UD5. Biblioteca java.util.concurrent

Concurrència i Sistemes Distribuïts

Objectius de la Unitat Didàctica

- Identificar les limitacions de les primitives bàsiques Java estudiades en la UD3 (on es van explicar els monitors).

- Conèixer la biblioteca java.util.concurrent, que ofereix interfícies i classes per a facilitar el desenvolupament d'aplicacions concurrents.
  - Il·lustrar la utilització de les eines que ens ofereix aquesta biblioteca o package.

Contingut

- Inconvenients de les primitives bàsiques de Java
- Java.util.concurrent
- Elements de la biblioteca

Inconvenients de les primitives bàsiques de Java

- Les primitives bàsiques de Java vistes fins ara són molt útils en aplicacions simples, però insuficients per a aplicacions complexes
  - Recordatori:
    - Suport per a monitors, on es garanteix exclusió mútua entre tots els mètodes qualificats com "synchronized" per a un determinat objecte.
    - Declaració implícita d'una condició interna per a cada objecte
      - Es poden utilitzar els mètodes wait(), notify() i notifyAll(), que actuen sobre la condició interna implícita.
      - Amb açò s'implementa una forma limitada de la variant de Lampson i Redell.

Inconvenients de les primitives bàsiques de Java

- Limitacions de les primitives bàsiques de Java relacionades amb **l'exclusió mútua**:
  1. No es pot establir un termini màxim d'espera a l'hora de "sol·licitar" l'entrada al monitor
     - Si el monitor està ocupat, el fil es queda esperant i no es pot interrompre voluntàriament.
  2. No es pot preguntar per l'estat del monitor abans de sol·licitar l'accés.
     - En ocasions interessa consultar si el *lock* (o monitor) està lliure o ocupat sense necessitat de bloquejar-se (Exemple: usant `tryLock`)
       - Especialment útil per a evitar possibles interbloquejos.
       - Amb mètodes `synchronized` aquesta consulta no és possible.

Inconvenients de les primitives bàsiques de Java

- Limitacions de les primitives bàsiques de Java relacionades amb **l'exclusió mútua** (cont.):

3. Les eines que garanteixen exclusió mútua estan orientades a blocs.
   - És a dir, estan orientades a mètodes complets o a seccions de codi més xicotetes.
   - No podem adquirir un *lock* en un mètode i alliberar-lo dins d'un altre mètode.

4. No podem estendre la seua semàntica.
   - Exemple: en problema lectors-escriptors, no podem utilitzar aquestes construccions per a definir exclusió mútua entre fils escriptors o entre escriptors i lectors, però no entre múltiples lectors.

Inconvenients de les primitives bàsiques de Java

- Limitacions relacionades amb **la sincronització condicional:**
  1. Solament podrà existir una única condició en cada monitor.
     - Tots els fils que se suspenen en un monitor van a parar a una mateixa cua, amb independència del motiu (condició) pel qual se suspenen.
  2. S'utilitza la variant de Lampson i Redell.
     - El programador està obligat a utilitzar una estructura del tipus:

       `while (expressió lògica) wait();`

       per a consultar l'estat del monitor i suspendre's.

Contingut

- Inconvenients de les primitives Java
- Java.util.concurrent
- Elements de la biblioteca

Biblioteca java.util.concurrent

- J2ES 5.0 introdueix el paquet java.util.concurrent
  - Ofereix construccions d'alt nivell.
  - Garanteixen:
    - + Productivitat
      - Facilita desenvolupament/manteniment d'aplicacions concurrents de qualitat.
    - + Prestacions
      - Més eficient i escalable que les implementacions típiques.
    - + Fiabilitat
      - Comprovacions extensives contra interbloquejos, condicions de carrera, inanició...

Elements en java.util.concurrent

- La biblioteca inclou diversos elements útils:
  - Locks
  - Variables condició
  - Col·leccions concurrents
  - Variables atòmiques
  - Sincronització: Semàfors i Barreres
  - Entorn per a l'execució de fils
  - Temporització precisa

Locks

- java.util.concurrent.locks proporciona diferents classes i interfícies per a la gestió i desenvolupament de múltiples tipus de *locks*.
- Característiques:
  - Permet especificar si es requereix una gestió equitativa (*fair*) de la cua d'espera mantinguda pel *lock*.
  - Es faciliten diferents tipus de *locks*, amb semàntica diferent.
    - Exemple: *locks* orientats a exclusió mútua, *locks* que resolen el problema de lectors-escriptors.
  - Ofereix un mètode tryLock() que no suspèn l'invocador si el *lock* ja ha sigut tancat per un altre fil.
    - Es trenca així la condició de retenció i espera (vista en la Unitat 4 – Interbloquejos).

Locks: ReentrantLocks

- Exemple de classe *lock* oferida: ReentrantLock
  - Implementa un *lock* reentrant:
    - Dins de la secció de codi protegida pel *lock* es podrà tornar a utilitzar aquest mateix *lock* sense que hi haja problemes de bloqueig.
  - Resol les limitacions de la sentència 'synchronized'
    - Implementació molt eficient.
    - Permet especificar un termini màxim d'espera per a obtenir el *lock*.
      - Utilitzant el mètode tryLock indicant un *timeout*.
    - Permet definir diferents variables condició, segons es requerisca en el monitor.
      - Utilitzant el mètode newCondition*()*
    - Permet tancar i obrir els *locks* en diferents mètodes de l'aplicació.
      - No restringeix l'ús dels objectes *lock* a la construcció de monitors.
    - Suport per a interrupcions sobre fils que esperen adquirir un *lock*.
      - Es poden interrompre les esperes usant Thread.interrupt()

Locks: ReentrantLocks

- Inconvenients:
  - Amb un monitor bàsic de Java, la gestió de *locks* és implícita.
    - El programador no ha de preocupar-se del tancament i obertura dels *locks*.
    - Però amb **ReentrantLock**, el programador ha d'assegurar-se d'obrir el *lock* corresponent en alliberar un recurs que s'use en exclusió mútua.
  - S'han de controlar les excepcions que es puguen generar dins d'una secció crítica protegida per ***un ReentrantLock.***
    - El codi associat a l'excepció ha d'assegurar que s'execute el mètode unlock() sobre aquest ReentrantLock.

- Recomanació: seguir aquesta estructura dels protocols d'entrada i eixida:

```
Lock x = new ReentrantLock ();
x.lock();                    // Protocol d'entrada.
try {

... //Secció crítica  (on s'actualitza l'estat de l'objecte)
} finally {
   x.unlock();               // Protocol d'eixida.
}
```

Variables condició

- Recordem que la classe **Object** inclou mètodes especials per a sincronització entre fils: wait, notify, notifyAll.
  - És fàcil usar-los de forma incorrecta.
  - Existeix interacció entre notificació i locking.
    - Per a esperar o notificar cal tancar prèviament el "*lock*" associat a l'objecte.
- Amb java.util.concurrent tenim construccions de major nivell, i per tant es redueix la necessitat d'utilitzar wait/notify/notifyAll.
- Però si es necessiten, podem utilitzar *variables condició*:
  - La interfície **Condition** permet declarar qualsevol nombre de variables condició en un *lock*.
    - En molts casos utilitzar múltiples variables condició permet un codi més llegible i eficient.
      - Per exemple, en el problema dels productors/consumidors.
  - L'objecte *lock* actua com una factoria per a variables condició associades a aquest.
    - Només podem definir variables **condició** dins d'un *lock*.

Variables Condició

- El mètode newCondition() de la classe **ReentrantLock** permet generar totes les cues d'espera (condicions) que resulten necessàries.
- Aquest mètode retorna un objecte que implanta la interfície Condition, que ofereix els següents mètodes:
  - **await**(): suspèn un fil en la condició.
    - Inclou variants per a especificar temps màxims d'espera, *deadlines*, gestió d'interrupcions.
  - **signal**(): notifica l'ocurrència de l'esdeveniment esperat a un dels fils que estiguera esperant-lo.
  - **signalAll**(): notifica l'ocurrència de l'esdeveniment esperat a tots els fils suspesos en la condició.
- Observeu que en els monitors bàsics, els mètodes amb funcionalitat similar s'anomenen: wait(), notify(), notifyAll().

Exemple.- Lock i condition

```
class BufferOk implements Buffer {
    private int elems, cap, cua, N;
    private int[] dades;
    Condition noPle, noBuit;
    ReentrantLock lock;

    public BufferOk(int N) {
        dades= new int[N];
        this.N=N;
        cap = cua = elems = 0;
        lock= new ReentrantLock();
        noPle=lock.newCondition();
        noBuit=lock.newCondition();
    }
```

```
    public int get() {
        int x;
        try {
            lock.lock();
            while (elems==0) {
                System.out.println("Consumidor esperant ...");
                try {noBuit.await();}
                catch(InterruptedException e) {}
            }
            x=dades[cap];  cap= (cap+1)%N;  elems--;
            noPle.signal();
        } finally {lock.unlock();}
        return x;
    }

    public void put(int x) {
        try{
            lock.lock();
            while (elems==N) {
                System.out.println("Productor esperant ...");
                try {noPle.await();}
                catch(InterruptedException e) {}
            }
            dades[cua]=x; cua= (cua+1)%N;  elems++;
            noBuit.signal();
        } finally {lock.unlock();}
    }
}
```

Col·leccions concurrents

- Moltes aplicacions requereixen accés a col·leccions d'objectes
  - *Ex.- Entre les biblioteques* Java apareixen Map *(vector associatiu)*, List *(talla dinàmica)*, Queue *(política* FIFO)
  - Però no són *thread-safe (no es poden compartir de forma segura entre diversos fils).*
- Java.util.concurrent inclou versions *thread-safe*:
  - Classes ConcurrentHashMap, ConcurrentSkipListMap, implantant la interfície Map.
  - CopyOnWriteArrayList, per a implantar la interfície List.
  - Interfície BlockingQueue, estén a Queue.
    - Implementacions concurrents, eficients de Queue:
      - ArrayBlockingQueue, ConcurrentLinkedQueue, LinkedBlockingQueue, PriorityBlockingQueue, DelayQueue, SynchronousQueue.

Col·leccions concurrents.- Exemple: BlockingQueue

- Mètodes de la interfície **BlockingQueue**:
  - Per a inserir elements en la cua:
    - **add**(): si no hi ha espai, es genera una excepció.
    - **offer**(): si no hi ha espai, retorna **false**.
    - **put**(): si no hi ha espai, el fil es queda esperant.
  - Per a extraure elements de la cua:
    - **take**(): recupera i elimina el primer element de la cua. Espera si fóra necessari fins que hi haja algun element que extraure.
    - **poll**(): igual que **take**, però si no hi ha elements en cua, s'espera com a màxim l'interval especificat.
    - **remove**(): elimina de la cua la instància de l'objecte subministrada.
    - **peek**(): retorna el primer element de la cua, sense extraure'l.
  - Altres mètodes:
    - **remainingCapacity**(): retorna el nombre d'elements que encara es poden inserir en la cua.
    - **contains**(): retorna **true** si la cua conté l'objecte indicat com a argument.
    - **drainTo**(): elimina tots els elements de la cua i els afig a la col·lecció subministrada.

Col·leccions concurrents.- Exemple: BlockingQueue

- Problema del productor/consumidor
- El *buffer* és una BlockingQueue:
  - Cua (política FIFO) de capacitat configurable.
  - El fil que vol extraure un element espera si la cua està buida.
  - El fil que vol inserir un element espera si no hi ha espai.

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { queue.put(produce()); }
    } catch (InterruptedException ex) {...}
  }
  Object produce() { ... }
}
class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { consume(queue.take()); }
    } catch (InterruptedException ex) {...}
  }
  void consume(Object x) { ... }
}
class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

Variables atòmiques (java.util.concurrent.atomic)

- Defineix classes que suporten l'accés concurrent segur a variables simples
  - tipus primitius (AtomicBoolean, AtomicInteger, AtomicLong)
  - referències (AtomicReference)
- Útil per a:
  - Implementar algorismes concurrents de forma eficient.
  - Implementació de comptadors.
  - Generadors de seqüències de nombres.
- Permet tractar atòmicament diferents operadors
  - L'operació ++ sobre un enter no és atòmica, però AtomicInteger té una operació d'increment atòmica.
  - També operacions com getAndSet, compareAndSet, etc.
- La implementació és molt eficient
  - Més del que podem obtenir utilitzant synchronized

Variables Atòmiques.- Exemple

- Amb la nostra classe

```
class ID {
   private static long nextID = 0;
   public static synchronized
      long getNext() {
         return nextID++;
   }
}
public class ExCounter extends Thread{
   ID counter;
   public ExCounter(ID c) {counter=c;}
   public void run() {
      System.out.println("counter value: "+
         (counter.getNext()));
   }
   public static void main(String[] args) {
      ID counter= new ID();
      new ExCounter(counter).start();
      new ExCounter(counter).start();
      new ExCounter(counter).start();
   }
}
```

- Amb AtomicLong
  - Mètodes *addAndGet, compareAndSet, decrementAndGet, incrementAndGet, getAndDecrement, getAndIncrement, getAndSet, toString, get, …*

```
public class ExCounter extends Thread{
   AtomicLong counter;
   public ExCounter(AtomicLong c) {counter=c;}
   public void run() {
      System.out.println("counter value: "+
         (counter.getAndIncrement()));
   }
   public static void main(String[] args) {
      AtomicLong counter= new AtomicLong(0);
      new ExCounter(counter).start();
      new ExCounter(counter).start();
      new ExCounter(counter).start();
   }
}
```

Semàfors

- La classe Semaphore permet la sincronització entre fils
  - En lloc de P() i V() s'usen els mètodes acquire() i release()
    - **acquire**(): espera fins a disposar d'un permís, i llavors el consumeix
    - **release**(): afig un permís, i possiblement allibera un fil esperant en un acquire()
  - Es pot emprar per a protegir seccions crítiques (ex: mutex) o per a sincronitzar fils.
    - Porta associat un comptador que s'inicialitza en la creació.
    - Si s'inicialitzen a 1 permeten garantir exclusió mútua en l'accés a una determinada secció de codi, usant acquire() com a protocol d'entrada i release() com a protocol d'eixida.
    - Si s'inicialitzen a un valor positiu **superior a** 1 poden utilitzar-se per a limitar el grau de concurrència en l'execució d'una determinada secció de codi.
    - Si s'inicialitzen a **zero** garanteixen cert ordre d'execució entre dos o més fils.

Sincronització.- Classe Semaphore

- Exemple 1: es pot utilitzar per a protegir seccions crítiques (iniciant el comptador a 1).

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

class Process2 extends Thread {
      private int id;
      private Semaphore sem;

      private void busy() {
            try {sleep(new java.util.Random().nextInt(500));}
            catch (InterruptedException e) {}
      }
      private void msg(String s)     {System.out.println("Thread " + id + s);}
      private void noncritical()     {msg(" is NON critical"); busy();}
      private void critical()        {msg(" entering CS"); busy(); msg(" leaving
CS");}

      public Process2(int i, Semaphore s) {id = i; sem = s;}
      public void run() {
            for (int i = 0; i < 2; ++i) {
                  noncritical();
                  try {sem.acquire();} catch (InterruptedException e) {}
                  critical();
                  sem.release();
            }
      }
      public static void main(String[] args) {
            Semaphore sem = new Semaphore(1, true); //permisos, fair?
            for (int i = 0; i < 4; i++)
                  new Process2(i, sem).start();
      }
}
```

```
C:\CSD>java Process2
Thread 0 is NON critical
Thread 3 is NON critical
Thread 1 is NON critical
Thread 2 is NON critical
Thread 2 entering CS
Thread 2 leaving CS
Thread 2 is NON critical
Thread 1 entering CS
Thread 1 leaving CS
Thread 1 is NON critical
Thread 3 entering CS
Thread 3 leaving CS
Thread 3 is NON critical
Thread 0 entering CS
Thread 0 leaving CS
Thread 0 is NON critical
Thread 2 entering CS
Thread 2 leaving CS
Thread 3 entering CS
Thread 3 leaving CS
Thread 0 entering CS
Thread 0 leaving CS
Thread 1 entering CS
Thread 1 leaving CS
```

Sincronització.- Classe Semaphore

- Exemple 2: sincronització

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

class ProdCons extends Thread {
    static final int N=6; // buffer size
    static int head=0, tail=0, elems=0;
    static int[] data= new int[N];
    static Semaphore item= new Semaphore(0,true);
    static Semaphore slot= new Semaphore(N,true);
    static Semaphore mutex= new Semaphore(1,true);

    public static void main(String[] args) {
        new Thread(new Runnable() { // producer
            public void run() {
                for (int i=0; i<10; i++) {
                    busy();
                    put(i);
                }
            }
        }).start();
        new Thread(new Runnable() {// consumer
            public void run() {
                for (int i=0; i<10; i++) {
                    busy();
                    System.out.println(get());
                }
            }
        }).start();
    }

    private static void busy() {
        try {sleep(new java.util.Random().nextInt(500));}
        catch (InterruptedException e) {}
    }

    public static int get() {
        try {item.acquire();} catch (InterruptedException e) {}
        try {mutex.acquire();} catch (InterruptedException e) {}
        int x=data[head]; head= (head+1)%N;  elems--;
        mutex.release();
        slot.release();
        return x;
    }

    public static void put(int x) {
        try {slot.acquire();} catch (InterruptedException e) {}
        try {mutex.acquire();} catch (InterruptedException e) {}
        data[tail]=x; tail= (tail+1)%N;  elems++;
        mutex.release();
        item.release();
    }
}
```

Barreres

- Permeten sincronitzar a múltiples fils d'execució.
- En java.util.concurrent existeixen dos tipus diferents de barreres:
  - CyclicBarrier, CountDownLatch.

- CyclicBarrier:
  - Permet que un conjunt de fils s'esperen mútuament en un punt comú.
  - Poden continuar quan determinat nombre de fils arriben a la barrera.
    - Els fils es van suspenent, usant el mètode await() de la barrera, fins que l'últim dels fils invoque await(). Llavors, tots els fils del grup es reactiven i podran continuar.
  - El nombre de fils a suspendre s'indica en el constructor de la barrera.
  - És reutilitzable (per açò es diu que és cíclica)
    - Quan es permet passar als fils, es restauren les condicions inicials.
    - Torna a permetre que esperen els fils que arriben, etc.
  - Ideal per a aplicacions iteratives.
    - Ex.- en cada iteració un fil treballador (Worker) processa una columna de la matriu, i un coordinador (Solver) barreja els resultats

Barreres: CyclicBarrier

- En el constructor de CyclicBarrier s'indica:
  - Nombre de fils que formen el grup.
  - Opcional: argument Runnable amb codi a executar abans de la reactivació dels fils.
- El mètode principal és **await**().

```
class Solver {
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;

  class Worker implements Runnable {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run() {
      while (!done()) {
        processRow(myRow);
        try {barrier.await();}
        catch (InterruptedException e) {return;}
        catch (BrokenBarrierException e) {return;}
      }
    }
  }
  public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;
    barrier = new CyclicBarrier(N,
      new Runnable() {
      public void run() {
          mergeRows(...);
      }
    });
    for (int i = 0; i < N; ++i)
      new Thread(new Worker(i)).start();
    waitUntilDone();
  }
}
```

Barreres: CountDownLatch

- CountDownLatch:
  - Permet suspendre un grup de fils, quedant aquests a l'espera que ocórrega algun esdeveniment que ha de ser generat per un fil aliè al grup.
  - Es manté un comptador d'esdeveniments. En el constructor s'especifica el valor inicial del comptador.
  - La barrera es crea inicialment tancada.
  - Mètodes que ofereix:
    - await(): els fils es bloquegen en la barrera mentre estiga tancada (és a dir, el seu comptador és superior a zero)
    - countDown(): decrementa en una unitat el valor del comptador (si era superior a zero).
      - Si el comptador passa a valdre 0, la barrera s'obri, alliberant tots els fils bloquejats.
      - Una vegada oberta, la barrera romandrà en aquest estat.
      - Amb la barrera oberta, si algun fil usa await(), no es bloqueja.
  - La barrera és d'un sol ús:
    - Una vegada a zero, roman així.
    - Si es desitja reutilització, usar CyclicBarrier.

Barreres: CountDownLatch

- Exemple d'ús:
  - Tenim dues barreres CountDownLatch.
    - startSignal: perquè tots els *Worker* comencen alhora.
    - doneSignal: on *Driver* espera que tots els *Worker* finalitzen el seu treball.
  - Inicialment, tots els *Worker* se suspenen en startSignal.
    - Esperen ací fins que el fil *Driver* fa un startSignal.countDown*()*.
  - El *Driver* es queda suspès en doneSignal, fins que el comptador (inicialment a N) arribe a 0.
    - Cada *Worker*, en acabar, el decrementa en una unitat amb doneSignal.countDown()

```
class Driver { // …
  void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);

    for (int i = 0; i < N; ++i)
      new Thread(new Worker(startSignal,
doneSignal)).start();
    doSomethingElse();
    startSignal.countDown(); //all threads proceed
    doSomethingElse();
    doneSignal.await();    // wait for all to finish
  }
}
class Worker implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;
  Worker(CountDownLatch start, CountDownLatch done) {
    startSignal = start;
    doneSignal = done;
  }
  public void run() {
    try {
      startSignal.await();
      doWork();
      doneSignal.countDown();
    } catch (InterruptedException ex) {return;}
  }
  void doWork() { ... }
}
```

Entorn per a l'execució de fils

- La classe Executor ofereix un entorn per a la invocació, planificació, execució i control de polítiques d'execució.
- Segons el tipus específic d'Executor que usem, podem executar tasques en:
  - Un fil de nova creació.
  - Un fil d'execució ja existent.
  - Un fil únic en *background* (ex.- com els esdeveniments en JavaFx).
  - Un *thread-pool* (ex.- per a servidor)
    - Mantenir un conjunt de fils ja creats, reciclant-los perquè executen noves tasques.
- Polítiques configurables:
  - Límit en la longitud de la cua, política de saturació...
  - Possibilitat d'executar repetidament un fil amb un període donat.
  - Preveu el consum desmesurat de recursos.
    - Genera aplicacions més estables

Temporització precisa

- Java bàsic mesura el temps mitjançant el mètode System.currentTimeMillis(), que retorna el valor del rellotge mantingut pel sistema operatiu.
  - Retorna el nombre de mil·lisegons transcorreguts des de l'1 de gener de 1970.
  - Per a mesurar intervals (ex. per a valorar rendiment) restem el valor de dues invocacions del mètode en diferents instants.
- Inconvenients de System.currentTimeMillis()
  - Aquest rellotge està fora de la JVM, i pot canviar de forma impredictible (ex. un servei NTP –servidor de temps en xarxa- pot ajustar periòdicament el rellotge).
  - La precisió depèn de la plataforma (ex.- precisió d'1ms en Unix, 50ms en Windows).
  - Conseqüència: no mesura intervals de manera exacta.

- Java 1.5.0 va introduir el mètode System.nanoTime(), que permet mesurar intervals de manera més precisa.
  - Retorna el valor actual del temporitzador més exacte del sistema (ex. temporitzador de la CPU).
  - Resultat en nanosegons.

```
long startTime = System.nanoTime();
// ... the code being measured ...
long estimatedTime = System.nanoTime() - startTime;
```

Temporització precisa

- Per a especificar la durada d'un interval de bloqueig, la biblioteca java.util.concurrent.TimeUnit introdueix el tipus enumerat TimeUnit.
  - Permet especificar la unitat temporal quan es defineix un interval de temps: MICROSECONDS, MILLISECONDS, NANOSECONDS, SECONDS
  - Es pot utilitzar per a indicar durada en tots els mètodes que accepten timeouts:
    - BlockingQueue.offer*()*, BlockingQueue.poll*()*, Lock.tryLock*()*, Condition.await*()*, Thread.sleep*()*
  - Inclou mètodes per a realitzar les conversions entre les unitats.
  - Ex.- es vol especificar que la durada d'una espera de tryLock és de 180 microsegons:

```
Lock lock = ...;
if ( lock.tryLock(180L, TimeUnit.MICROSECONDS) ) ...
```

Resultats d'aprenentatge de la Unitat Didàctica

- En finalitzar aquesta unitat, l'alumne ha de ser capaç de:
  - Identificar els inconvenients de les primitives bàsiques de Java.
  - Descriure les eines del package java.util.concurrent que faciliten el desenvolupament d'aplicacions concurrents:
    - Il·lustrar la utilització dels *locks* i les *condicions*. Comparar-los amb els monitors.
    - Interpretar ús de col·leccions concurrents *thread-safe* (ex. BlockingQueue)
    - Descriure el funcionament de les classes atòmiques. Comparar AtomicInteger amb els monitors.
    - Il·lustrar l'ús de semàfors (Semaphore) per a sincronització.
    - Il·lustrar el funcionament de les barreres, distingint entre CyclicBarrier i CountDownLatch.
    - Identificar les interfícies per a la gestió de la generació de fils, com la interfície Executor.
    - Identificar el mecanisme de temporització precisa que s'ofereix.