Curs 6

Programare Paralela si Distribuita

Thread Safety

Forme de sincronizare - Java

Thread Safety si *Shared Resources*

- Codul care poate fi apelat simultat de mai multe threaduri si produce intotdeauna rezultatul dorit/asteptat se numeste ***thread safe.***

- Daca o bucata de cod este *thread safe* atunci nu contine *critical race conditions*.

- In multithreading *Race condition* apare atunci cand mai multe threaduri actualizeaza resurse partajate.
  - care pot fi acestea acestea….?

Thread Control Escape Rule

- Daca o resursa este creata, folosita si eliminata in interiorul controlului aceluiasi thread atunci folosirea acelei resurse este *thread safe*.

Variabile Locale

- Sunt stocate pe stiva de executie a fiecarui thread.
- Prin urmare nu sunt niciodata partajate.
  - => *thread safe.*

```
public void someMethod(){
 long threadSafeInt = 0;
 threadSafeInt++;
}
```

Referinte Locale

- Referintele nu sunt partajate (orice obiect este accesibil printr-o referinta).
- Obiectul referit este partajat (*shared heap*).
- Daca un obiect creat local nu se foloseste decat local in metoda care il creeaza atunci este *thread safe.*
- Daca un obiect creat local este transferat altor metode dar nu este transferat altor threaduri atunci este *thread safe*.

Cum se asigura ca nu va fi transferat altor threaduri???

Ex:

```
public void someMethod(){
  LocalObject localObject = new LocalObject();
  localObject.callMethod();
  method2(localObject);
}
public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

Thread-safe class

- ***Thread-safe class***
  - daca comportamentul instantelor sale este corect chiar daca sunt accesate din threaduri multiple - indiferent de executia intretesuta a lor(interleaving)
    fara sa fie nevoie de sincronizari aditionale sau alte conditii impuse codului apelant.
  - sincronizarile sunt incapsulate in interior si astfel clientii clasei nu trebuie sa foloseasca altele speciale.
- Similar ***Thread-safe code***

Exemplu: not thread safe

```
public class NotThreadSafe{

   StringBuilder builder =
        new StringBuilder();

   public void add(String text){
      this.builder.append(text);
   }

public static void main(String[]a){

NotThreadSafe sharedInstance =
      new NotThreadSafe();

new Thread(new
      MyRunnable(sharedInstance)).start();
new Thread(new
      MyRunnable(sharedInstance)).start();

}
```

```
public class MyRunnable implements
      Runnable{
  NotThreadSafe instance = null;

  public MyRunnable(NotThreadSafe
      instance){
    this.instance = instance;
  }

  public void run(){
    this.instance.add(“text LUNG");
  }
}
```

Thread-Safe shared variables

- Daca mai multe thread-uri folosesc o variabila mutabila(modificabila) fara sa foloseasca sincronizari codul ***nu este safe***.

- Solutii:
  – eliminarea partajarii valorii variabilei intre threaduri
  – transformarea variabile in variabila_imutabila (var imutabile sunt thread-safe)
  – sincronizarea accesului la starea variabilei

Forme de sincronizare Java

Excludere mutuala

- Fiecare obiect din Java are un *lock/mutex* care poate fi blocat sau deblocat in blocurile sincronizate:

- *Bloc sincronizat*

```
Object critical_object = new Object();
synchronized (critical_object) {
    // critical section
}
```

:> sau *metoda* (obiectul blocat este “this”)

```
synchronized type metoda(args) {
    // body
}
```

- echivalent

```
type metoda(args) {
    synchronized (this) {
        // body
    }
}
```

monitor

Exemplu

```java
public class SynchronizedCounter {
    private int c = 0;


     public void increment() {
         synchronized (this) {
                c++;
         }
    }
    public synchronized void decrement() {
        c--;
    }
```

este necesar?

```java
    public synchronized int value() {
        return c;
    }

}
.... SynchronizedCounter co=..new SynchronizedCounter ();

T1 = new MyThread(co)....   ...run(){   co.increment();  }
T2 = new MyThread(co) ..........run(){  co.decrement();  }
T3 = new MyThread(co) ..........run(){  co.value();  }
```

Monitor in Java

Prin metodele synchronized monitoarele pot fi emulate

- nu e monitor original
- variabilele conditionale nu sunt explicit disponibile, dar metodele
  - wait()
  - notify() // signal
  - notifyAll() // signal_all

pot fi apelate din orice cod synchronized

≈ variabila conditionala implicita

- Disciplina = ‘ Signal and Continue’
- nu este starvation-free – notify() deblocheaza un proces arbitrar.

Synchronized Static Methods

```
Class Counter{
static int count;
int x;
  public synchronized  int getX(){ return x;}

  public static synchronized void add(int value){
     count += value;
  }
  public static synchronized void decrease(int value){
     count -= value;
 }
}
```

-> blocare pe *class object of the class* => Counter.class

- Ce se intampla daca sunt mai multe metode statice sincronizate ?

fine-grained synchronization

```
public class CounterC1C2 {

   private long c1 = 0;
   private long c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void inc1() {
      synchronized(lock1) {
         c1++;
      }
   }

   public void inc2() {
      synchronized(lock2) {
         c2++;
      }
   }

}
```

•Ce se intampla daca lock1 sau lock2 se modifica?

- Ce se intampla daca sunt metode de tip instanta sincronizate dar si metode statice sincronizate?

Counter => fine-grained synchronization

```
public class Counter {
   private long c = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void inc() {
      synchronized(lock1) {
         c++;
      }
   }
   public void dec() {
      synchronized(lock2) {
         c--;
      }
   }
}
```

- Este corect?

- Ce probleme exista?

Nonblocking Counter – Atomic variables

```
public class NonblockingCounter {
   private AtomicInteger value;

   public int getValue() {
      return value.get();
   }

   public int increment() {
      int v;
      do {
         v = value.get();
      } while (!value.compareAndSet(v, v + 1));
      return v + 1;
   }
}
```

Operatii atomice

- Operaţii cu întregi:
  - incrementare, decrementare, adunare, scădere
  - **compare-and-swap** (**CAS**) operatii

CAS – instructiune ***atomica*** care compara continutul memoriei cu o valoare data si doar daca aceastea sunt egala modifica continutul locatiei de memorie cu noua valoare data.

The value of CAS is that it is implemented in hardware and is extremely lightweight (on most processors).

Compare and swap (CAS)
http://www.ibm.com/developerworks/library/j-jtp11234/

- Intel...The first processors that supported concurrency provided atomic test-and-set operations, which generally operated on a single bit.

- The most common approach taken by current processors, including Intel and Sparc processors, is to implement a primitive called *compare-and-swap*, or CAS.

- On Intel processors, compare-and-swap is implemented by the cmpxchg family of instructions.
- PowerPC processors have a pair of instructions called "load and reserve" and "store conditional" that accomplish the same goal; similar for MIPS, except the first is called "load linked.”

CAS operation

- A CAS operation includes three operands
  - a memory location (V),
  - the expected old value (A), and
  - a new value (B).
- The processor will atomically update the location to the new value(V==B) if the value that is there matches the expected old value (V==A), otherwise it will do nothing.
- In either case, it returns the value that was at that location prior to the CAS instruction.

CAS synchronization

- The natural way to use CAS for synchronization is to read a value A from an address V, perform a multistep computation to derive a new value B, and then use CAS to change the value of V from A to B.
  - The CAS succeeds if the value at V has not been changed in the meantime.

- Instructions like CAS allow an algorithm to execute a read-modify-write sequence without fear of another thread modifying the variable in the meantime, because if another thread did modify the variable, the CAS would detect it (and fail) and the algorithm could retry the operation.

java.util.concurrent.atomic

Class Summary

Class	Description
AtomicBoolean	A `boolean` value that may be updated atomically.
AtomicInteger	An `int` value that may be updated atomically.
AtomicIntegerArray	An `int` array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile` `int` fields of designated classes.
AtomicLong	A `long` value that may be updated atomically.
AtomicLongArray	A `long` array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile` `long` fields of designated classes.
AtomicMarkableReference<V>	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated `volatile` reference fields of designated classes.
AtomicStampedReference<V>	An `AtomicStampedReference` maintains an object reference along with an integer "stamp", that can be updated atomically.

A small toolkit of classes that support lock-free thread-safe programming on single variables.

AtomicInteger

java.lang.Object
java.lang.Number
java.util.concurrent.atomic.AtomicInteger

int	**addAndGet**(int delta)
boolean	**compareAndSet**(int expect, int update)
int	**decrementAndGet**()
double	**doubleValue**()
float	**floatValue**()
int	**get**()
int	**getAndAdd**(int delta)
int	**getAndDecrement**()
int	**getAndIncrement**()
int	**getAndSet**(int newValue)
int	**incrementAndGet**()
int	**intValue**()
void	**lazySet**(int newValue)
long	**longValue**()
void	**set**(int newValue)
String	**toString**()
boolean	**weakCompareAndSet**(int expect, int update)

Exemplu

```
class Sequencer {

   private final AtomicLong sequenceNumber  = new AtomicLong(0);

   public long next() {
     return sequenceNumber.getAndIncrement();
   }
 }
```

Thread Signaling

- Permite transmiterea de semnale/mesaje de la unul thread la altul.
- Un thread poate astepta un semnal de la altul.
- Asteptare conditionata

Signaling via Shared Objects

- Setarea unei variabile partajate- ***comunicare prin variabile partajate.***

```
public class MySignal{

  protected boolean hasDataToProcess = false;

  public synchronized boolean hasDataToProcess(){
    return this.hasDataToProcess;
  }

  public synchronized
      void setHasDataToProcess(boolean hasData){
    this.hasDataToProcess = hasData;
  }

}
```

Busy Wait - INCORECT

- Thread B asteapta ca data sa devina disponibila pentru a o procesa.
    - => asteapta un semnal de la threadul A

=> hasDataToProcess() to return true.

- **Busy waiting NU implica o utilizare eficienta a CPU** (cu exceptia situatiei in cate timpul mediu de asteptare este foarte mic).
- este de dorit ca asteptarea sa fie inactiva (fara folosire procesor) –
- poate sa produca blocaj!

```
protected MySignal sharedSignal = ...

...

while( ! sharedSignal.hasDataToProcess()){
  //do nothing... busy waiting  ;
}
```

Operatii

- wait()
- notify() // signal
- notifyAll() // signal_all

Exemplul 1 –> Producator- Consumator / Buffer de dimensiune = 1

```
public class Producer extends Thread {

… ITER

    private Location loc;

    private int number; //id

    public Producer(Location c, int number) {
        loc = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < ITER; i++) {
                loc.put(i);

        }
    }
}
```

```
public class Consumer extends Thread {

… ITER

    private Location loc;

    private int number; //id

    public Consumer(Location c, int number) {
        loc = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < ITER; i++) {
            value = loc.get();

        }
    }
}
```

```
public class Location {
   private int contents;          // shared data : didactic
   private boolean available = false;

/* Method used by the consumer to access the shared data */
   public synchronized int get() {
      while (available == false) {
         try {
            this.wait();          // Consumer enters a wait state until notified by the Producer
         } catch (InterruptedException e) { }
      }
      available = false;
      this.notifyAll();           // Consumer notifies Producers that it can store new contents
      return contents;
   }

/* Method used by the consumer to store the shared data */
   public synchronized void put (int value) {
      while (available == true) {
         try {
            this.wait();          // Producer who wants to store contents enters
                        // a wait state until notified by the Consumer
         } catch (InterruptedException e) { }
      }
      contents = value;
      available = true;
      this.notifyAll();           // Producer notifies Consumer to come out
                        // of the wait state and consume the contents
   }
}
```

Exemplul 2: BlockingQueue : buffer size >1

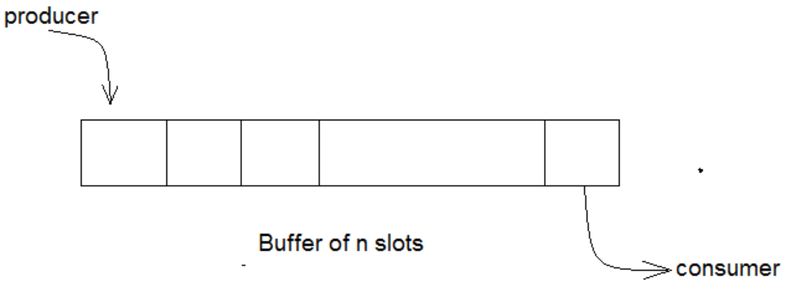
```
class BlockingQueue {
  int n = 0;
  Queue data = ...;

  public synchronized Object remove() {
    // wait until there is something to read
    while (n==0)
            this.wait();

    n--;
    // return data element from queue
  }

  public synchronized void write(Object o) {
    n++;
    // add data to queue (considere that there is unlimited space)

     notifyAll();
  }
}
```



Missed Signals- Starvation

- Apelurile metodelor notify() si notifyAll() nu se salveaza in cazul in care nici un thread nu asteapta atunci cand sunt apelate.
- Astfel semnalul notify se poate pierde.
- Acest lucru poate conduce la situatii in care un thread asteapta nedefinit, pentru ca mesajul corespunzator de notificare s-a pierdut anterior.

- Propunere:
  - Evitarea problemei prin salvarea semnalelor in interiorul clasei care le trimite.
- =>analiza!

```
public class MyWaitNotify2{

  MonitorObject myMonitorObject = new MonitorObject();
  boolean wasSignalled = false;

  public void doWait(){
    synchronized(myMonitorObject){
      if(!wasSignalled){
        try{
          myMonitorObject.wait();
         } catch(InterruptedException e){...}
      }
      //clear signal and continue running.
      wasSignalled = false;
    }
  }

  public void doNotify(){
    synchronized(myMonitorObject){
      wasSignalled = true;
      myMonitorObject.notify();
    }
  }
}
```

Lock

Oracle docs:

public interface **Lock**

- Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.
- **They allow more flexible structuring**, may have quite different properties, and may support **multiple associated Condition objects.**

Modifier and Type	Method and Description
void	**lock**()Acquires the lock.
void	**lockInterruptibly**()Acquires the lock unless the current thread is **interrupted**.
Condition	**newCondition**()Returns a new **Condition** instance that is bound to this Lock instance.
boolean	**tryLock**()Acquires the lock only if it is free at the time of invocation.
boolean	**tryLock**(long time, **TimeUnit** unit)Acquires the lock if it is free within the given waiting time and the current thread has not been **interrupted**.
void	**unlock**()Releases the lock.

Lock (java.util.concurrent.locks.Lock)

```
public class Counter{

  private int count = 0;

  public int inc(){
    synchronized(this){
      return ++count;
    }
  }
}
```

```
public class Counter{
  private
Lock lock = new ReentrantLock();
  private int count = 0;

  public int inc(){
    lock.lock();
        try{
                int newCount = ++count;  }
        finally{
    lock.unlock();  }
    return newCount;
  }
}
```

Metode ale interfetei Lock

lock()
lockInterruptibly()
tryLock()
tryLock(long timeout, TimeUnit timeUnit)
unlock()

The lockInterruptibly() method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

Diferente Lock vs synchronized

- Nu se poate trimite un parametru la intrarea intr-un bloc synchronized => nu se poate preciza o valoare timp corespunzatoare unui interval maxim de asteptare-> timeout.

- Un bloc synchronized trebuie sa fie complet continut in interiorul unei metode
  - lock() si unlock() pot fi apelate in metode separate.

Lock Reentrance

- Blocurile sincronizate in Java au proprietatea de a permite ‘reintrarea’ (*reentrant Lock)*.
- Daca un thread intra intr-un bloc sincronizat si blocheaza astfel monitorul obiectului corespunzator, atunci threadul poate intra in alt cod sincronizat prin monitorul aceluiasi obiect.

```
public class Reentrant{
 public synchronized outer(){
  inner();
 }
 public synchronized inner(){
  //do something
 }
}
```

Conditions in Java

- java.util.concurrent.locks
- Interface Condition

- Avantaj fata de "wait-notify" din monitorul definit pentru Object
- Imparte metodele (wait, notify , notifyAll) in obiecte distincte pentru diferite conditii
  - permite mai multe *wait-sets per object.*

Exemplu – Prod-Cons FIFO Buffer

```
class BoundedBuffer {
   static final MAX = 100;
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[MAX];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException
{
    lock.lock();
    try {
      while (count == items.length)
                        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```

```
public Object take() throws InterruptedException {
   lock.lock();
   try {
     while (count == 0)
       notEmpty.await();
     Object x = items[takeptr];
     if (++takeptr == items.length) takeptr = 0;
     --count;
     notFull.signal();
     return x;
   } finally {
     lock.unlock();
   }
  }
}
```

Semaphore
(java.util.concurrent.Semaphore)

- Semafor binar (=> excludere mutuala)

```
Semaphore semaphore = new Semaphore(1);

//critical section
semaphore.acquire();
…
semaphore.release();
```

- Fair/Strong Semaphore

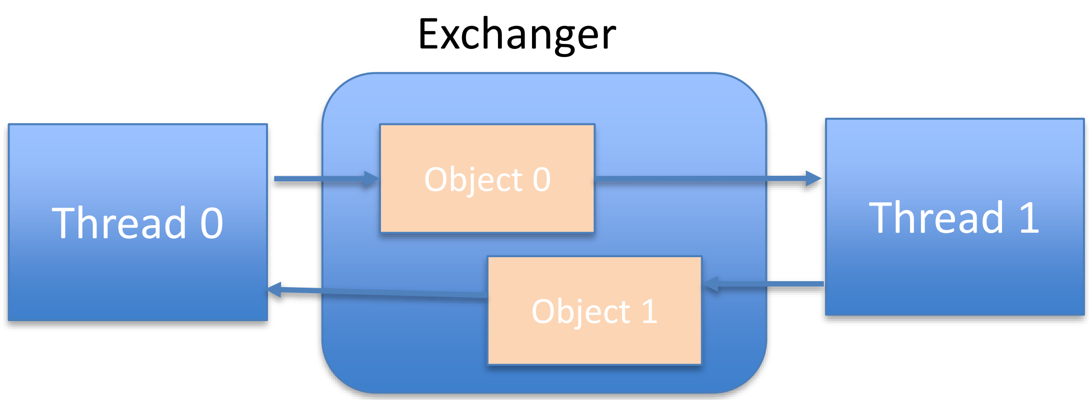
```
Semaphore semaphore = new Semaphore(1, true);
```

Conceptul de intalnire (**Rendez-vous**)

- Conceptul de întâlnire (rendez-vous) a fost introdus initial in limbajul Ada pentru a facilita comunicarea între două task-uri.

a) Procesul B este gata să transmită informaţiile, dar procesul A nu le-a cerut încă. In acest caz, procesul B rămâne în aşteptare până când procesul A i le cere.
b) Procesul B este gata să transmită informaţiile cerute, iar procesul A cere aceste date. In acest caz, se realizează un *rendez-vous*, cele două procese lucrează sincron până când îşi termină schimbul, după care fiecare îşi continuă activitatea independent.
c) Procesul A a lansat o cerere, dar procesul B nu este în măsură să-i furnizeze informaţiile solicitate. In acest caz, A rămâne în aşteptare până la întâlnirea cu B.

Java Exchanger <-> Rendez-Vous



Jacob Jenkov Tutorial

Thread-0 exchanged Object0 for Object1
Thread-1 exchanged Object1 for Object0

Exchanger
(java doc)

public V exchange(V x)   throws InterruptedException

- Waits for another thread to arrive at this exchange point (unless the current thread is interrupted), and then transfers the given object to it, receiving its object in return.
- If another thread is already waiting at the exchange point then it is resumed for thread scheduling purposes and receives the object passed in by the current thread.
  - The current thread returns immediately, receiving the object passed to the exchange by that other thread.

If no other thread is already waiting at the exchange then the current thread is disabled for thread scheduling purposes and lies dormant until one of two things happens:

- Some other thread enters the exchange; or
- Some other thread interrupts the current thread.

If the current thread:

- has its interrupted status set on entry to this method; or
- is interrupted while waiting for the exchange,
  - then InterruptedException is thrown and the current thread's interrupted status is cleared.

```
Exchanger exchanger = new Exchanger();

ExchangerRunnable exchangerRunnable1 =    new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 =    new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();
new Thread(exchangerRunnable2).start();
```

```
public class ExchangerRunnable implements Runnable{
   Exchanger exchanger = null;
   Object    object    = null;

   public ExchangerRunnable(Exchanger exchanger, Object object) {
      this.exchanger = exchanger;
      this.object = object;
   }

   public void run() {
      try {
         Object previous = this.object;

                  Object received =
                                          this.exchanger. exchange (this.object);
                  this.object = received;

         System.out.println(
               Thread.currentThread().getName() +
               " exchanged " + previous + " for " + this.object
         );
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

Class SynchronousQueue (->*Rendez-vous*) Java doc

java.lang.Object
java.util.AbstractCollection<E>
java.util.AbstractQueue<E>
java.util.concurrent.SynchronousQueue<E>

A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. ***A synchronous queue does not have any internal capacity, not even a capacity of one.*** You cannot peek at a synchronous queue because an element is only present when you try to remove it; ***you cannot insert an element (using any method) unless another thread is trying to remove it***; you cannot iterate as there is nothing to iterate. The *head* of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements [https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html]

- boolean **offer(E e)**
  - Inserts the specified element into this queue, if another thread is waiting to receive it.
- void **put(E o)**
  - Adds the specified element to this queue, waiting if necessary for another thread to receive it.

- **E** **poll()**
  - Retrieves and removes the head of this queue, if another thread is currently making an element available.

ReadWriteLock

- Read Access -> daca nici un alt thread nu scrie si nici nu cere acces pt scriere.
- Write Access -> daca nici alt un thread nici nu scrie nici nu citeste.

public interface **ReadWriteLock**

- A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing.
- The read lock may be held simultaneously by multiple reader threads, so long as there are no writers.
- The write lock is exclusive.

Exemplu

```java
public class TSArrayList<E>
{
   private final ReadWriteLock readWriteLock =
                   new ReentrantReadWriteLock();

   private final Lock readLock = readWriteLock.readLock();
   private final Lock writeLock = readWriteLock.writeLock();

   private final List<E> list = new ArrayList<>();
```

```java
public void set(E o)
{
   writeLock.lock();
   try
   {
                  list.add(o);

   }
   finally
   {
      writeLock.unlock();
   }
}
```

```java
 public static void main(String[] args)
   {
      TSArrayList<String> threadSafeArrayList =
                                   new TSArrayList<>();
      threadSafeArrayList.set("1");
      threadSafeArrayList.set("2");
      threadSafeArrayList.set("3");

      System.out.println(threadSafeArrayList.get(1));
   }
}
```

```java
public E get(int i)
{
   readLock.lock();
   try
   {
            return list.get(i);
   }
   finally
   {
      readLock.unlock();
   }
}
```

Bariera de sincronizare

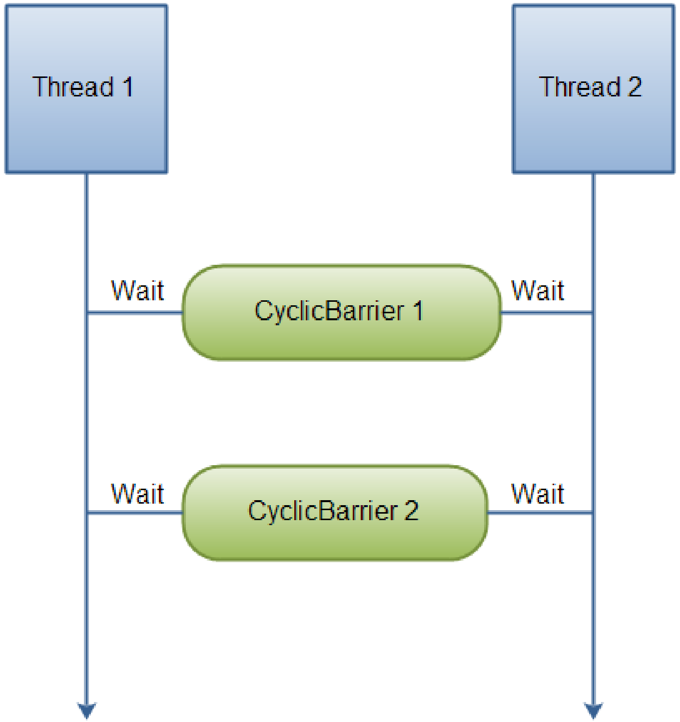
```
CyclicBarrier barrier = new CyclicBarrier(2);
// 2 = no_of_threads_to_wait_for

barrier.await();

barrier.await(10, TimeUnit.SECONDS);
```

Bariera de sincronizare:

- Bariera secventiala – pt implementare se fol. in general 2 variabile – {no_threads(0..n), state(stop/pass)}
- Bariera ierarhica (tree-barrier)



Jacob Jenkov Tutorial

java.util.concurrent>

java.util.concurrent.CyclicBarrier (java documentation)

- A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other.
- The barrier is called cyclic because it can be re-used after the waiting threads are released.
- A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.
  - CyclicBarrier(int parties)
  - CyclicBarrier(int parties, Runnable barrierAction)