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)

➢ 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()**

➢ Mecanismul de sincronizarea thread-urilor prin lacatul intern

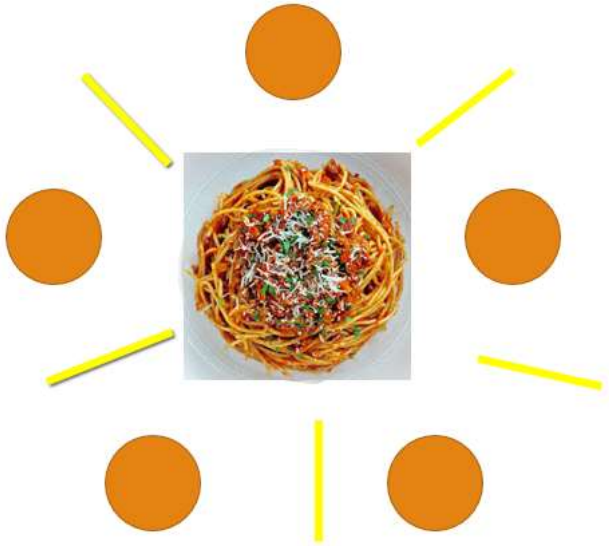
- Orice obiect din Java are asociat un lacat (monitor).
- O metoda sincronizata, atunci cand este invocata, va fi executata numai daca detine lacatul obiectului, acesta fiind eliberat automat dupa ce metoda este executata.
- Numai un singur thread poate detine lacatul obiectului la un moment dat.
- Un 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() ;}
}
```

Astfel, se evita situatia in care un thread intra in deadlock incercand sa detina un lacat pe care deja il detine.

➢ Modele de interactiune concurenta

Problema filozofilor



http://rosettacode.org/wiki/Dining_philosophers

- Problema filozofilor

Fiecare filozof executa
la infinit urmatorul ciclu

asteapta sa manance

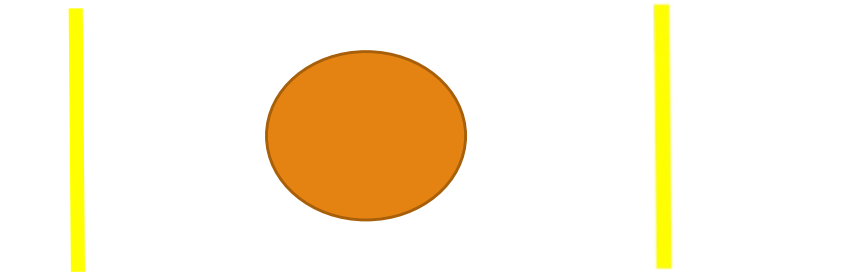
ia furculitele

mananca

elibereaza furculitele

mediteaza

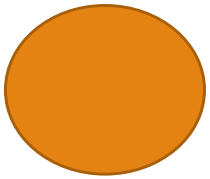
n = numarul de filozofi



```
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();
 }}
```

```
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; }
}
```



```
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) {}
 }
```

```
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) {}
  }}
```

```
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.
```

```
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

```
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) {}
 }
```

```
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(
}
```

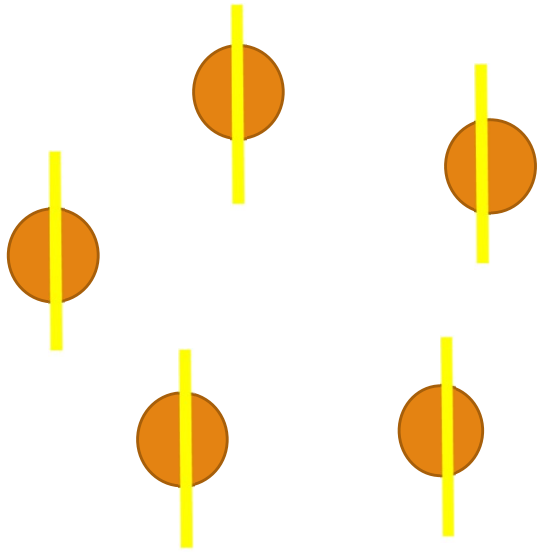
```
PS C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg> java DiningPhilosophers
Phil0 is hungry.
Phil3 is hungry.
Phil1 is hungry.
Phil2 is hungry.
Phil4 is hungry.
```

```
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) {
        … }}}
```

> *"[…] 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

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

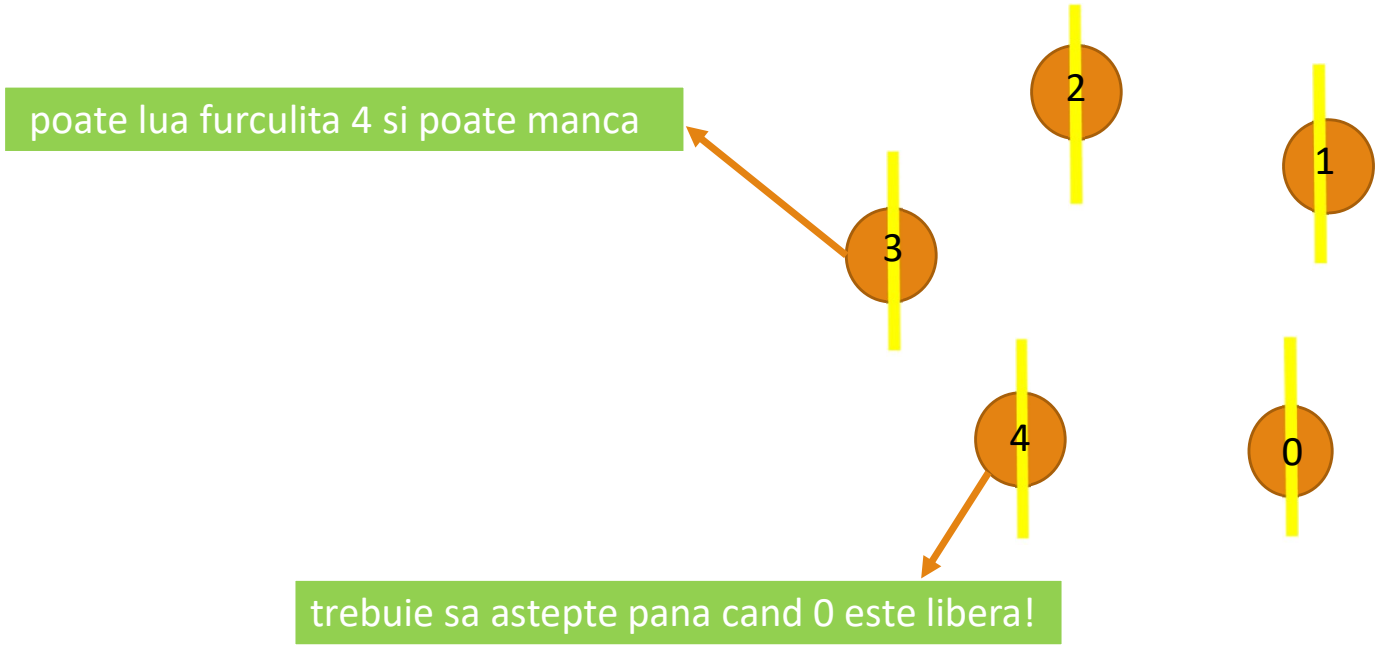
deadlock



- 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)



```
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) {
          …
       } … }}
```

```
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

➢ Modele de interactiune concurenta

Modelul Producator-Consumator

➢ **Modelul Producator-Consumator**

Doua threaduri **comunica prin intermediul unui buffer** (memorie partajata):

- thread-ul Producator creaza datele si le pune in buffer
- thread-ul Consumator ia datele din buffer si le prelucreaza

Probleme de coordonare:

- Producatorul si consumatorul nu vor accesa bufferul simultan
- Producatorul va astepta daca bufferul este plin
- Consumatorul va astepta daca bufferul este gol
- Cele doua thread-uri se vor anunta unul pe altul cand starea buferului s-a schimbat

- Modelul Producator-Consumator

Doua threaduri **comunica prin intermediul unui buffer** (memorie partajata):

- thread-ul Producator creaza datele si le pune in buffer
- thread-ul Consumator ia datele din buffer si le prelucreaza

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
   }}
```

- Modelul Producator-Consumator


```
public class PCDrop {

   private String message;
   private boolean empty = true;

   public synchronized String take() {
                …
                return message;  }

  public synchronized String put(String message) { …   }
```

implementarea buffer-ului:
accesul se face prin metode sincronizate

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

- Thread-ul **producator**

```
import java.util.Random;

class PCProducer implements Runnable {

    private PCDrop drop;
    public PCProducer(PCDrop drop) {this.drop = drop;}

   public void run() {
      String importantInfo[] = {   "m1", "m2", "m3", "m4"};
      Random random = new Random();

      for (int i = 0;  i < importantInfo.length; i++) {
                                                    drop.put(importantInfo[i]);
                                                    try {
                                                          Thread.sleep(random.nextInt(5000)
                                                       } catch (InterruptedException e) {}
                                                    }
      drop.put("DONE");   }}
```

metoda sincronizata a obiectului **drop**

- Thread-ul **consumator**

```
class Consumer implements Runnable {

  private PCDrop drop;
  public  Consumer(PCDrop drop) {   this.drop = drop;}

  public void run() {
     Random random = new Random();

     for (String message = drop.take();  ! message.equals("DONE"); message = drop.take())
                 {
                     System.out.format("MESSAGE RECEIVED: %s%n", message);
                     try {
                           Thread.sleep(random.nextInt(5000));
                          } catch (InterruptedException e) {}
     }}}
```

Metoda sincronizata a obiectului **drop**

➢ Metode ale obiectelor

Sincronizarea accesului la buffer se face folosind metodele obiectelor:

- **void wait()**
  threadul intra in asteptare pana cand primeste **notifyAll()** sau **notify()** de la alt thread

- **void wait(milisecunde)**
  threadul intra in asteptare maxim **milisecunde**

- **void notifyAll()**
  trezeste toate threadurile care asteapta lacatul obiectului

- **void notify()**
  trezeste un singur thread, ales arbitrar, care asteapta lacatul obiectului;

➢ 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
  apare in urma unui apel **ob.wait()** sau **t.join()**
- TIMED_WAITING: asteapta un alt thread, dar numai un timp limitat
  apare in urma unui apel **ob.wait(ms), t. join(ms), t.sleep(ms)**
- TERMINATED: thread-ul si-a terminat executia

➢ **Ciclul de viata al unui thread**

- exemplu – HowToDoInJava
- exemplu – javatpoint.com

Thread.State (Java SE 23 & JDK 23)

➢ ob.wait()

- Fiecare obiect, pe langa lacatul intern, are asociata si o **multime** de thread-uri in asteptare (**wait set**). Initial, aceasta multime este vida.

- Multimea thread-urilor in asteptare asociata unui obiect poate fi manipulate numai prin metodele **wait(), notify(), notifyAll().**

- Daca un thread **t** detine lacatul unui obiect **ob** de **n** ori, la apelul lui **ob.wait() ,** thread-ul **t** este adaugat in multimea thread-urilor in asteptare **M** si elibereaza de **n** ori lacatul obiectului. Thread-ul t nu mai executa instructiuni pana cand nu iese din **M**. Thread-ul t poate iesi din **M** astfel:
  - este selectat de ob.notify(),
  - la apelul ob.notifyall(),
  - la apelul t.interrupt(),
  - la expirarea timpului in cazul metodei wait(ms),
  - prin "treziri" accidentale

- Metoda **wait()** trebuie sa fie apelata numai din interiorul unei metode sincronizate sau al unui bloc sincronizat.

https://docs.oracle.com/javase/specs/jls/se23/html/jls-17.html#jls-17.2

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

➢ wait() vs sleep()

- **ob.wait()**
  - poate fi apelata de orice obiect ob
  - trebuie apelata din blocuri sincronizate
  - elibereaza lacatul intern al obiectului
  - asteapta sa primeasca o notificare prin **notify()** / **notifyAll()**
  - thread-ul current (care detine lacatul obiectului) va fi in starea WAITING iar dupa ce primeste notificare re-incearca sa detina lacatul obiectului

- **Thread.sleep()**
  - poate fi apelata oriunde
  - thread-ul curent se va opri din executie pentru perioada de timp precizata (va fi in starea BLOCKED)
  - nu elibereaza lacatele pe care le detine

Metodele **wait(), sleep() si join()** pot arunca **InterruptedException** daca un alt thread intrerupe threadul care le executa.

- Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() {
      if (empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      empty = true;
      notifyAll();
      return message;
}
public synchronized String put(String message) {..}}
```

- implementarea foloseste *blocuri cu garzi*
- thread-ul este suspendat pana cand o anume conditie este satisfacuta

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

- Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() { … return message;}

public synchronized void put(String message) {
            if (!empty) {
                        try {
                              wait();
                        } catch (InterruptedException e) {}
                        }
      empty = false;
      this.message = message;
      notifyAll(); }}
```

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

- Modelul Producator-Consumator

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
   }}
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java ProducerConsumer
Messace received: m1
Messace received: m2
Messace received: m3
Messace received: m4
```

- Modelul Producator-Consumator

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
      (new Thread(new PCConsumer(drop))).start();
   }}
```

```
Message received: m1
Message received: m2
Message received: m2
Message received: m3
Message received: m4
```

- Modelul Producator-Consumator

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();

      (new Thread(new PCConsumer(drop))).start();
      (new Thread(new PCConsumer(drop))).start();
   }}
```

```
Message received: m1
Message received: m2
Message received: m2
Message received: m3
Message received: m4
```

comportament nedorit

- Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() {
      while (empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      empty = true;
      notifyAll();
      return message;
}
public synchronized String put(String message) {..}}
```

- implementarea foloseste *blocuri cu garzi*
- thread-ul este suspendat pana cand o anume conditie este satisfacuta

- **testarea unei conditii** se face intotdeauna folosind **while**

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

- Modelul Producator-Consumator – implementarea buffer-ului

```
public class PCDrop {

   private String message;
   private boolean empty = true;

public synchronized String take() { … return message;}

public synchronized void put(String message) {
            while (!empty) {
                        try {
                              wait();
                        } catch (InterruptedException e) {}
                  }
      empty = false;
      this.message = message;
      notifyAll(); }}
```

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

➢ Modelul Producator-Consumator

```
public class ProducerConsumer {

   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new PCProducer(drop))).start();
      (new Thread(new PCConsumer(drop))).start();
      (new Thread(new PCProducer(drop))).start();
      (new Thread(new PCConsumer(drop))).start();

   }}
```

```
Message received: m1
Message received: m1
Message received: m2
Message received: m3
Message received: m2
Message received: m4
Message received: m3
Message received: m4
```

➢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
  }}
```

- Contor implementat cu 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

- Interfata Lock

interface **Lock**

class **ReentrantLock**

Metode:
lock(), unlock(), tryLock()

Lock vs **synchronized**

- **syncronized** acceseaza lacatul intern al resursei si impune o programare structurata: primul thread care detine resursa trebuie sa o si elibereze

- obiectele din clasa **Lock** nu acceseaza lacatul resursei ci **propriul lor lacat**, permitand mai multa flexibilitate

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

- 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

- class ReentrantLock

```
import java.util.concurrent.locks.*;
public class Interferencelock {

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

```
class Counter{
private int counter = 0;
private Lock counter_lock = new ReentrantLock();
public  void performTask () {
    counter_lock.lock();
     try {  …
         }
  finally{counter_lock.unlock();}
  }}
```

- class ReentrantLock

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

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

➢ 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

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

- Exemplul Producator-Consumator cu obiecte Lock in locul metodelor sincronizate

```
public class PCDrop {

   private String message;
   private boolean empty = true;

   private Lock dropLock = new ReentrantLock();
   private Condition cond_dropLock =   dropLock.newCondition();

   public String take() {  …
                              return message;  }

  public String put(String message) {  …   }
```

- Exemplul Producator-Consumator in care folosim obiecte Lock in locul metodelor sincronizate

```
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();}
   }
```

- Exemplul Producator-Consumator cu doua obiecte Condition pentru acelasi obiect Lock

```
public class PCDrop {

   private Queue<String> drop = new LinkedList<>();
   private static int Max = 5;

   private Lock dlock = new ReentrantLock();
   private Condition cond_empty = dlock.newCondition();
   private Condition cond_full = dlock.newCondition();

   public String take() {  …
                              return message;  }

  public String put(String message) {  …   }
```

buffer cu capacitate

cond_empty semnaleaza ca *exista spatiu* pentru a pentru a produce

cond_full semnaleaza ca *exista produse* care pot fi consumate

- Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

```
public String take() {
     dlock.lock();
      try{
      while (drop.size() == 0) {
         try {
            cond_full.await();
         }
         catch (InterruptedException ex) {}
      }
      String message  = drop.remove();
      System.out.format("Buffer items: %d%n", drop.size());

      cond_empty.signalAll();
      return message;
      }       finally { dropLock.unlock(); }
   }
```

cond_empty semnaleaza ca *exista spatiu* pentru a pentru a produce

cond_full semnaleaza ca *exista produse* care pot fi consumate

- Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

```
public String put() {
      dlock.lock();
       try{
       while (drop.size() == Max) {
          try {
             cond_empty.await();
          }
          catch (InterruptedException ex) {}
       }
       drop.add(message);
       System.out.format("Buffer items: %d%n", drop.size());

       cond_full.signalAll();
        }       finally { dropLock.unlock(); }
   }
```

cond_empty semnaleaza ca *exista spatiu* pentru a pentru a produce

cond_full semnaleaza ca *exista produse* care pot fi consumate

- Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

```
class PCProducer implements Runnable {
   private PCDrop drop;

   public PCProducer(PCDrop drop) {
      this.drop = drop;
   }

   public void run() {
      Random random = new Random();

      while (true) {
         drop.put("Message" + random.nextInt(50));
         try {
            Thread.sleep(random.nextInt(50));
         }
         catch (InterruptedException ex) {

         }}}}
```

```
class PCConsumer implements Runnable {
   private PCDrop drop;

   public PCConsumer(PCDrop drop) {
      this.drop = drop;}

   public void run() {
      Random random = new Random();
      while (true) {String message = drop.take();
         System.out.format("Message received:
                                          %s%n", message);
         try {
               Thread.sleep(100);
         }
         catch (InterruptedException ex) {
         }}}
```

- Exemplul Producator-Consumator cu doua obiecte Condition si coada cu capacitate limitata

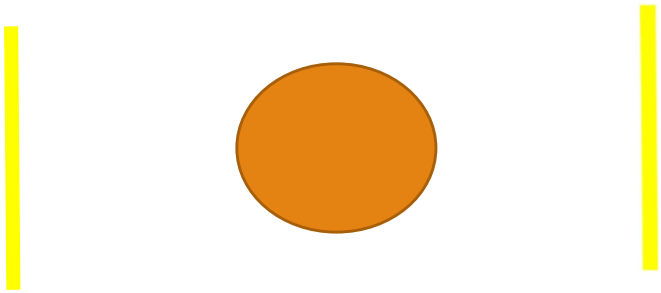
```
public class ProducerConsumerlockcond {

public static void main(String[] args) {
        PCDrop drop = new PCDrop();
        (new Thread(new PCProducer(drop))).start();
        (new Thread(new PCProducer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();
        (new Thread(new PCConsumer(drop))).start();
    }}
```

```
Buffer items: 1
Buffer items: 2
Buffer items: 1
Message received: Message49
Buffer items: 0
Message received: Message34
Buffer items: 1
Buffer items: 0
Message received: Message44
Buffer items: 1
Buffer items: 2
Buffer items: 3
Buffer items: 4
Buffer items: 5
Buffer items: 4
Message received: Message46
Buffer items: 3
Message received: Message14
Buffer items: 4
Buffer items: 5
Buffer items: 4
Message received: Message42
Buffer items: 5
Buffer items: 4
Message received: Message34
Buffer items: 3
```

Verificam ca bufferul nu va depasi capacitatea maxima

- 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;   }
```

```
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.

```
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(){…}
}
```

```
public void run() {
  try {

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

```
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

```
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.

```
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)

- 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();
             }}}
```

```
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
```

Pe săptămâna viitoare!