IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

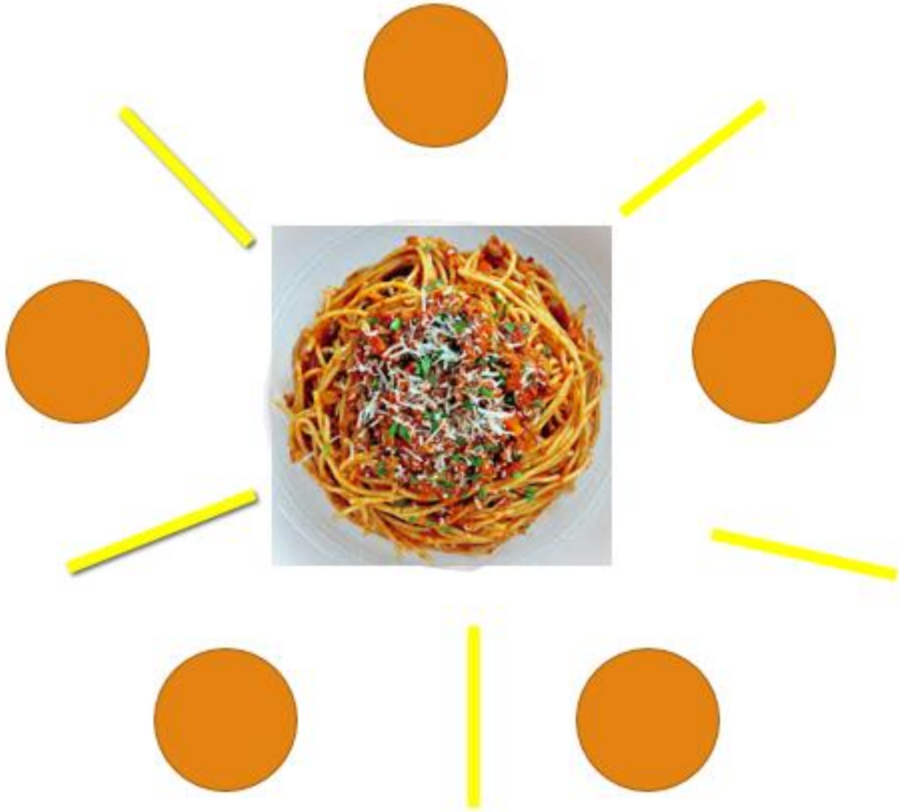
Ioana Leustean

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)

- The Dining Philosophers



http://rosettacode.org/wiki/Dining_philosophers

"In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. The names of the philosophers were PHIL0, PHIL1, PHIL2, PHIL3, PHIL4, and they were disposed in this order anticlockwise around the table. To the left of each philosopher there was laid a golden fork, and in the center stood a large bowl of spaghetti, which was constantly replenished.

A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his own chair, picked up his own fork on his left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. The philosopher therefore had also to pick up the fork on his right. When we was finished he would put down both his forks, get up from his chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If the other philosopher wants it, he just has to wait until the fork is available again."

C.A.R. Hoare, Communicating Sequential Processes, 2004
(formulate initial de E. Dijkstra

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

➢Observatii

- Excludere mutuala - doi filozofi diferiti nu pot folosi aceeasi furculita simultan
- Coada circulara – actiunile unui filozof sunt conditionate de actiunile vecinilor

➢Probleme

Deadlock

Fiecare filozof are o furculita si asteapta ca ceilalti vecini sa elibereze o furculita

Starvation

Un filozof nu mananca niciodata
(ex: unul din vecini nu elibereaza furculita)

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() ;}
}
```

- Dining Philosophers

Fiecare filozof executa
la infinit urmatorul ciclu

n = numarul de filozofi

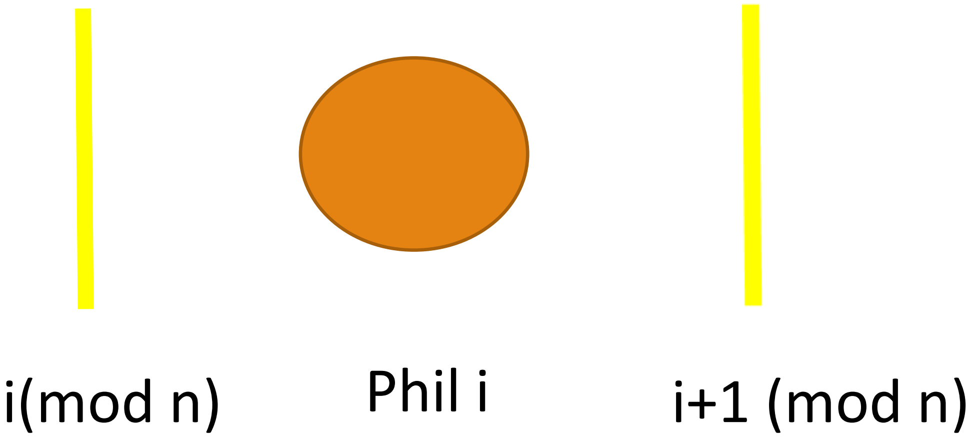
asteapta sa manance

ia furculitele

mananca

elibereaza furculitele

mediteaza



```
public class DiningPhilosophers {

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

  Chopstick[] chopsticks = new Chopstick[5];     //  pentru crearea betelor

  Philosopher[] philosophers = new Philosopher[5]; // crearea thread-urilor filozof
                                                      parametrizate de bete

   for (int i = 0; i < 5; ++i)    chopsticks[i] = new Chopstick(i);

   for (int i = 0; i < 5; ++i) {
     philosophers[i] = new Philosopher("Phil"+i,, chopsticks[i], chopsticks[(i + 1) % 5]);
     philosophers[i].start();
   }
   for (int i = 0; i < 5; ++i)
     philosophers[i].join();
 }}
```

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

```
public class DiningPhilosophers {

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

    Philosopher[] philosophers = new Philosopher[5];
    Chopstick[] chopsticks = new Chopstick[5];

    for (int i = 0; i < 5; ++i)  chopsticks[i] = new Chopstick(i);

    for (int i = 0; i < 5; ++i) {
            philosophers[i] = new Philosopher("Phil"+i,, chopsticks[i], chopsticks[(i + 1) % 5]);
            philosophers[i].start();
    }
    for (int i = 0; i < 5; ++i)
         philosophers[i].join();
  }
}
```

```
class Chopstick {
 private int id;
 public Chopstick(int id) { this.id = id; }
 public int getId() { return id; }
}
```

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

```
class Philosopher extends Thread {

  private String name;
  private Chopstick first, second;

 public Philosopher(String name, Chopstick left, Chopstick right) {
   this.name=name;
   this.first=… ; this.second=… // ia furculitele }

public void run() {
while(true) {
// vrea sa manance
//mananca cand  poate
//gandeste
}}
}
```

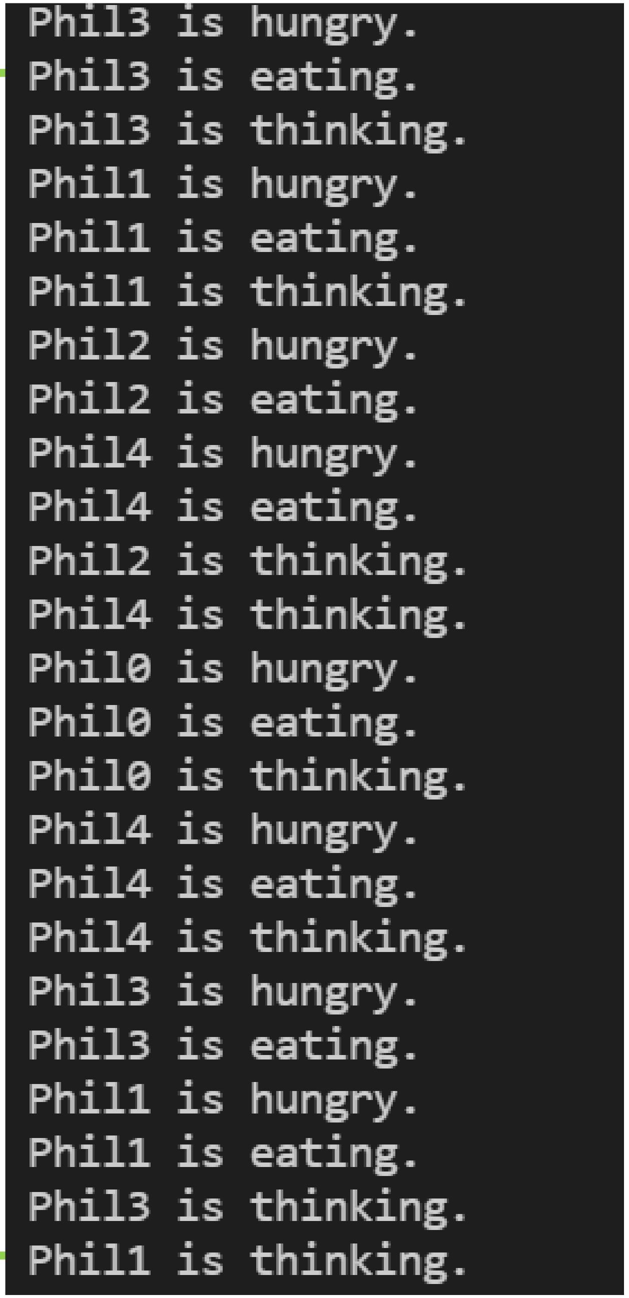
```
public void run() {
   try {
     while(true) {
       System.out.println(name + " is hungry.");   // vrea sa manance

       synchronized(first) {
         synchronized(second) {
           System.out.println(name + " is eating.");
            Thread.sleep(ThreadLocalRandom.current().nextInt(1000)); //  mananca }
       }}

     System.out.println(name + " is thinking.");
     Thread.sleep(ThreadLocalRandom.current().nextInt(10000));  // gandeste
     }
   } catch(InterruptedException e) {}
 }
```

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

```
class Philosopher extends Thread {
   private String name;   private Chopstick first, second;

 public Philosopher(String name, Chopstick left, Chopstick right) {
  this.name=name;
  this.first= left; this.second= right; // ia furculitele }

public void run() {
  try {
   while(true) {
    System.out.println(name + " is hungry.");   // vrea sa manance
    synchronized(first) {
     synchronized(second) {
      System.out.println(name + " is eating.");
       Thread.sleep(ThreadLocalRandom.current().nextInt(1000)); //  mananca }
    }}
   System.out.println(name + " is thinking.");
   Thread.sleep(ThreadLocalRandom.current().nextInt(10000));  // gandeste
   }  } catch(InterruptedException e) {}
 }}
```

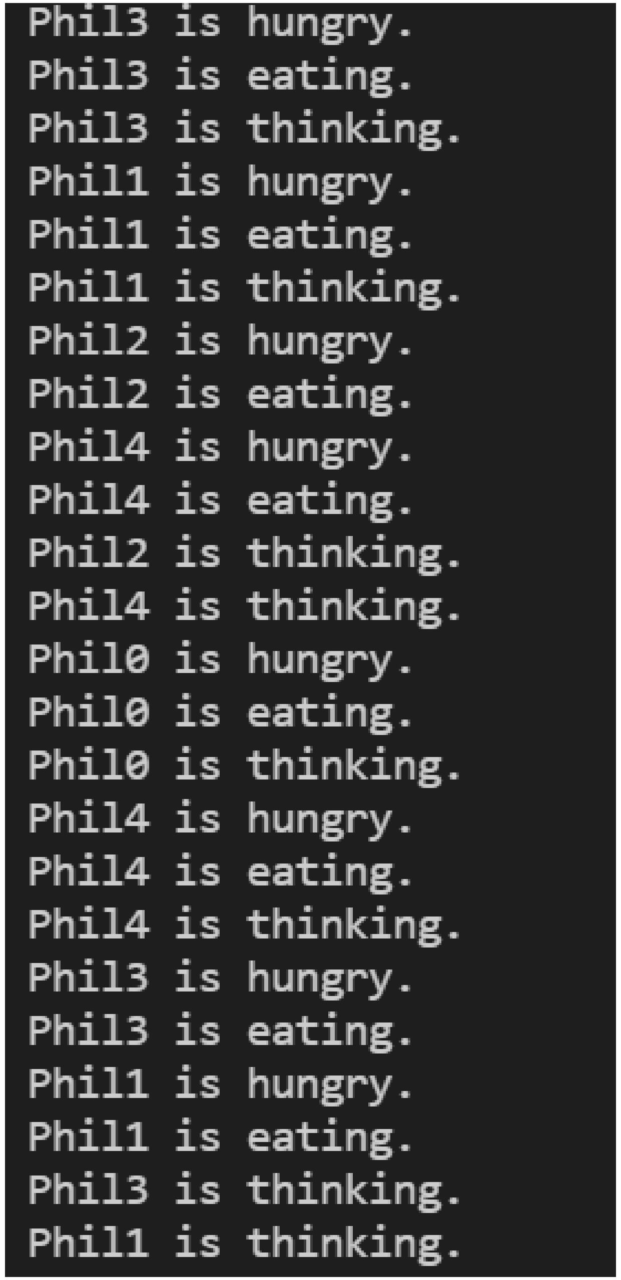
https://docs.oracle.com/javase/tutorial/essential/concurrency

```
class Philosopher extends Thread {
   private String name;   private Chopstick first, second;

 public Philosopher(String name, Chopstick left, Chopstick right) {
  this.name=name;
  this.first= left; this.second= right; // ia furculitele }

public void run() {
   try {
     while(true) {
      System.out.println(name + " is hungry.");   // vrea sa manance
      synchronized(first) {
       synchronized(second) {
        System.out.println(name + " is eating.");
         Thread.sleep(ThreadLocalRandom.current().nextInt(1000)); //  mananca }
      }}
     System.out.println(name + " is thinking.");
     Thread.sleep(ThreadLocalRandom.current().nextInt(10000));  // gandeste
     }  } catch(InterruptedException e) {}
 }}
```



```
Phil3 is hungry.
Phil3 is eating.
Phil3 is thinking.
Phil1 is hungry.
Phil1 is eating.
Phil1 is thinking.
Phil2 is hungry.
Phil2 is eating.
Phil4 is hungry.
Phil4 is eating.
Phil2 is thinking.
Phil4 is thinking.
Phil0 is hungry.
Phil0 is eating.
Phil0 is thinking.
Phil4 is hungry.
Phil4 is eating.
Phil4 is thinking.
Phil3 is hungry.
Phil3 is eating.
Phil1 is hungry.
Phil1 is eating.
Phil3 is thinking.
Phil1 is thinking.
```

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



```
Phil3 is hungry.
Phil3 is eating.
Phil3 is thinking.
Phil1 is hungry.
Phil1 is eating.
Phil1 is thinking.
Phil2 is hungry.
Phil2 is eating.
Phil4 is hungry.
Phil4 is eating.
Phil2 is thinking.
Phil4 is thinking.
Phil0 is hungry.
Phil0 is eating.
Phil0 is thinking.
Phil4 is hungry.
Phil4 is eating.
Phil4 is thinking.
Phil3 is hungry.
Phil3 is eating.
Phil1 is hungry.
Phil1 is eating.
Phil3 is thinking.
Phil1 is thinking.
```

"[…] I set five of these going simultaneously, they typically run very happily for hours on end (my record is over a week). Then, all of a sudden, everything grinds on a halt."

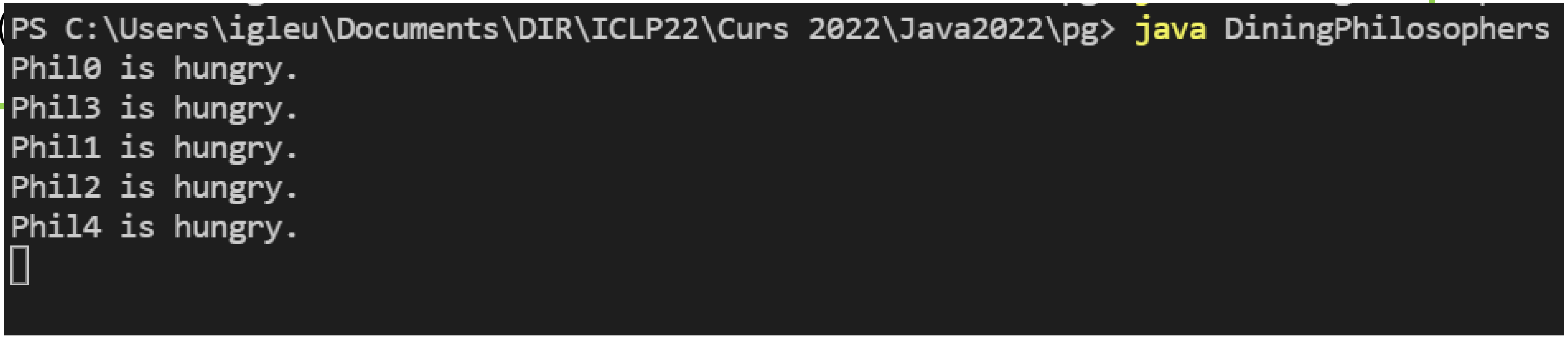
P. Butcher, Seven Concurrency Models in Seven Weeks

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

```
public void run() {
   try {
     while(true) {
       System.out.println(name + " is hungry.");   // vrea sa manance
       synchronized(first) {
         Thread.sleep(ThreadLocalRandom.current().nextInt(10));
         synchronized(second) {
          System.out.println(name + " is eating.");
           Thread.sleep(ThreadLocalRandom.current().nextInt(1000)); //  mananca }
       }}
     System.out.println(name + " is thinking.");
     Thread.sleep(ThreadLocalRandom.current().nextInt(10000));  // gandeste
     }
   } catch(InterruptedException e) {}
 }
```

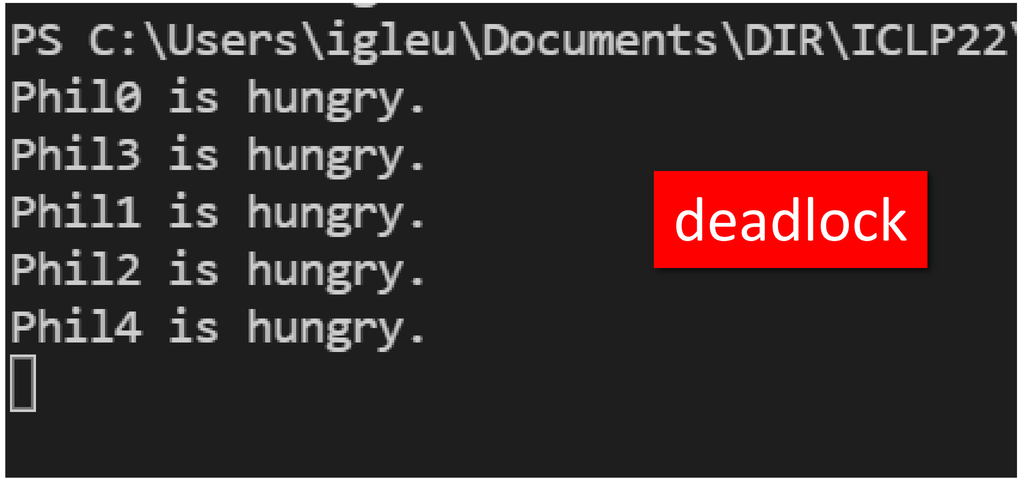
https://docs.oracle.com/javase/tutorial/essential/concurrency

```
public void run() {
  try {
    while(true) {
      System.out.println(name + " is hungry.");   // vrea sa manance
      synchronized(first) {
        Thread.sleep(ThreadLocalRandom.current().nextInt(10));
        synchronized(second) {
         System.out.println(name + " is eating.");
          Thread.sleep(ThreadLocalRandom.current().nextInt(1000)); //  mananca }
      }}
    System.out.println(name + " is thinking.");
    Thread.sleep(ThreadLocalRandom.current().nextInt(10000));  // gandeste
    }
  } catch(
 }
```



```
PS C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg> java DiningPhilosophers
Phil0 is hungry.
Phil3 is hungry.
Phil1 is hungry.
Phil2 is hungry.
Phil4 is hungry.
```

```
public Philosopher(String name, Chopstick left,
Chopstick right) {
  this.name=name;
  this.first= left; this.second= right; // ia furculitele }

public void run() {
    …
   synchronized(first) {
       synchronized(second) {
        …  }}}
```

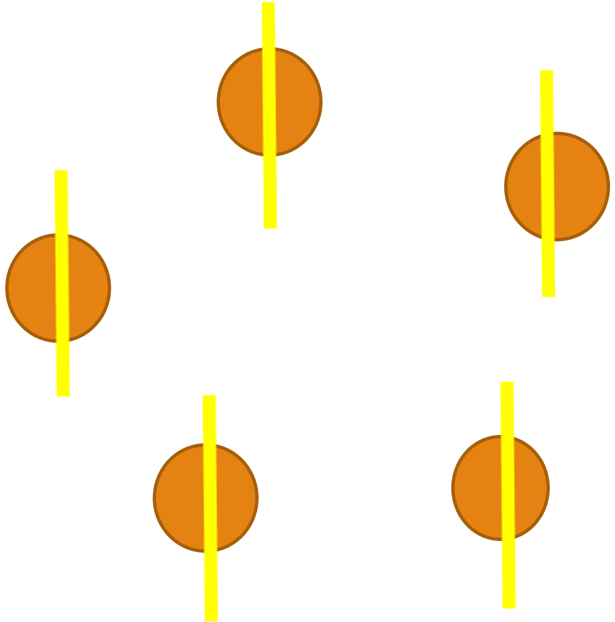


```
PS C:\Users\igleu\Documents\DIR\ICLP22
Phil0 is hungry.
Phil3 is hungry.
Phil1 is hungry.
Phil2 is hungry.
Phil4 is hungry.
```

deadlock

"[…] I set five of these going simultaneously, they typically run very happily for hours on end (my record is over a week). Then, all of a sudden, everything grinds on a halt."

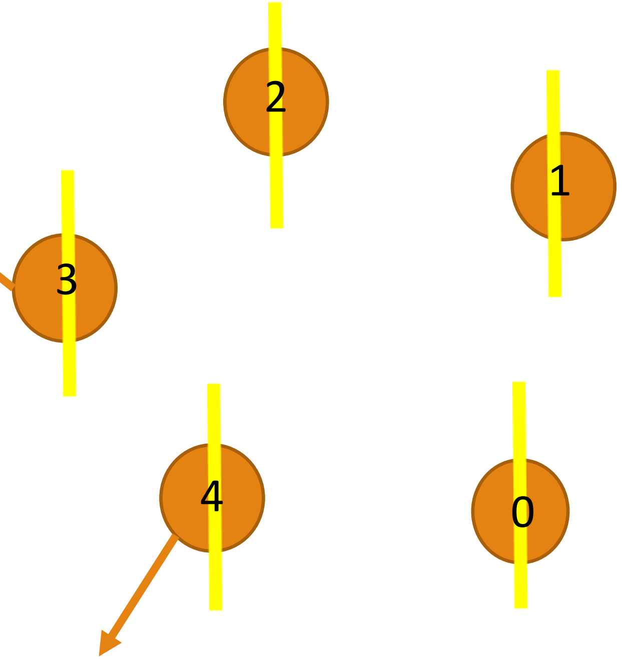
P. Butcher, Seven Concurrency Models in Seven Weeks



- este posibil ca toti sa ia furculita stanga simultan
- raman blocati asteptand sa ia furculita din dreapta

SOLUTIA (Dijskstra)

- **ordine globala** pe lacate (furculite)
- lacatele (furculitele) sunt luate **in ordine**:
  - intai cea mai mica (in ordinea globala)
  - apoi cea mai mare (in ordinea globala)



poate lua furculita 4 si poate manca

trebuie sa astepte pana cand 0 este libera!

```
class Philosopher extends Thread {
  private String name;
  private Chopstick first, second;

   public Philosopher(String name, Chopstick left, Chopstick right) {
   this.name=name;
   if(left.getId() < right.getId()) {
      first = left; second = right;
    } else {
      first = right; second = left;
    }
   }
```

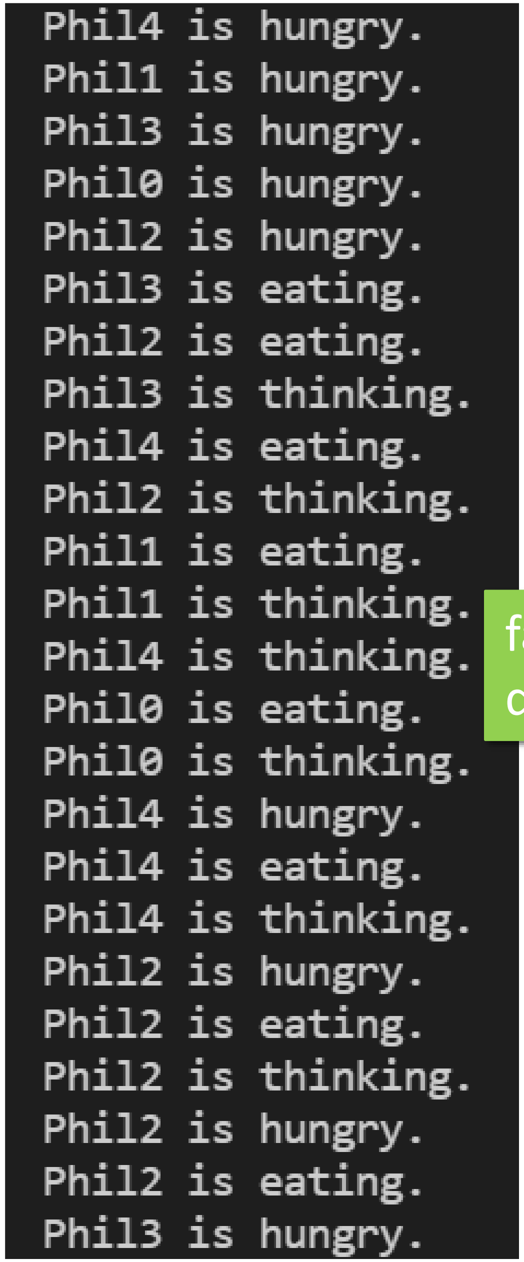
- ordine globala pe lacate (furculite)
- lacatele (furculitele) sunt luate in ordine :
    - intai cea mai mica (in ordinea globala)
    - apoi cea mai mare (in ordinea globala)

```
public void run() {
      ...
    synchronized(first ) {
   // Thread.sleep(ThreadLocalRandom.current().nextInt(10));
          synchronized(second) {
            ...
        } ... }}
```

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

```
class Philosopher extends Thread {
  private String name;
  private Chopstick first, second;

  public Philosopher(String name, Chopstick left, Chopstick right) {
   this.name=name;
   if(left.getId() < right.getId()) {
      first = left; second = right;
    } else {
      first = right; second = left;
    }
   }
```

```
public void run() {
      …
    synchronized(first ) {
      Thread.sleep(ThreadLocalRandom.current().nextInt(10));
         synchronized(second) {
           …
       } … }}
```



```
Phil4 is hungry.
Phil1 is hungry.
Phil3 is hungry.
Phil0 is hungry.
Phil2 is hungry.
Phil3 is eating.
Phil2 is eating.
Phil3 is thinking.
Phil4 is eating.
Phil2 is thinking.
Phil1 is eating.
Phil1 is thinking.
Phil4 is thinking.
Phil0 is eating.
Phil0 is thinking.
Phil4 is hungry.
Phil4 is eating.
Phil4 is thinking.
Phil2 is hungry.
Phil2 is eating.
Phil2 is thinking.
Phil2 is hungry.
Phil2 is eating.
Phil3 is hungry.
```

fara deadlock

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

Interfata Lock

interface Lock

class ReentrantLock

```
import java.util.concurrent.locks.*

 Lock obLock = new ReentrantLock();
    obLock.lock();
    try {
           //  acceseaza resursa protejata de obLock
} finally {
              obLock.unlock();
           }
```

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/concurrent/locks/Lock.html

➢ Interface Condition

- conditiile sunt legate de un obiect Lock

```
Lock obiectLock = new ReentrantLock();
Condition cond_obiectLock = obiectLock.newCondition();
```

- pot exista mai multe conditii pentru acelasi obiect Lock.

- implementeaza metode asemanatoare cu **wait()**, **notify()** si **notifyall()** pentru obiectele din clasa Lock

    ○ **await()**, **cond.await(long time, TimeUnit unit)**
    thread-ul current intra in asteptare

    ○ **signall()**
    un singur thread care asteapta este trezit

    ○ **signalAll()**
    toate thread-urile care asteapta sunt trezite

[Condition (Java SE 23 & JDK 23) (oracle.com)](https://docs.oracle.com)

- Exemplul Producator-Consumator cu Lock si Condition

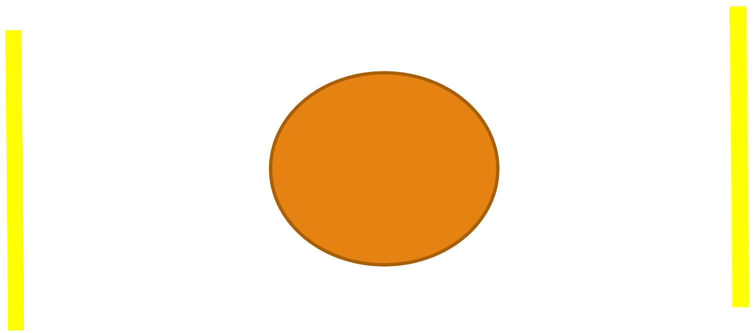
```
private boolean empty = true;
private Lock dropLock = new ReentrantLock();
private Condition cond_dropLock =   dropLock.newCondition();
```

```
public String take() {
       dropLock.lock();
       try{
       while (empty) {
           try {
               cond_dropLock.await();
           } catch (InterruptedException e) {}
       }

       empty = true;
       cond_dropLock.signalAll();
       return message;}
       finally { dropLock.unlock(); }
   }
```

```
public  void put(String message) {
    dropLock.lock();
     try{
        while (!empty) {
            try {
                cond_dropLock.await();
            } catch (InterruptedException e) {}
        }
        empty = false;
        this.message = message;
        cond_dropLock.signalAll();
      }
     finally {dropLock.unlock();}
    }
```

- Dining Philosophers

n = numarul de filozofi



Phil i

- Vom rezolva problema folosind un **ReentrantLock** folosind **cate un obiect Condition** pentru fiecare filozof

➢ Varianta folosind un **ReentrantLock** cu un obiect **Condition** pentru fiecare filozof

- Furculitele nu sunt definite explicit

- Actiunile unui filozof sunt
    - mananca
    - gandeste

- Un filozof poate manca numai cand filozofii vecini gandesc

- **ReentrantLock table** este un **lacat comun**

- Fiecare filozof are un obiect **Condition** propriu asociat lacatului comun

- Fiecare filozof are o variabila booleana proprie eating care descrie starea filozofului: manaca sau gandeste

```
public Philosopher(String name, ReentrantLock table) {
        this. name = name;
        this.table = table;
        condition = table.newCondition();
        eating = false;   }
```

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

```
public class DiningPhilosophers {

  public static void main(String[] args) throws InterruptedException {
    Philosopher[] philosophers = new Philosopher[5];
    ReentrantLock table = new ReentrantLock();

    for (int i = 0; i < 5; ++i)
      philosophers[i] = new Philosopher("Phil"+i,table);

    for (int i = 0; i < 5; ++i) {
      philosophers[i].setLeft(philosophers[(i + 4) % 5]);
      philosophers[i].setRight(philosophers[(i + 1) % 5]);
      philosophers[i].start();
    }
    for (int i = 0; i < 5; ++i)
              philosophers[i].join();
  }}
```

Fiecare filozof trebuie sa acceseze starea filozofilor vecini pentru a sti daca acestia mananca sau gandesc.

```java
class Philosopher extends Thread {
  private String name;    private boolean eating;
  private Philosopher left;    private Philosopher right;
  private ReentrantLock table;  private Condition condition;

   public Philosopher(String name, ReentrantLock table) {
      this. name = name;
      this.table = table;

      condition = table.newCondition();
      eating = false;
   }

  public void setLeft(Philosopher left) { this.left = left; }
  public void setRight(Philosopher right) { this.right = right; }

public void run(){…}
}
```

```java
public void run() {
   try {

      while (true) {
        think();
        eat();
      }
   } catch (InterruptedException e) {}
 }
```

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

```
private void eat() throws InterruptedException {
  table.lock();

  try {
    while (left.eating || right.eating) { condition.await();}
    eating = true;
  } finally { table.unlock(); }

  System.out.println( name + " is eating");
  Thread.sleep(ThreadLocalRandom.current().nextInt(1000));
}
```

Un thread filozof trebuie sa detina lacatul pentru a incepe sa manance si pentru aceasta asteapta pana cand ambii vecini au terminat de mancat.

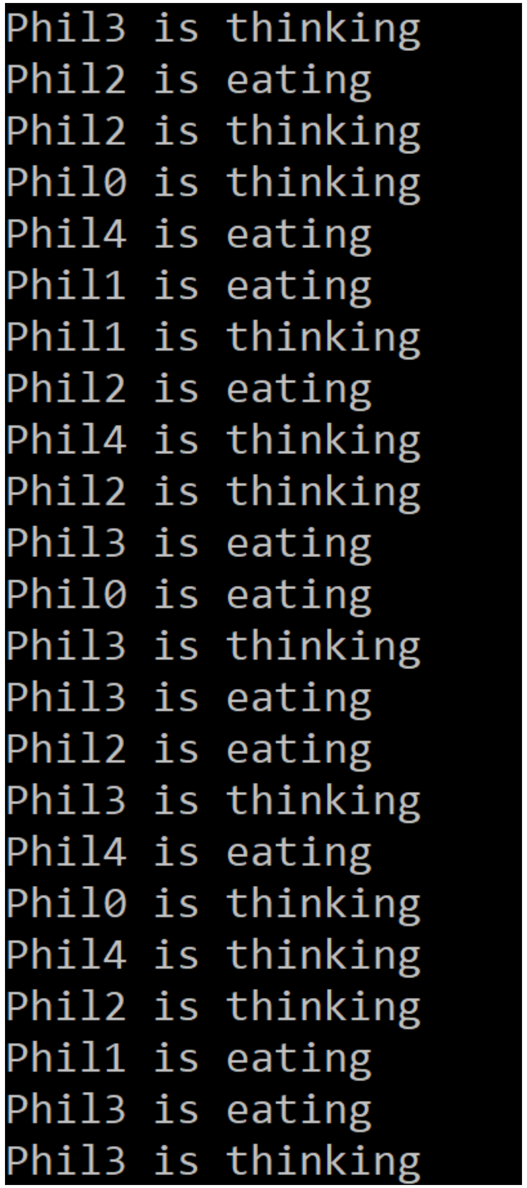
await() elibereaza lacatul

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

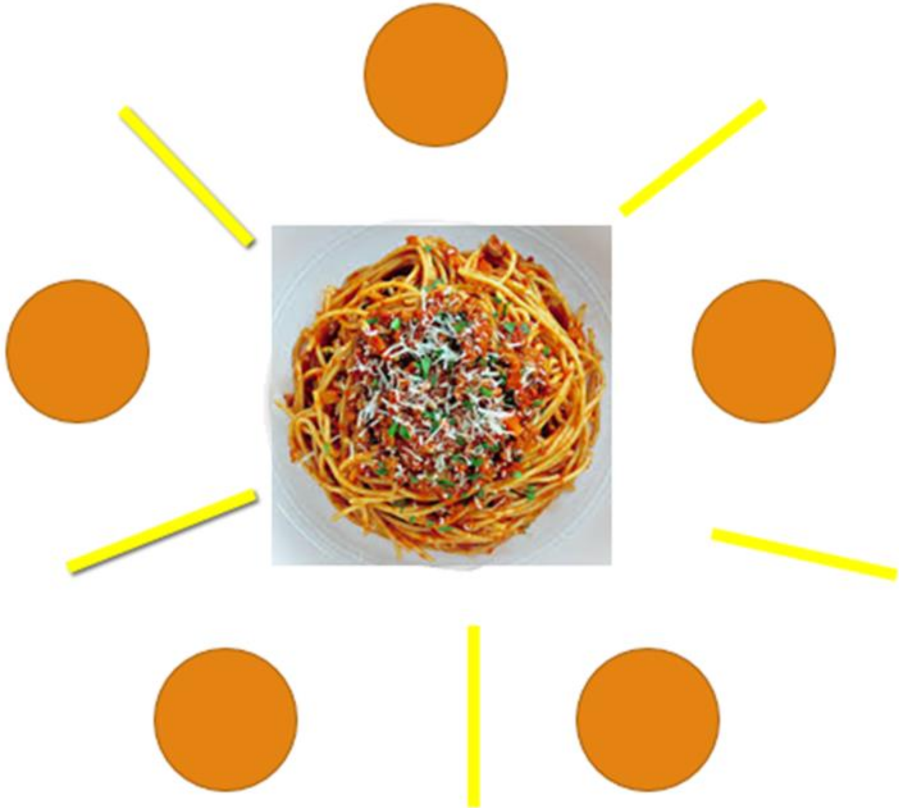
```
private void think() throws InterruptedException {

  table.lock();

  try {
    eating = false;
    left.condition.signal();
    right.condition.signal();
  } finally { table.unlock(); }

    System.out.println( name + " is thinking");
    Thread.sleep(ThreadLocalRandom.current().nextInt(1000));
 }
```

Cand termina de mancat semnalizeaza vecinilor ca pot incerca sa ia lacatul comun pentru a manca.

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



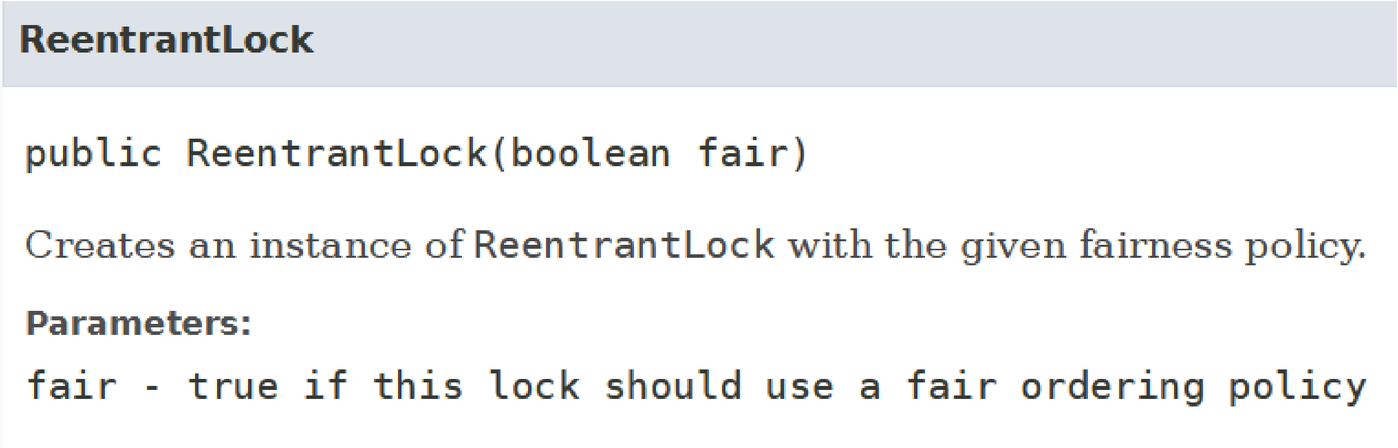
```
Phil3 is thinking
Phil2 is eating
Phil2 is thinking
Phil0 is thinking
Phil4 is eating
Phil1 is eating
Phil1 is thinking
Phil2 is eating
Phil4 is thinking
Phil2 is thinking
Phil3 is eating
Phil0 is eating
Phil3 is thinking
Phil3 is eating
Phil2 is eating
Phil3 is thinking
Phil4 is eating
Phil0 is thinking
Phil4 is thinking
Phil2 is thinking
Phil1 is eating
Phil3 is eating
Phil3 is thinking
```



➢ Interfata Lock

interface Lock
class ReentrantLock

"The constructor for this class accepts an optional fairness parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. Note however, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock."



ReentrantLock

```
public ReentrantLock(boolean fair)
```

Creates an instance of `ReentrantLock` with the given fairness policy.

Parameters:

`fair - true if this lock should use a fair ordering policy`

[ReentrantLock (Java SE 23 & JDK 23) (oracle.com)](https://docs.oracle.com)

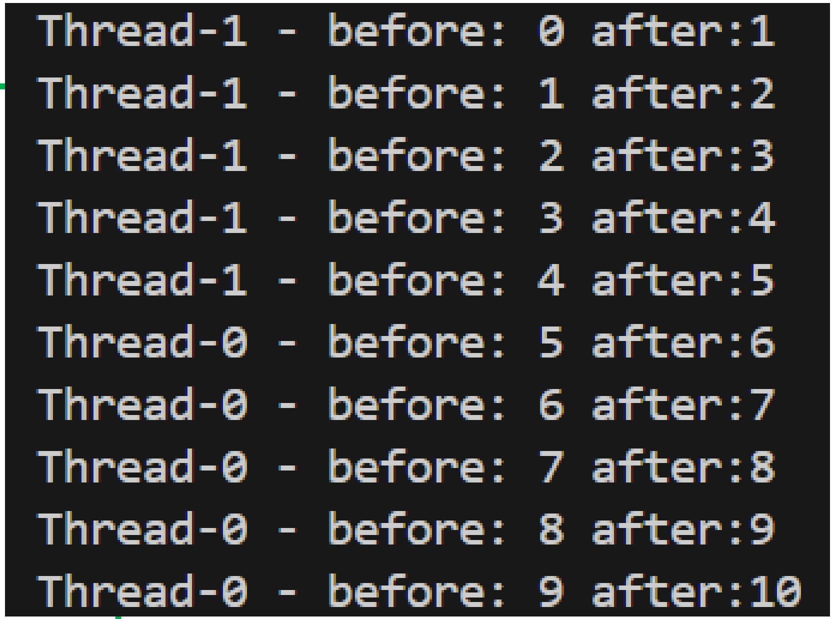
- Un contor incrementat de doua threaduri care il acceseaza repetat

```
public class Interferencelockfair {

    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 (Counter counter) {this.counter=counter;}

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

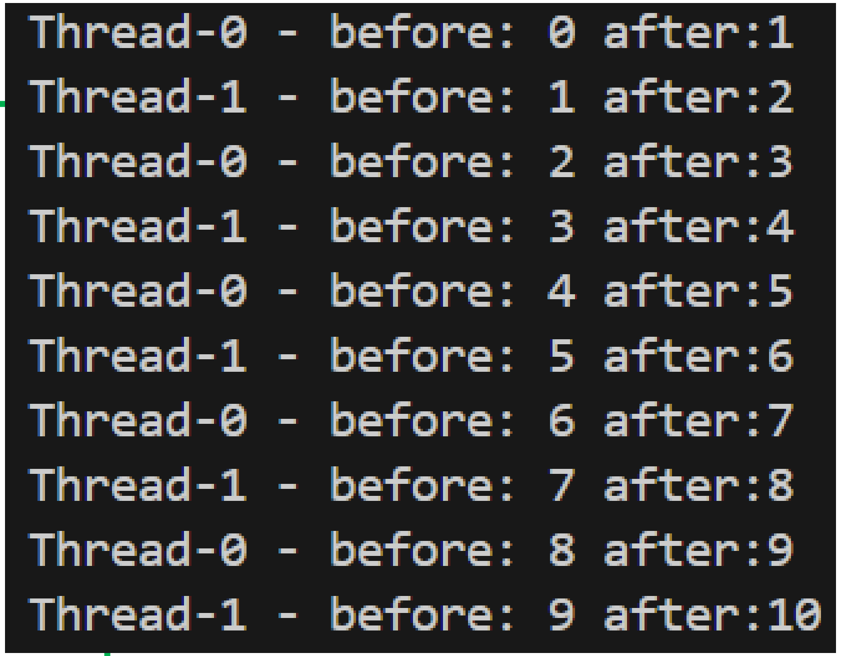
https://docs.oracle.com/javase/tutorial/essential/concurrency

```
class Counter{
 private int counter = 0;
 private Lock clock = new ReentrantLock(false);
 public  void performTask () {
      clock.lock();
       try {
        int temp = counter;
        counter++;
        System.out.println(Thread.currentThread()
                                .getName() + " - before: "+temp+" after:" + counter);
   }
   finally{clock.unlock();}
   }
 }
```



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

```
class Counter{
 private int counter = 0;
 private Lock clock = new ReentrantLock(true);
 public  void performTask () {
     clock.lock();
      try {
       int temp = counter;
       counter++;
       System.out.println(Thread.currentThread()
                               .getName() + " - before: "+temp+" after:" + counter);
   }
   finally{clock.unlock();}
   }
 }
```



```
Thread-0 - before: 0 after:1
Thread-1 - before: 1 after:2
Thread-0 - before: 2 after:3
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-1 - before: 9 after:10
```

Modele de interactiuni concurente:

- ✓ Dinning Philosophers
- ✓Producer - Consumer
- ➢ Reader - Writer

➢Modelul de interactiune Cititori-Scriitori (Reader-Writers)

- Mai multe threaduri au acces la o resursa.
- Unele thread-uri scriu (writers), iar altele citesc (readers).
- Resursa poate fi accesata **simultan** de **mai multi cititori**.
- Resursa poate fi accesata de un **singur scriitor**.
- Resursa **nu** poate fi accesata simultan de cititori si de scriitori

➢ Interface ReadWriteLock

interface Lock
interface ReadWriteLock   extends Lock
class ReentrantReadWriteLock

- mentine o pereche de lacate: unul pentru citire si unul pentru scriere

- lacatul pentru citire poate fi detinut de mai multe thread-uri simultan, daca nu exista o solicitare pentru scriere

- lacatul pentru scriere poate fi detinut de un singur thread

- metoda **readLock()**   intoarce lacatul pentru cititori

- metoda **writeLock()**   intoarce lacatul pentru scriitori

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

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

public class ReaderWriter {
    private static Integer counter = 0;  // resursa

    private static ReadWriteLock lock = new ReentrantReadWriteLock();

    public static void main(String[] args) {

        (new Thread(new TaskW())).start();
        (new Thread(new TaskR())).start();
        (new Thread(new TaskR())).start();
        (new Thread(new TaskW())).start();
        (new Thread(new TaskR())).start();
        (new Thread(new TaskR())).start();
        (new Thread(new TaskR())).start();
        (new Thread(new TaskW())).start();
        }
```

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

Thread-ul **Writer**

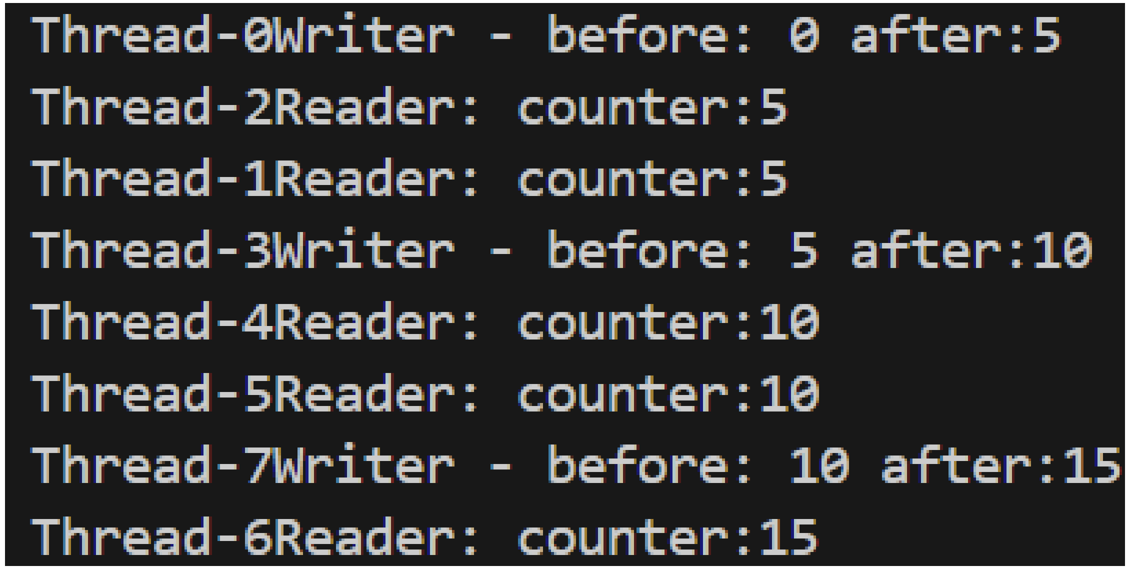
```
private static class TaskW implements Runnable {
    public void run () {

      lock.writeLock().lock();

      try{
      int temp = counter;
      for (int i=0;i<5;i++) {counter++;            // fiecare thread Writer incrementeaza contorul de 5 ori
                             Thread.currentThread().sleep(1);}
      System.out.println(Thread.currentThread().getName() +
                                                " Writer - before: "+temp+" after:" + counter);}
      catch (InterruptedException e){}
      finally {
       lock.writeLock().unlock();}
      }}
```

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

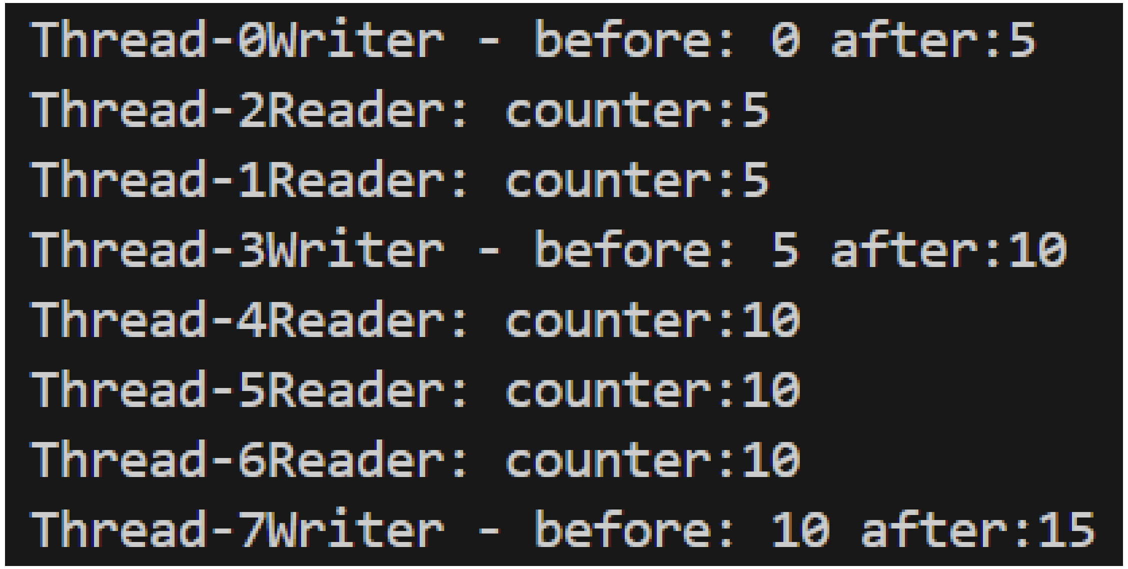
Thread-ul **Reader**

```
private static class TaskR implements Runnable {

    public void run () {

        lock.readLock().lock();

     try{
     System.out.println(Thread.currentThread().getName() + "Reader counter:" + counter);}

      finally { lock.readLock().unlock();}
 }}}
```

➢ Modelul de interactiune Cititori-Scriitori (Reader-Writers)



```
Thread-0Writer - before: 0 after:5
Thread-2Reader: counter:5
Thread-1Reader: counter:5
Thread-3Writer - before: 5 after:10
Thread-4Reader: counter:10
Thread-5Reader: counter:10
Thread-7Writer - before: 10 after:15
Thread-6Reader: counter:15
```



```
Thread-0Writer - before: 0 after:5
Thread-2Reader: counter:5
Thread-1Reader: counter:5
Thread-3Writer - before: 5 after:10
Thread-4Reader: counter:10
Thread-5Reader: counter:10
Thread-6Reader: counter:10
Thread-7Writer - before: 10 after:15
```

➢ Semafor cu cantitate (quantity semaphore)

```
public class Semaphore
extends Object

Semaphore(int permits)  // constructor
```

- implementeaza un semafor cu cantitate
  care coordoneaza accesul la un numar precizat de resurse

- metoda **aquire() / acquire(int permits)**
  thread-ul care apeleaza aquire cere accesul la o resursa;
  daca nu sunt resurse, thread-ul este blocat

- metoda **release() / release(int permits)**
  thread-ul care apeleaza release elibereaza accesul la o resursa

```
Semaphore sem = new Semaphore(n);

sem.acquire();

       ...  //sectiune critica


sem.release();
```

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

```
public class Semaphore
extends Object

Semaphore(int permits)  // constructor

Semaphore sem = new Semaphore(n);

sem.acquire();

       ...  //sectiune critica
sem.release();
```

Diferenta dintre un obiect construit cu **Semaphore(1)** si unul din clasa **Lock** este urmatoarea:

- lacatul intern al obiectului din clasa **Semaphore** este eliberat de orice thread care face **release**
- lacatul intern al obiectului din clasa **Lock** este eliberat numai de thread-ul care il detine

Varianta **Semaphore(int permits, true)** thread-urile care asteapta sa faca aquire sunt FIFO

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

Exemplu:

- un semafor coordoneaza accesul la 3 resurse
- exista 4 thread-uri care cer accesul la resursa
- dupa ce primeste accesul, fiecare thread executa 3 task-uri, apoi elibereaza resursa

```
public class Semaphores{

    static Semaphore semaphore = new Semaphore(3);

    static class MyThread extends Thread {

        // thread-ul va face aquire, va executa task-urile, apoi va face release
    }

public void main(String[] args) { …}
}
```

```
public static void main(String[] args)   {
        MyThread t1 = new MyThread("A"); t1.start();
         …..
        MyThread t4 = new MyThread("D"); t4.start();
}
```

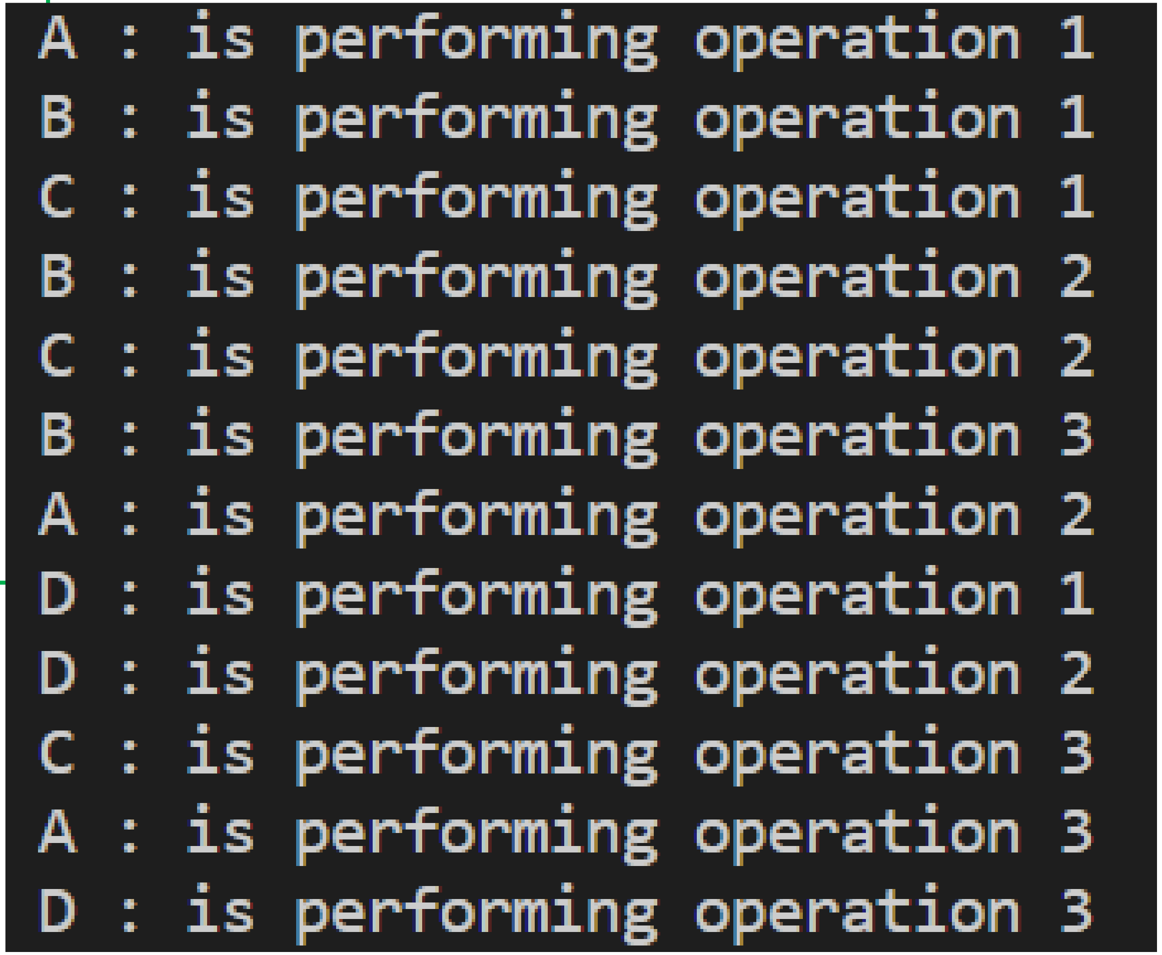
http://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/

```
static class MyThread extends Thread {
        String name = "";
        MyThread(String name) { this.name = name;}

public void run()  {
   try {
         semaphore.acquire();
                 try {

              for (int i = 1; i <= 3; i++) {
                  System.out.println(name + " : is performing operation " + i }
              Thread.sleep(ThreadLocalRandom.current().nextInt(1000));}

                   } finally { semaphore.release();}

    }  catch (InterruptedException e) {}
}}
```

aquire pune thread-urile in asteptare, deci poate arunca o exceptie

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

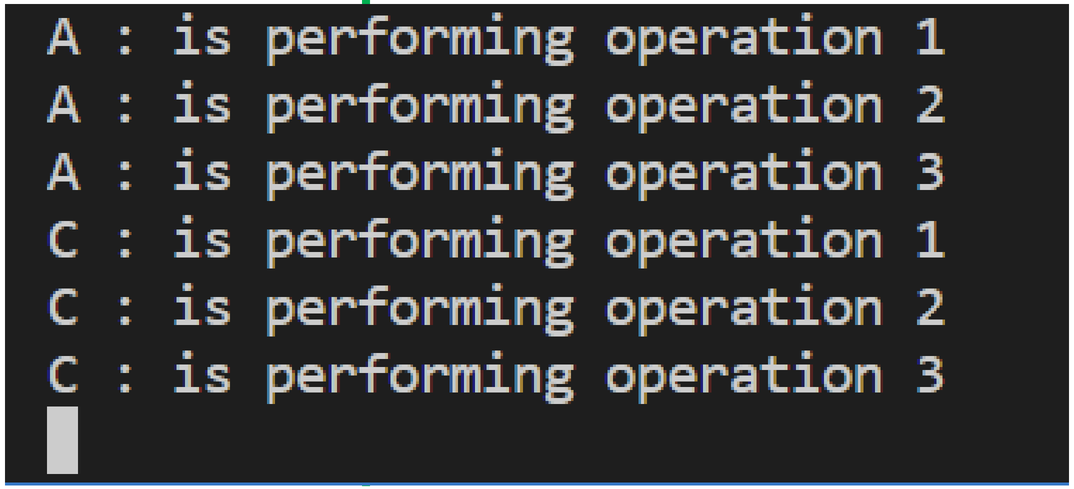
```
public class Semaphores{

  static Semaphore semaphore = new Semaphore(3);
  static class MyThread extends Thread {…}

public void main(String[] args) {
MyThread t1 = new MyThread("A"); t1.start();
MyThread t2 = new MyThread("B"); t2.start();
MyThread t3 = new MyThread("C"); t3.start();
MyThread t4 = new MyThread("D"); t4.start();
}}
```



```
A : is performing operation 1
B : is performing operation 1
C : is performing operation 1
B : is performing operation 2
C : is performing operation 2
B : is performing operation 3
A : is performing operation 2
D : is performing operation 1
D : is performing operation 2
C : is performing operation 3
A : is performing operation 3
D : is performing operation 3
```

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

```
static Semaphore semaphore = new Semaphore(3);

static class MyThread extends Thread {
        String name = "";
        MyThread(String name) { this.name = name;}

public void run()  {
   try {
           semaphore.acquire(2); // are nevoie de 2 resurse pentru executie
                   try {…
          } finally { semaphore.release(1);}
    }  catch (InterruptedException e) {}
}}
public void main(String[] args) {
MyThread t1 = new MyThread("A"); t1.start();
MyThread t2 = new MyThread("B"); t2.start();
MyThread t3 = new MyThread("C"); t3.start();
MyThread t4 = new MyThread("D"); t4.start();
}
```



```
A : is performing operation 1
A : is performing operation 2
A : is performing operation 3
C : is performing operation 1
C : is performing operation 2
C : is performing operation 3
█
```

Thread-urile B si D nu au cum sa faca acquire pe 2 resurse

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