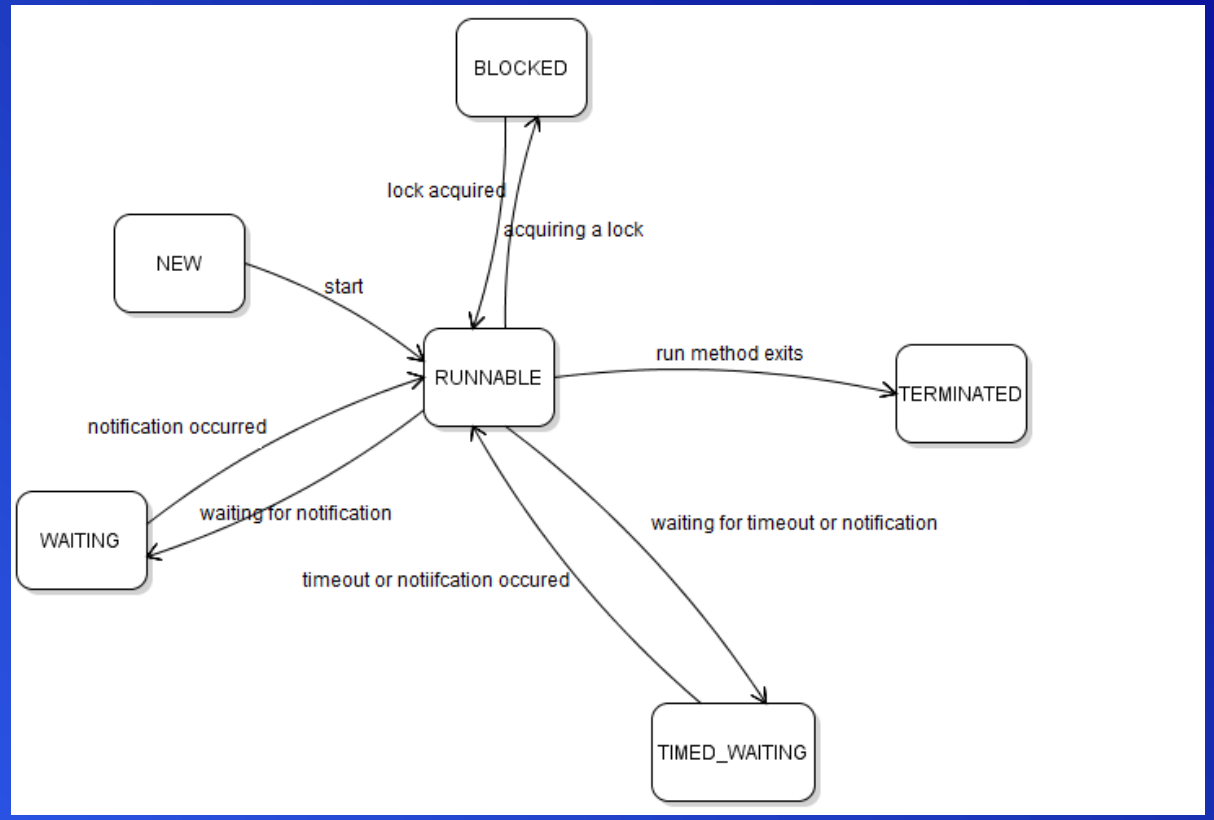
```
Thread t = new Thread(() -> {});
```

- RUNNABLE- Thread-ul este **pregătit să ruleze** sau **rulează efectiv** pe CPU.

- BLOCKED- Thread-ul **așteaptă un lock** (monitor) pentru a intra într-o secțiune sincronizată (synchronized).

```
synchronized(obj) {
    // alt thread ține deja lock-ul → thread-ul B devine BLOCKED
}
```

- WAITING -Thread-ul așteaptă **fără timeout**, până când alt thread îl trezește. Când apare WAITING:
  - Object.wait()
  - Thread.join() fără timeout



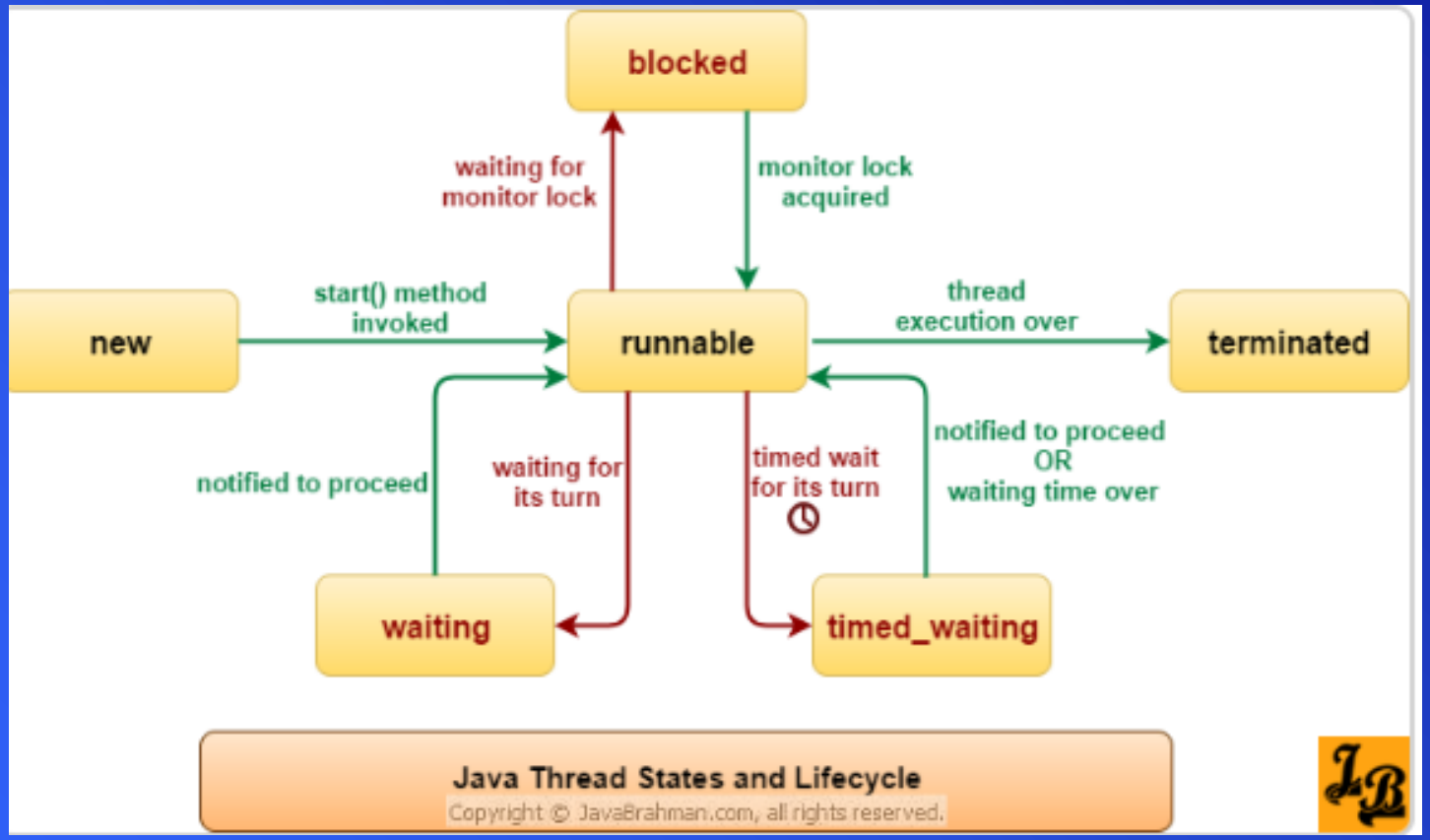
Stările unui thread

```java
public class Main {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(() -> {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {}
        });

        System.out.println(t.getState()); // NEW

        t.start();
        System.out.println(t.getState()); // RUNNABLE

        Thread.sleep(100);
        System.out.println(t.getState()); // TIMED_WAITING (sleep)

        t.join();
        System.out.println(t.getState()); // TERMINATED
    }
}
```

- TIMED_WAITING- Thread-ul așteaptă **cu timeout. Apare în Thread.sleep(1000), wait(1000), join(1000).**

- TERMINATED- Thread-ul și-a **terminat execuția**:
  - run() s-a încheiat normal
  - sau a aruncat o excepție
  - Este starea finală a unui thread.

Ciclul de viață al unui thread & stările unui thread



MHP
A PORSCHE COMPANY

Sincronizarea thread-urilor

Sincronizarea thread-urilor

- O caracteristică importantă a firelor de executie este că ele văd același heap.
- Acest lucru poate duce la multe probleme în cazul în care un obiect oarecare (care e o resursă comună pentru toate firele) este accesat din două fire de executie diferite.
- Metode de sincronizare/cooperare:
  - Mecanismul de excludere mutuală (mutex-semafor)
  - Comunicare prin condiții
- Sincronizarea este capacitatea de a controla accesul la multiple threaduri pentru resurse partajate.
- Trebuie să ne asigurăm că o resursă este accesată de un singur thread într-un anumit moment.

Excluderea mutuală

- **Excluderea mutuală** înseamnă că *doar un singur thread* poate executa o anumită secțiune de cod la un moment dat.
  Acea secțiune de cod se numește **critical section** – o zonă unde se lucrează cu date comune (shared data).

- Dacă nu controlăm accesul, thread-urile pot modifica simultan aceleași variabile → apar erori greu de găsit (**race conditions**).

- Cum facem excludere mutuală în Java?
  - **synchronized (cel mai simplu și frecvent)**
    - Poate fi aplcat pe metode și blocuri :
      - synchronized void metoda() {...}
      - synchronized (lockObject) {...}
  - **ReentrantLock**
    - mai flexibil decât synchronized
    - oferă metode precum tryLock(), condiții (Condition), fairness, timeout etc.

Exemplu synchronized

- Două thread-uri încearcă să depună bani într-un cont bancar în același timp.
  Folosim excludere mutuală pentru a evita modificări simultane ale soldului.

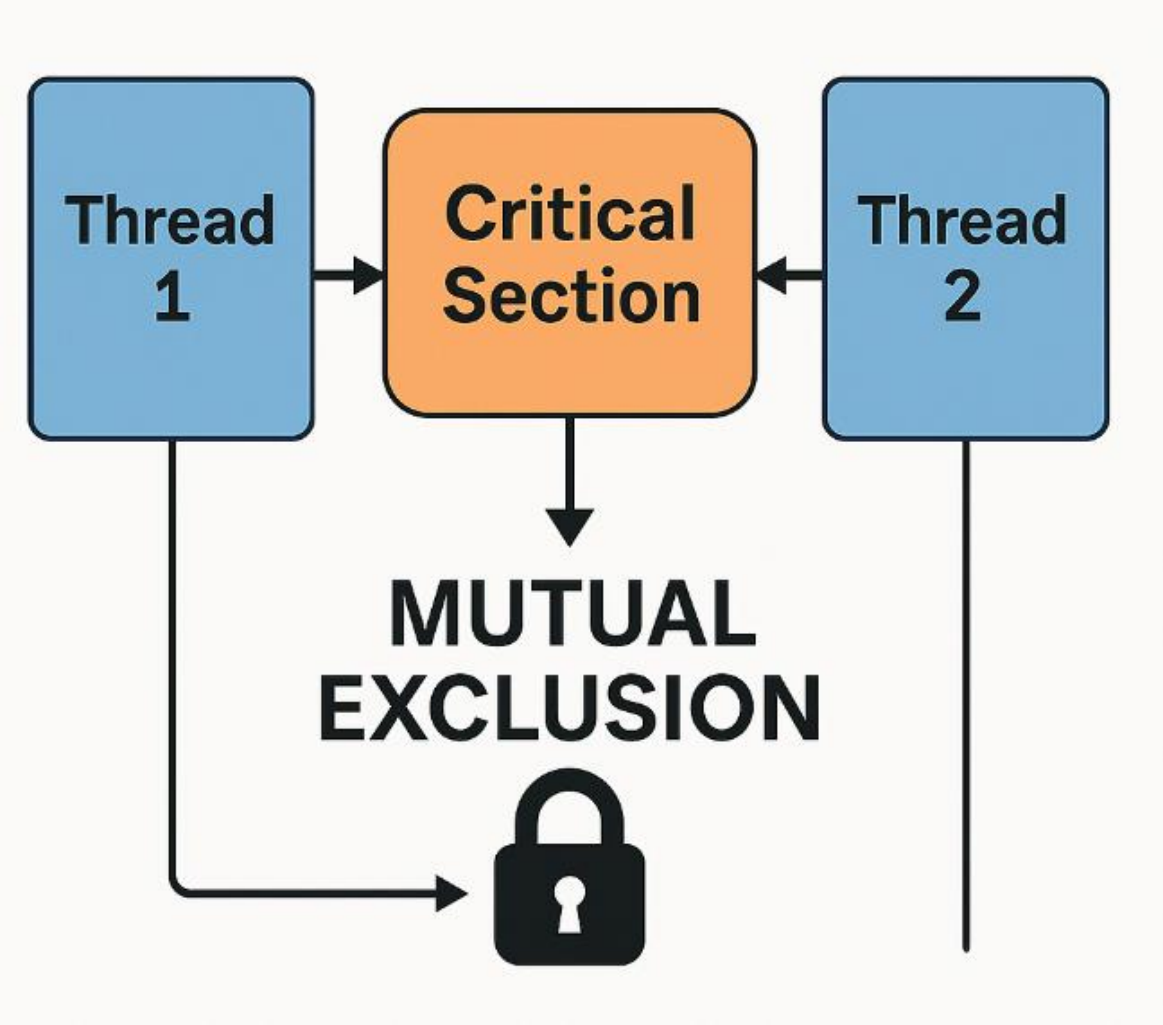
```
class BankAccount {  2 usages
    private int balance = 0;  4 usages

    // Metoda critică: modifică variabila shared "balance"
    public synchronized void deposit(int amount) {  2 usages
        int newBalance = balance + amount;

        // Simulăm un task care durează (pentru a evidenția problema fără sincronizare)
        try {
            Thread.sleep( millis: 100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        balance = newBalance;
        System.out.println(Thread.currentThread().getName() + " a depus " + amount + ", noul sold = " + balance);
    }

    public int getBalance() {  1 usage
        return balance;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        // Creăm două thread-uri care depun bani simultan
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.deposit(100);
            }
        }, "Thread-1");

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.deposit(100);
            }
        }, "Thread-2");

        // Pornim thread-urile
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Sold final = " + account.getBalance());
    }
}
```

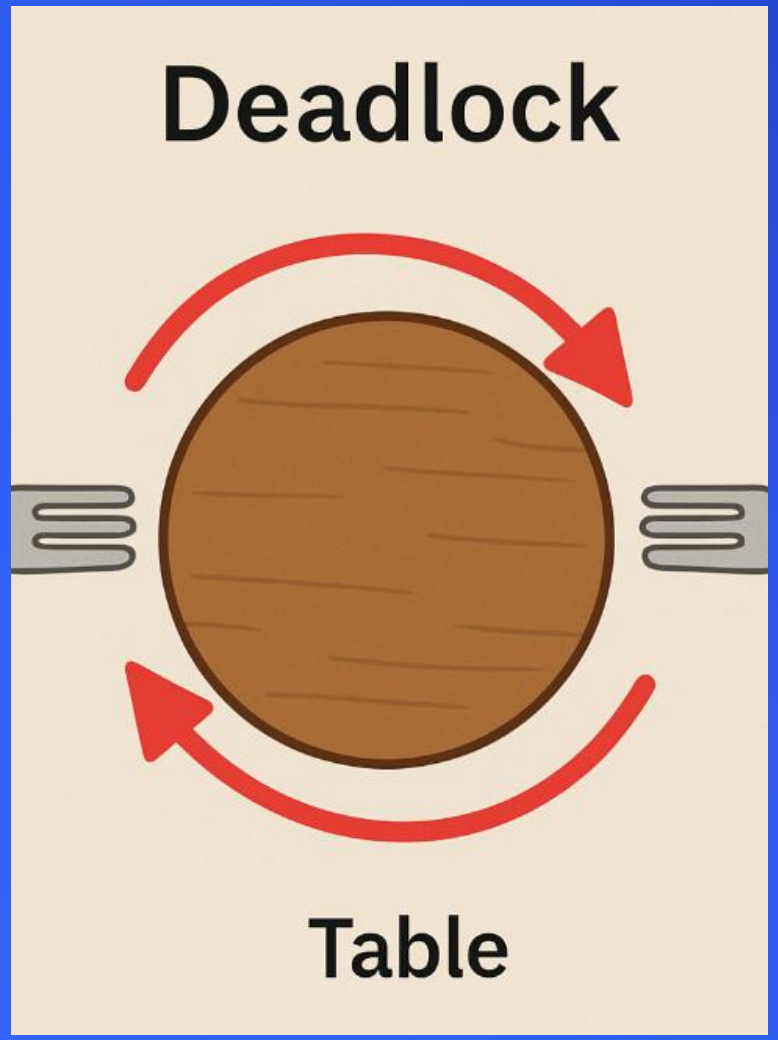
Exemplu synchronized

- Ambele thread-uri încearcă să depună bani în același timp
- Fără sincronizare → soldul final ar fi greșit (race condition)
- Cu synchronized → doar un thread intră în metoda deposit() la un moment dat
- Soldul este actualizat correct

Situație	Ce se întâmplă	Rezultat
Fără synchronized	Threadurile lucrează simultan → se suprascriu	❌ Sold greșit (ex: 500)
Cu synchronized	Threadurile intră pe rând → actualizări corecte	✔️ Sold corect: 1000



Impas (deadlock)



- Excluderea mutuală rezolvă doar o singură problemă:

  “Doar un thread poate intra în secțiunea critică la un moment dat.”

- Asta previne:
  - suprascrieri de date,
  - race conditions,
  - inconsistențe.

- DAR… în programele reale, thread-urile nu au doar **o** secțiune critică.
  De multe ori au **mai multe resurse diferite** (de ex. două lock-uri, două conturi, două fișiere etc).

- **Deadlock** = două (sau mai multe) thread-uri se blochează reciproc, fiecare așteptând ceva ce celălalt deține.

- Niciun thread nu poate continua → programul ”îngheață”.

Cum rezolvăm deadlock-ul?

- Folosește aceeași ordine globală pentru lock-uri
- Toate thread-urile trebuie să ia lock-urile în aceeași ordine

```
Object lock1 = new Object();
Object lock2 = new Object();

void safeMethod() {
    synchronized (lock1) {     // mereu lock1 primul
        synchronized (lock2) { // apoi lock2
            // code
        }
    }
}
```

- Folosește structuri lock-free/ atomice

- AtomicInteger
- AtomicBoolean
- ConcurrentHashMap
- Etc.

- Folosește un singur lock, nu multe

```
synchronized(lock1) { ... }
synchronized(lock2) { ... }
```


```
synchronized(globalLock) { ... }
```

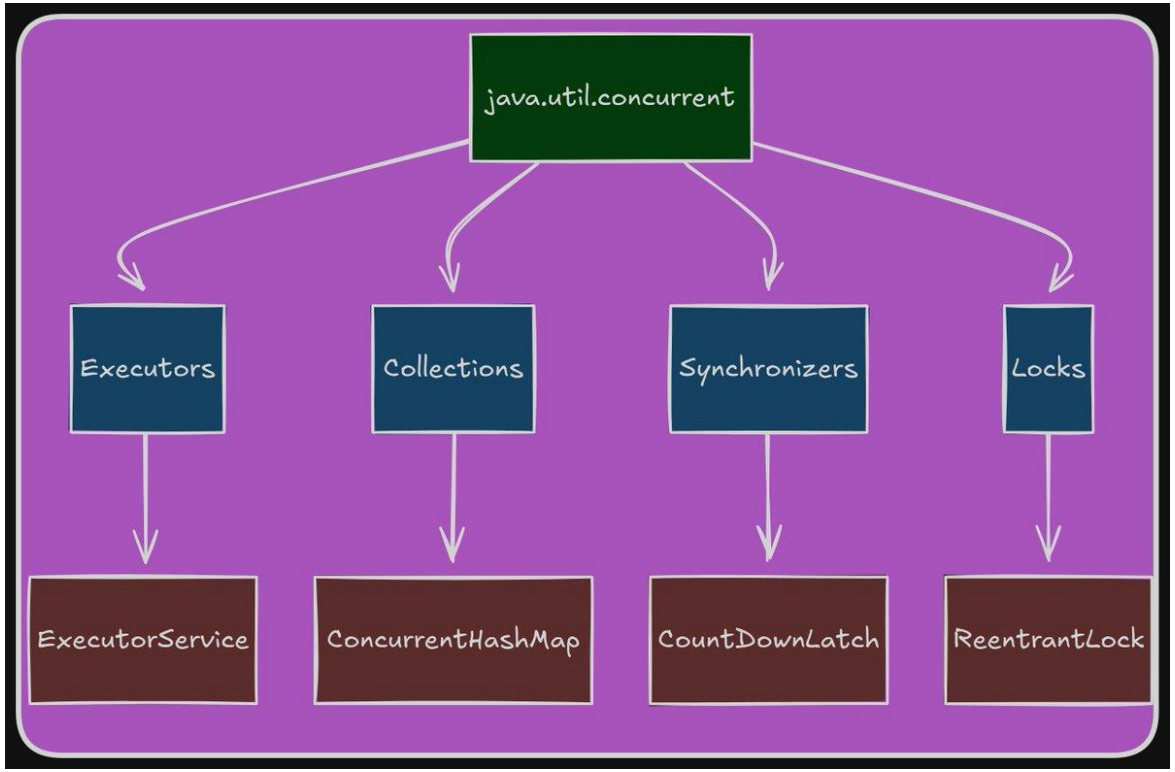
- Folosește tryLock() cu timeout
- Funcționează doar cu ReentrantLock

```
Lock lock1 = new ReentrantLock();
Lock lock2 = new ReentrantLock();

void safeMethod() throws InterruptedException {
    if (lock1.tryLock(100, TimeUnit.MILLISECONDS)) {
        try {
            if (lock2.tryLock(100, TimeUnit.MILLISECONDS)) {
                try {
                    // Critical section
                } finally {
                    lock2.unlock();
                }
            } else {
                System.out.println("Nu pot lua lock2 → evit deadlock");
            }
        } finally {
            lock1.unlock();
        }
    } else {
        System.out.println("Nu pot lua lock1 → evit deadlock");
    }
}
```

Pachetul java.util.concurrent

Pachetul java.util.concurrent

- aduce un set bine testat și foarte performant de funcții și structuri de date pentru concurență care ajută programatorul, cu un efort redus de programare, sa proiecteze aplicatii concurente care:
  - Au performanță ridicată
  - Sunt de încredere
  - Ușor de întreținut



Pachetul java.util.concurrent

MHP
A PORSCHE COMPANY

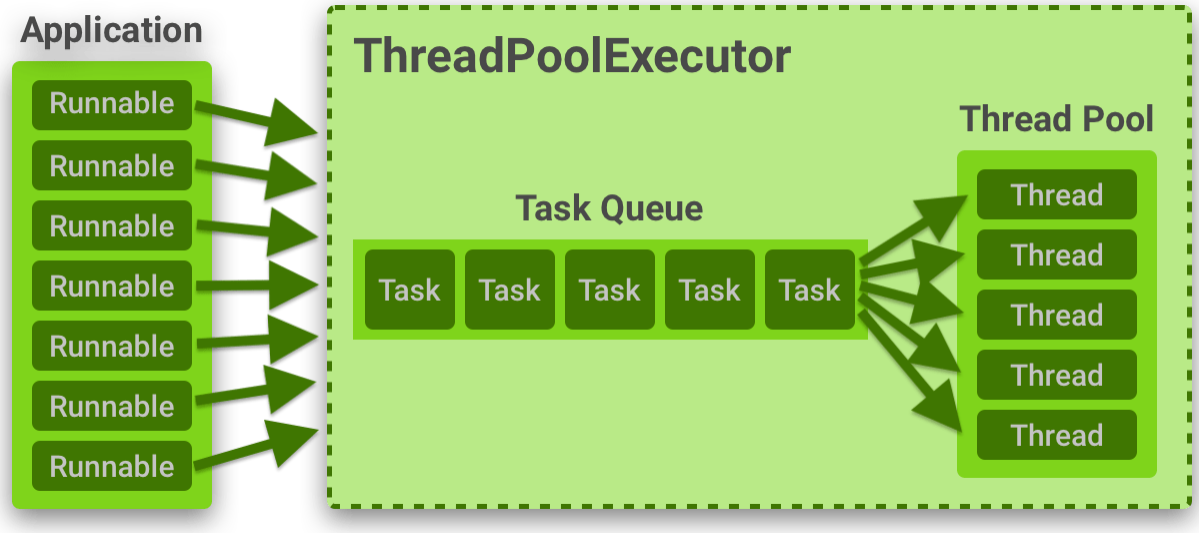
- Îmbunătățirile aduse din perspectiva suportului pentru concurență sunt structurate în 3 categorii:
  - **Modificări la nivelul mașinii virtuale java:** Procesoarele moderne oferă suport hardware pentru concurență, de obicei în forma unor instrucțiuni *compare-and-swap* (CAS)...
  - **Clase utilitare de nivel scăzut – locks și variabile atomice:** De exemplu clasa *ReentrantLock* oferă funcționalitate asemănătoare cu soluția *synchronized*, dar cu un control mai bun asupra blocării (timed locks, lock polling, etc.) și o mai bună scalabilitate.
  - **Clase utilitare la nivel înalt:** Clase care implementează: mutexuri, semafoare, locks, bariere, thread pools și colecții thread-safe. Acestea sunt oferite dezvoltatorilor de aplicații pentru a construi diverse soluții.

MHP
A PORSCHE COMPANY

Thread Pools

Ce este un thread pool?

- Un thread pool este un set de thread-uri pre-create, gata să execute task-uri.
- **Avantaje:**
  - Nu mai creezi/distrugi thread-uri non-stop → **performanță mai bună**.
  - Controlezi numărul maxim de thread-uri → **eviți să omori sistemul**.
  - Ai API mai simplu pentru a trimite task-uri.
  - Poți planifica, obține rezultate, trata excepții, etc.
- Work queue este o coadă de taskuri ce trebuie procesate.
- Thread pool-ul extrage task-uri din coadă și le execute.



Executor & Executor Service

- Executor – interfață simplă, cu o singură metodă: execute(Runnable command).

```
private final ExecutorService executor = Executors.newSingleThreadExecutor();
```

- ExecutorService – extinde Executor și adaugă:
  - submit(...) – pentru Runnable/Callable, întoarce Future.
  - shutdown(), shutdownNow() – pentru închiderea pool-ului.
  - invokeAll(), invokeAny(), etc.

```
executor.submit(
    () -> {
      try {
        threadUtils.setCurrentThreadContext(threadContext);
        exportPricesService.setPriceTransferStatusFor(request.getData());
      } catch (Exception e) {
        log.error("There is an exception when price exports status is set", e);
      }
    });
```

- Clasa utilitară java.util.concurrent.Executors oferă metode statice pentru a crea thread pools

Tipuri de thread pool

- **Fixed thread pool** – specifică număr fix de thread-uri, daca vine un număr mai mare decât numărul specificat, atunci task-urile se pun în coadă
  - **Util** pentru sarcini CPU sau când vrei control strict.

```
ExecutorService pool = Executors.newFixedThreadPool(4);
```

- **Cached thread pool-**creează thread-uri noi după nevoie, re-folosește thread-uri vechi dacă sunt libere, poate crește până la un număr foarte mare de thread-uri (atenție!).

```
ExecutorService pool = Executors.newCachedThreadPool();
```

  - **Util** pentru multe task-uri mici, de scurtă durată, mai ales I/O.
- **Single thread executor-** un singur thread, task-urile sunt executate **secvențial**, în ordinea sosirii.
  - **Util** când vrei să eviți probleme de sincronizare, dar să ai totuși o coadă de task-uri.

```
ExecutorService pool = Executors.newSingleThreadExecutor();
```

- **Scheduled thread pool**- pentru task-uri **planificate în timp** (la un delay, periodic- la fiecare X secunde)

```
scheduler.schedule(
    () -> System.out.println("Rulez după 3 secunde"),
    3, TimeUnit.SECONDS
);

scheduler.scheduleAtFixedRate(
    () -> System.out.println("Mesaj periodic"),
    1, 5, TimeUnit.SECONDS // initialDelay=1s, period=5s
);
```

Exemplu Thread Pool

MHP
A PORSCHE COMPANY

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExample {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService pool = Executors.newFixedThreadPool( nThreads: 4);

        for (int i = 1; i <= 10; i++) {
            int taskId = i;
            pool.submit(() -> {
                System.out.println("Task " + taskId + " rulează în thread " +
                        Thread.currentThread().getName());
                try {
                    Thread.sleep( millis: 1000); // simulez o operație care durează
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
                System.out.println("Task " + taskId + " s-a terminat");
            });
        }

        // Semnalăm că nu mai trimitem task-uri noi
        pool.shutdown();

        // Așteptăm să termine toate task-urile
        if (!pool.awaitTermination( timeout: 30, TimeUnit.SECONDS)) {
            pool.shutdownNow();
        }

        System.out.println("Toate task-urile s-au terminat.");
    }
}
```

```java
public class ThreadPoolUtils {  3 usages

  private static final int NUMBER_OF_THREADS = 10;  1 usage
  private static final ExecutorService executorService =  2 usages
      Executors.newFixedThreadPool(NUMBER_OF_THREADS);

  /**
   * This method executes the given {@link Runnable} asynchronously inside {@link ThreadPoolUtils#executorService}.
   *
   * @param runnable the {@link Runnable} that should be executed asynchronously.
   */
  public static void execute(Runnable runnable) { executorService.execute(runnable); }
}
```

Runnable, Callable, Future

- Runnable este o **interfață funcțională** cu o singură metodă: void run()

```
Runnable task = () -> {
    System.out.println("Rulez în thread: " + Thread.currentThread().getName());
};
```

- Callable –versiunea „avansată” a lui Runnable
- Callable<V> – task cu rezultat și cu excepții checked:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

- Future<V> este **un obiect care reprezintă rezultatul unei operații care poate nu s-a terminat încă**.
- Future.get() blochează **până când task-ul e gata** și returnează rezultatul

```
import java.util.concurrent.*;

public class CallableExample {
    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(2);

        Callable<Integer> sumTask = () -> {
            int sum = 0;
            for (int i = 1; i <= 100; i++) {
                sum += i;
            }
            return sum;
        };

        Future<Integer> future = pool.submit(sumTask);

        // faci alte lucruri...

        Integer rezultat = future.get(); // blocant: așteaptă până e gata
        System.out.println("Suma este: " + rezultat);

        pool.shutdown();
    }
}
```

Runnable, Callable, Future

- **Runnable** → „du-te și fă asta, nu mă interesează ce obții”
- **Callable** → „du-te, calculează ceva și adu-mi rezultatul”
- **Future** → „îți dau un tichet, cu el poți ridica rezultatul când e gata”

Cazurile ideale

✔ **Runnable**

- trimiți un email
- scrii log-uri
- faci cleanup
- rulezi sarcini scurte, fără rezultat

✔ **Callable**

- calcule numerice
- citire din fișiere / procesări
- cereri HTTP (returnezi răspunsul)
- orice sarcină care „produce ceva”

✔ **Future**

- când vrei să:
  - aștepți un rezultat,
  - verifici progresul,
  - anulezi un task.

Completable Future

- CompletableFuture (Java 8+) este o implementare avansată a interfeței **Future** + **CompletionStage**.

- Este o abstracție pentru **programare asincronă** și **paralelă** care permite:
  - executarea de task-uri în fundal,
  - obținerea rezultatului *fără blocare*,
  - atașarea de callback-uri (ca în JavaScript Promises),
  - combinarea și compunerea rezultatelor,
  - tratarea elegantă a erorilor,
  - orchestrarea mai multor task-uri în paralel.

- **Gândește-l ca un „Future cu superputeri”.**

- **runAsync()- Runnable**
- **supplyAsync()- Supplier**

```
CompletableFuture<Void> cf = CompletableFuture.runAsync(() -> {
    System.out.println("Task fără rezultat, rulează în alt thread");
});
```

```
CompletableFuture<Integer> cf = CompletableFuture.supplyAsync(() -> {
    return 10 + 20;
});
```

Clase de sincronizare

Clase de sincronizare

- Exemple de clase de sincronizare: Semaphore, Mutex, CyclicBarrier, CountDownLatch, etc.
- **Semaphore:** doar un anumit nr de thread-uri pot avea simultan acces concurent la o resursă.
  - Implementează un semafor clasic, care are un număr dat de permisii ce pot fi cerute și eliberate.
  - Este folosit pentru a restricționa numărul de thread-uri ce pot avea simultan acces concurent la o resursă.
  - Înainte să obțină o resursă, un thread trebuie să obțină permisiunea de la semafor – adică resursa este disponibilă.

  - Apoi, când termină de utilizat resursa respectivă, thread-ul se întoarce la semafor pentru a semnala că aceasta este din nou disponibilă.

Exemplu Semaphore

```java
import java.util.concurrent.Semaphore;

class Parcare {
    // Avem 2 locuri disponibile
    private final Semaphore locuri = new Semaphore(2);

    public void parcheaza(String masina) {
        try {
            System.out.println(masina + " încearcă să intre în parcare...");

            // Așteaptă până când există un loc liber
            locuri.acquire();
            System.out.println(masina + " a parcat.");

            // Simulăm timpul petrecut în parcare
            Thread.sleep(2000);

            System.out.println(masina + " a ieșit din parcare.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // Eliberăm locul la ieșire
            locuri.release();
        }
    }
}
```

```java
public class ExempluSemaphore {
    public static void main(String[] args) {
        Parcare parcare = new Parcare();

        for (int i = 1; i <= 5; i++) {
            String numeMasina = "Mașina " + i;

            new Thread(() -> parcare.parcheaza(numeMasina)).start();
        }
    }
}
```

- Creează un semafor cu **2 permise** → **maxim 2 thread-uri în același timp**.
- locuri.acquire();
  Thread-ul **așteaptă** dacă nu există permise disponibile.
- locuri.release();
  Thread-ul **eliberează** un loc când termină.
- Programul lansează 5 thread-uri, dar în același timp doar 2 "mașini" pot fi în parcare.

Mutex

- Mutex - un caz special de semafor, cu o singură permisie (permite acces exclusiv)
- Semaphore mutex = new Semaphore(1);
  → un singur permis → acces exclusiv → mutex.
- mutex.acquire();
  → thread-ul așteaptă dacă mutex-ul este deja luat.
- Codul dintre acquire() și release() este **secțiunea critică**.
- mutex.release();
  → eliberează mutex-ul pentru următorul thread.

```java
class Contor {
    // Mutex: doar un singur thread poate intra în secțiunea critică
    private final Semaphore mutex = new Semaphore(1);

    private int valoare = 0;

    public void increment(String numeThread) {
        try {
            // Thread-ul cere acces la secțiunea critică
            mutex.acquire();

            System.out.println(numeThread + " a intrat în secțiunea critică.");

            // Secțiune critică
            int copie = valoare;
            Thread.sleep(500); // simulăm timp de lucru
            valoare = copie + 1;

            System.out.println(numeThread + " a incrementat valoarea la: " + valoare);

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // Thread-ul eliberează mutex-ul
            mutex.release();
            System.out.println(numeThread + " a ieșit din secțiunea critică.");
```

Cyclic Barrier

- CyclicBarrier - Așteaptă până când **un anumit număr de thread-uri** ajung într-un punct comun.
- După ce *toate* thread-urile au ajuns, bariera se resetează automat și poate fi reutilizată.
- De ce cyclic?
  - După ce trece primul ciclu, bariera se resetează automat
  - Poate fi folosită din nou pentru sincronizări repetate

```java
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

class Lucrator implements Runnable {  3 usages
    private final CyclicBarrier barrier;  2 usages
    private final String nume;  5 usages

    public Lucrator(CyclicBarrier barrier, String nume) {  3 usages
        this.barrier = barrier;
        this.nume = nume;
    }

    @Override
    public void run() {
        try {
            System.out.println(nume + " lucrează la faza 1...");
            Thread.sleep((long) (1000 + Math.random() * 2000));

            System.out.println(nume + " a terminat faza 1 și așteaptă ceilalți...");

            // Toți lucrătorii se opresc aici până când ajung toți
            barrier.await();

            System.out.println(nume + " începe faza 2!");
            Thread.sleep((long) (1000 + Math.random() * 2000));

            System.out.println(nume + " a terminat faza 2.");

        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

Cyclic Barrier

- CyclicBarrier barrier = new CyclicBarrier(3);
  - Asta înseamnă că **3 thread-uri** trebuie să apeleze await() ca bariera să se deschidă.

- Opțional, poți da o acțiune care se execută **doar o dată**, când ultimul thread ajunge:

  new CyclicBarrier(3, () -> System.out.println("Toți au ajuns!"));
- barrier.await() - Fiecare thread care ajunge la punctul de sincronizare execute await().

- Dacă este ultimul necesar, atunci bariera execută acțiunea (dacă există) și eliberează toate thread-urile.

- Este ideal când:
  - ai **mai multe thread-uri care fac operații în etape**;
  - fiecare thread trebuie să aștepte ca toți ceilalți să termine o fază înainte de a continua;
  - fazele sunt repetate.

- **Exemple tipice:**
  - simulări fizice (fiecare thread calculează un segment → sincronizare → trec la următorul pas);
  - jocuri multiplayer (sincronizare între runde);
  - task-uri distribuite pe mai multe nuclee.

```java
public class ExempluCyclicBarrier {
    public static void main(String[] args) {
        // Runnable-ul care se execută când toți au ajuns la barieră
        Runnable actiuneDupaBariera = () ->
            System.out.println(">>> Toți lucrătorii au terminat faza 1! Se trece la faza 2...\n");

        // CyclicBarrier pentru 3 lucrători
        CyclicBarrier barrier = new CyclicBarrier(3, actiuneDupaBariera);

        // Pornim 3 thread-uri
        new Thread(new Lucrator(barrier, "Worker 1")).start();
        new Thread(new Lucrator(barrier, "Worker 2")).start();
        new Thread(new Lucrator(barrier, "Worker 3")).start();
    }
}
```

CountDown Latch

- oarecum similar cu *CyclicBarrier* prin faptul că permite coordonarea unui grup de thread-uri. Diferența e ca atunci când un thread ajunge la barieră, nu se blochează ci doar decrementează valoarea inițiala a lacătului.
- Este util când o problemă este divizată între mai multe thread-uri, fiecare făcând o parte. Când un thread termină de rezolvat decrementează contorul.
- Mecanism de sincronizare care
  - pornește cu un **număr** (count);
  - fiecare countDown() scade numărul;
  - când numărul devine **0**, se deblochează toate thread-urile aflate în await().

```java
import java.util.concurrent.CountDownLatch;

class Worker implements Runnable {
    private final CountDownLatch latch;
    private final String name;

    public Worker(CountDownLatch latch, String name) {
        this.latch = latch;
        this.name = name;
    }

    @Override
    public void run() {
        try {
            System.out.println(name + " începe lucrul...");
            Thread.sleep((long) (1000 + Math.random() * 2000));
            System.out.println(name + " a terminat!");

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // Semnalizează că un task a fost terminat
            latch.countDown();
        }
    }
}
```

CountDown Latch

- Exemple de utilizare
  - Serverul așteaptă X servicii să pornească.
  - Teste JUnit care pornesc mai multe fire și așteaptă să se termine.
  - Aplicația principală așteaptă încărcarea configurărilor.

Caracteristică	CountDownLatch	CyclicBarrier
Reutilizabilitate	❌ NU	✔ DA
Cine așteaptă?	Doar cei care fac `await()`	Toți participanții
Scop	Așteptarea terminării unor task-uri	Sincronizarea pe faze
Resetare	Nu poate fi resetat	Se resetează automat

```java
public class ExempluCountDownLatch {
    public static void main(String[] args) {
        // Latch cu 3 pași de contorizat
        CountDownLatch latch = new CountDownLatch(3);

        // Pornim 3 workeri
        new Thread(new Worker(latch, "Worker 1")).start();
        new Thread(new Worker(latch, "Worker 2")).start();
        new Thread(new Worker(latch, "Worker 3")).start();

        try {
            System.out.println("Main thread așteaptă finalizarea workerilor...");
            latch.await();  // Main blochează aici până când latch ajunge la 0
            System.out.println("Toți workerii au terminat! Pornim aplicația...");

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

MHP
A PORSCHE COMPANY

Colecții Thread-Safe

Colecții Thread-Safe

- **Colecțiile thread-safe** sunt concepute astfel încât:
  - să prevină coruperea datelor când sunt accesate din mai multe thread-uri;
  - să asigure (într-o anumită măsură) vizibilitatea corectă a modificărilor între thread-uri.
- Colecții vechi sincronizate: Vector, Hashtable (folosesc synchronized pe metode) -> nu se mai recomandă în cod nou
- Collections.synchronizedX- wrapping sincronizat
  - Ai colecții normale (ex: ArrayList, HashMap) și le "îmbraci" într-o versiune sincronizată:

```
List<String> list = new ArrayList<>();
List<String> syncList = Collections.synchronizedList(list);

Map<String, Integer> map = new HashMap<>();
Map<String, Integer> syncMap = Collections.synchronizedMap(map);
```

  - Toate metodele de bază (add, get, remove, etc.) sunt **sincronizate**.
  - TOTUȘI: la iterație, trebuie tu să sincronizezi manual, altfel riști ConcurrentModificationException.

```
synchronized (syncList) {
    for (String s : syncList) {
        System.out.println(s);
    }
}
```

Colecții moderne din java.util.concurrent

MHP
A PORSCHE COMPANY

ConcurrentHashMap<K, V>

- Un Map thread-safe
- Permite acces concurent pentru citire și scriere
- Operații specializate (putIfAbsent, computeIfAbsent, etc.) care sunt **atomice**.

```
ConcurrentHashMap<String, Integer> counts = new ConcurrentHashMap<>();

// incrementare sigură în prezența mai multor thread-uri
counts.merge("apple", 1, Integer::sum);
```

```
counts.putIfAbsent("key", 0);      // adaugă doar dacă nu există
```

CopyOnWriteArrayList și CopyOnWriteArraySet

- Idee: la fiecare modificare (add, remove), colecția:
  - face o **copie** a array-ului intern,
  - modifică copia,
  - înlocuiește referința.

Citirea e **lock-free** și foarte rapidă (doar citește un array imutabil).
Modificarea e costisitoare (pentru că se copiază tot array-ul).

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("A");
list.add("B");

for (String s : list) {  // iterare sigură, fără ConcurrentModificationException
    System.out.println(s);
}
```

Colecții moderne din java.util.concurrent

MHP
A PORSCHE COMPANY

ConcurrentLinkedQueue, LinkedBlockingQueue

- ConcurrentLinkedQueue – lock-free, potrivită pentru multe operații concurente.
- LinkedBlockingQueue / ArrayBlockingQueue – cozi **blocking** (thread-ul poate aștepta până când există elemente sau loc liber).

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();

// Producer
new Thread(() -> {
    try {
        queue.put("mesaj");
    } catch (InterruptedException e) { }
}).start();

// Consumer
new Thread(() -> {
    try {
        String msg = queue.take();  // așteaptă până apare un element
        System.out.println(msg);
    } catch (InterruptedException e) { }
}).start();
```

Colecții moderne din java.util.concurrent

MHP
A PORSCHE COMPANY

ConcurrentSkipListMap și ConcurrentSkipListSet

- Variante thread-safe și ordonate de Map și Set,
- Bazate pe skip-list (permit operații logaritmice),
- Folositoare când ai nevoie de colecții **ordonate** + acces concurent.

Cazuri de utilizare a colecțiilor

- Ai **multă scriere + multă citire** pe un map → ConcurrentHashMap
- Ai **multă citire, puțină scriere** pe o listă → CopyOnWriteArrayList
- Ai nevoie de **coadă între thread-uri** (pattern producer-consumer) → LinkedBlockingQueue sau alt BlockingQueue
- Ai nevoie de **map ordonat (sorted) + concurent** →ConcurrentSkipListMap
- Ai deja un ArrayList, puține thread-uri, nu e critică performanța → Collections.synchronizedList(list)

MHP
A PORSCHE COMPANY

Parallel Streams

Parallel Streams

- Un **parallel stream** folosește *ForkJoinPool.commonPool()* pentru a împărți automat elementele colecției în task-uri care rulează **în paralel** pe mai multe thread-uri.

```
list.stream()          // procesează secvențial (un singur thread)
list.parallelStream()  // procesează concurent (mai multe thread-uri)
```

- Cum funcționează?
  - ForkJoinPool.commonPool()
  - colecția e împărțită în bucăți
  - fiecare bucată e procesată pe un worker thread
  - rezultatele sunt combinate la final (reduce)

```
long count = list.parallelStream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .reduce(0, Integer::sum);
```

Parallel Stream

- **Parallel Streams NU sunt bune când accesezi resurse shared (**modifici colecții globale, scrii în fișiere la care accesează alte thread-uri, faci apeluri către API-uri non-thread-safe, etc).

```
List<Integer> output = new ArrayList<>();

list.parallelStream().forEach(output::add);  // RACE CONDITION
```


```
List<Integer> output = list.parallelStream()
                           .map(x -> x * 2)
                           .collect(Collectors.toList());
```

Parallel Stream

- Când parallel streams sunt mai lente decât sequential streams?
  - **Lista mică (cost de overhead > beneficiu) -** Pentru colecții < 10 000 elemente → overhead mare.
  - **Operații rapide-** Dacă operația durează < 100ns → parallel devine mai lent.
  - folosești colecții non-thread-safe extern

- Folosește parallel streams dacă:
  - colecția este mare (zeci de mii sau milioane)
  - operația este **CPU-bound**, cost per element mare
  - ordinea nu contează (sau folosești forEachOrdered)

```
rawData.getOffers().parallelStream()
    .forEach(
        capture of extends IPlanningOffer offer -> {
          PlanningDetailsOptionRowDto row =
              context.getOptionBasicIdToOptionMap().get(offer.getOptionBasic().getId());
          if (row != null
              && planningOptionBasicsIds.contains(row.getOption().getOptionBasic().getId())
              && context.getDerivateIdToColumnIndexMap().get(offer.getDerivative().getId())
                  != null) {
            PlanningDetailsCellDto cell =
                row.getCells()
                    .get(
                        context
                            .getDerivateIdToColumnIndexMap()
                            .get(offer.getDerivative().getId()));
            if (cell != null && planningDerivativesIds.contains(cell.getDerivative().getId())) {
              cell.getOffers().put(offer.getMarketUID().getMarketUID(), offer);
            }
          }
        });
```

Q & A