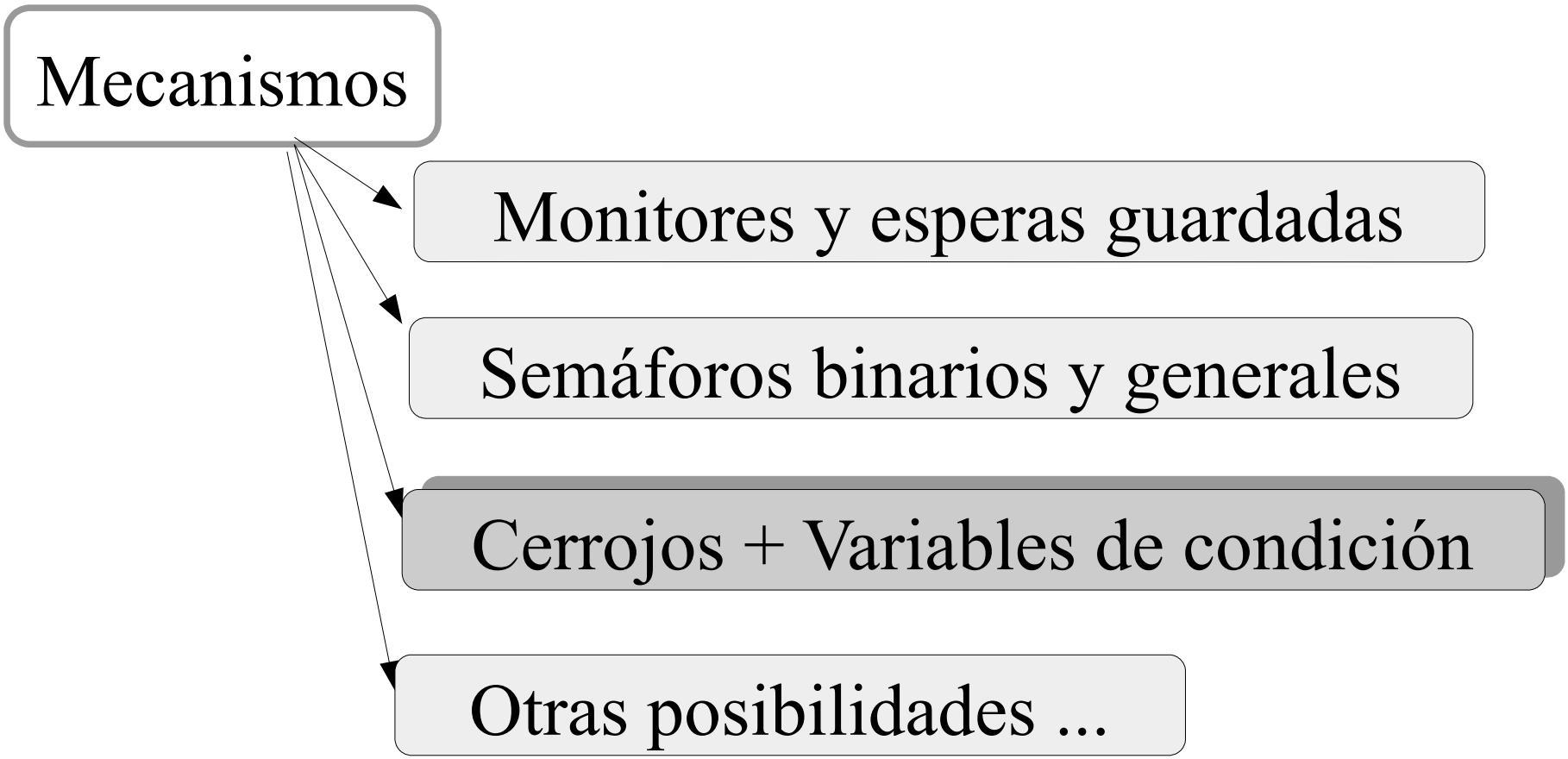
Programación Concurrente

Sincronización – Cerrojos - Barreras

Sincronización - Cerrojos



Exclusión - Sincronización

- Cada instancia de **Object** y sus subclases posee bandera de bloqueo **(lock)**
- Los tipos primitivos (no objetos) solo pueden bloquearse a través de los objetos que los encierran
- No pueden sincronizarse variables individuales
- Las variables pueden declararse como **“volatiles”**
- Los objetos arreglos cuyos elementos son tipos primitivos pueden bloquearse, pero sus elementos NO

Locks y semáforos

- **Son de bajo nivel**
  - **Primitivas independientes**
  - **Requieren que todos colaboren**

- **Locks - Sobre algo compartido**
  - **Variables de objeto**
  - **Variables de clase**
  - **objetos**
  - **clases**

Mecanismos de sincronización

- **Cerrojos o locks:** - Arquitecturas no estructuradas -
  - Sólo el hilo dueño de un cerrojo puede desbloquearlo
  - La readquisición de un cerrojo no está permitida
  - Primitivas ***init(), lock() y unlock().***
- **Semáforos**: - Arquitecturas no estructuradas - -
  - Bajo nivel de abstracción, de fácil comprensión y con una gran capacidad funcional.
  - Resultan peligrosos de manejar y son causa de muchos errores.
- **Bloques y métodos sincronizados -** Arquitecturas estructuradas -
  - Módulos de programación de alto nivel de abstracción
- **Monitores -** Arquitecturas estructuradas -
  - Módulos de programación de alto nivel de abstracción
  - Todos los métodos están sincronizados

Mecanismo del cerrojo

- Mecanismo simple basado en cerrojos

- Las operaciones de adquisición y liberación del cerrojo han de ser atómicas (su ejecución ha de producirse en una unidad de trabajo indivisible).

Lock, cómo se utiliza?

```
lock.lock();
    ... int valor = cc.getN(id);
     valor++;
     sleep(1000);
     cc.setN(id, valor);
lock.unlock();
```

```
try {
      lock.lock();
          //zona exclusiva
       }
finally {
       lock.unlock(); }
```

OJO:

- – hay que asegurarse de que se libera el cerrojo; por ejemplo, si hay excepciones

Locks intrínsecos y explícitos

- En Java 5.0, se introdujo un mecanismo de sincronización alternativo al lock intrínseco, que se define a través de la clase ReentrantLock (definida a través de la interfaz Lock).
- Limitaciones del lock intrínseco:
  - No es posible interrumpir un thread que espera un wait.
  - No es posible intentar de forma no bloqueante adquirir un lock sin suspenderse definitivamente en él.
  - Deben ser liberados en el mismo bloque de código en el cual se suspendió.
- Comparación:
  - El Lock intrínseco conduce a un estilo de programación sencillo, seguro y compatible con la gestión de excepciones
  - El ReentrantLock conduce a estrategias menos seguras, pero mas flexibles, proporciona mayor vivacidad y mejores características de respuesta.anismo de sincronización de hilos, es más flexible .

Interfaz Lock

- El método *lock()* bloquea la instancia si es posible. Si ya está bloqueada, el hilo que lo realizó se queda bloqueado hasta que el objeto se desbloquea.
- El método *lockInterruptibly()* trabaja de manera similar a menos que el hilo llamador al método sea interrumpido. Si un hilo se bloquea mediante este método y luego es interrumpido, retorna de esta llamada al mítedo.
- El método *tryLock()* intenta bloquear la instancia inmediatamente. Retorna true si fue exitosa, sino false. Este método nunca se bloquea
- El método *unlock()* destraba la instancia de la clase Lock.

Interfaz Lock

```
public interface Lock{
//Operaciones de locks usando métodos synchronized
void lock();                          // Adquiere el lock
void lockInterruptibly() throws InterruptedExcetion;
                              //lo adquiere a menos que se interrumpa el hilo
boolean tryLock();     // lo adquiere solo si está libre al invocar
boolean tryLock(long timeout,TimeUnit unit) throws InterruptedExcetion;
void unlock();              // libera el lock.
Condition newCondition();       //Retorna a instancia de condición
}
```

A diferencia del lock intrínseco o, la **interface Lock** ofrece diferentes formas de toma de un lock: incondicional, no bloqueante, temporizado o interrumpible.
Todas las operaciones de suspensión y liberación de un lock son explicitas

API *java.util.concurrent.locks*

- Proporciona clases para establecer sincronización de hilos alternativas a los bloques de código sincronizado y a los monitores
- Clases:
  - ***ReentrantLock:*** ,
  - ***LockSupport:*** ,
- Interfaz
  - ***Condition:***es una interface, cuyas instancias se usan asociadas a locks.
  - Implementan variables de condición y proporcionan una alternativa de sincronización a los métodos wait, notify y notifyAll de la clase Object. ,

Interfaz Lock

- El método *lock()* bloquea la instancia si es posible. Si ya está bloqueada, el hilo que lo realizó se queda bloqueado hasta que el objeto se desbloquea.
- El método *lockInterruptibly()* trabaja de manera similar a menos que el hilo llamador al método sea interrumpido. Si un hilo se bloquea mediante este método y luego es interrumpido, retorna de esta llamada al mítedo.
- El método *tryLock()* intenta bloquear la instancia inmediatamente. Retorna true si fue exitosa, sino false. Este método nunca se bloquea
- El método *unlock()* destraba la instancia de la clase Lock.

Objetos Condition

- Cuando se utiliza un *Lock explícito* para definir una región asíncrona, dentro de ella se utilizan los objetos Condition
- Los objetos Condicion se utilizan como mecanismo de sincronización entre threads.
  - Está estructuralmente ligado a un objeto Lock.
  - Sólo puede crearse invocando el método newCondition() sobre un objeto Lock.
  - Solo puede ser invocado por un thread que previamente haya tomado el Lock al que pertenece.

Interfaz Condition

También conocido como
colas de condición o *variables de condición*

- Hace que un hilo suspenda la ejecución hasta que otro hilo avise alguna condición de estado como cierta.
- El acceso al objeto protegido se produce en diferentes hilos
- El bloqueo está asociada con la Condición
- La propiedad que espera por una condición libera atómicamente el bloqueo asociado y suspende el subproceso actual

Interfaz Condition

Es ofrecida por variables de condición asociadas a un reentrantLock

- **throws InterruptedException**
  - void ***await()***

    //El hilo corriente espera hasta que es puesto en signal o interrumpido
  - long ***awaitNanos***(long nanos)

    //El hilo corriente espera hasta que es puesto en signal o interrumpido o se cumple ese tiempo
  - boolean ***await****(long time, TimeUnit unit)*
  - boolean ***awaitUntil****(Date deadline)*
- void ***awaitUninterruptibly()***

  //El hilo corriente espera hasta que es puesto en signal (no se interrumpe)
- void ***signal()*** *//Despierta un hilo*
- void ***signalAll()*** *//Despierta todos los hilos*

Ejemplo uso de objetos Condition

- Cuando se utiliza un Lock explícito para definir una región **asíncrona**, dentro de ella se utilizan los objetos **Condition** como mecanismo de sincronización entre threads.

```
class BufferDatos {
    Lock lock = new ReentrantLock();
    Condition lleno = lock.newCondition();
    Condition vacio = lock.newCondition();
```

- Un objeto Condition está estructuralmente ligado a un objeto Lock
  - Se crea mediante ***newCondition()*** sobre un objeto Lock.
- El objeto Condition solo puede ser invocado por un thread que previamente haya tomado el Lock al que pertenece.

Problema Prod/Cons -- objetos *Condition*

```
class BufferDatos {
    Lock lock = new ReentrantLock(true);
    Condition noLleno = lock.newCondition();
    Condition noVacio = lock.newCondition();
```

Los hilos productores quedarán bloqueados en espera de lugar en la cola de espera de la variable de condición “noLleno”

Los hilos consumidores quedarán bloqueados en espera de items para consumir en la cola de espera de la variable de condición “noVacio”

```
import java.util.concurrent.locks.*;
  class buffer_limitado {
      final Lock cerrojo = new ReentrantLock();  //declara y crea un cerrojo

// variables de condicion asociadas a "cerrojo" para control de buffer lleno y vacio
      final Condition noLleno = cerrojo.newCondition();
      final Condition noVacio = cerrojo.newCondition();

      //estructura elegida para mantener los items de datos
      final Object[] items = new Object[100];

       …  //otras variables necesarias

       public void poner(Object x) throws InterruptedException {
          cerrojo.lock();

          . . .
       . . .
```

```
import java.util.concurrent.locks.*;
 class buffer_limitado {
    final Lock cerrojo = new ReentrantLock();
    final Condition noLlena = cerrojo.newCondition();
    final Condition noVacia = cerrojo.newCondition();
    final Object[] items = new

    public void poner (Objec
       cerrojo.lock();
       …..
```

```
cerrojo.lock();
//duerme al hilo productor si buffer lleno  hasta
 //ser notificado
   try {while (lleno(items))
                    noLlena.await();
        }

       // se agrega el item x a la estructura

//se despierta a un hilo consumidor esperando
            noVacia.signal();
    } finally {cerrojo.unlock();}
}
```

Clase *ReentrantLock*

- Proporciona cerrojos de exclusión mutua de semántica equivalente a synchronized, pero más sencillo.
- Métodos:
  - ***lock()-unlock()***
  - ***isLocked()***
  - ***Condition newCondition()*** retorna una variable de condición asociada al cerrojo reentrantLock: ,
- Para controlar la exclusión mutua
  - *Definir el recurso compartido donde sea necesario*
  - *Definir un objeto c de clase* ***ReentrantLock*** *para control de la exclusión mutua.*
  - *Acotar la sección crítica con el par* ***c.lock() y c.unlock()***

Clase ReentrantLock (implementa interfaz Lock)

- Librerías que uiliza
  - import **java.util.concurrent.locks.Condition;**
  - import **java.util.concurrent.locks.Lock;**
  - import **java.util.concurrent.locks.ReentrantLock;**

- Creación de objetos ReentratLock
  - l = new ReentrantLock(true); //con política equitativa
  - l = new ReentrantLock(false); //política sin equidad

Exclusión Mutua con ReentrantLock

```
class Usuario {
private final ReentrantLock cerrojo = new ReentrantLock();
// ...
public void metodo() {
    cerrojo.lock();
try {
// ... cuerpo del metodo en e.m. }
finally {
    cerrojo.unlock()
}
}
}
```

Ejemplo Fumadores

```
clase DisparaSala
  //crea la sala
  //crea los 3 fumadores(id, sala)
  //comienza los 4 procesos
  ……
```

```
clase Fumador
  SalaFumador sala:
  int id;
  run(){
    while(true){
          sala.entrafumar(id);
          sala.terminafumar();
      } } }
```

```
clase Agente
  SalaFumador sala:
  run(){
     while(true){
      //nro Random entre 1 y 3
       sala.colocar (nro);
   } } }
```

```
clase SalaFumador;
  static int mesa= 0;
  static alguienFumando = false:
 public synchronized void entrafumar(int ingred){
       while(mesa != ingrediente || alguienFumando)
            wait();
      // se hace el cigarro
       mesa = 0; //mesa vacía
   alguienFumando = true;  }  //fuma
public synchronized void terminafumar()
       alguienFumando = false;
        notifyAll();}
 public synchronized void colocar(int noesta){
    while(mesa != 0 || alguienFumando)
          wait();
   mesa = noesta;
   notifyAll();}
```

Fumadores (Lock)

```
clase SalaFumadorLock
 static int mesa;
 static boolean alguienFuma;
 static Lock   l;
 static Condition [] puedoFumar;
 static Condition puedocolocar;

 public SalaFumadoresLock(){
  l = new ReentrantLock(true);
  puedofumar =  new Condition[3];
  puedofumar[0] = l.newCondition();
  ...  puedocolocar= l.newCondition();
  mesa = 0;
  alguienFuma = false;  }

public void entrafumar(int id){
  l.lock();   //usa un try-catch-finally
   while(mesa != id || alguienFuma){
     puedoFumar[id-1].await();
   mesa = 0;
   alguienFuma = true;
   l.unlock();  }
```

```
public void terminafumar(int id){
  l.lock();
  try { alguienFuma = false;
        puedocolocar.signal();
      } catch (InterruptedException e) {}
     finally{ l.unlock(); }
  }

  public void colocar (int ingrediente){
 //dentro de un try  - catch - finally
  l.lock();
     while( mesa !=0 || alguienFuma){
          puedocolocar.await();
     mesa = ingrediente;
     puedoFumar[mesa-1].signal();
   l.unlock();
  }
```

Bloques Locks vs Synchronized

- **Bloques synchronized:**
  - No garantizan la secuencia del acceso de que los hilos que esperan
  - No pueden pasar parámetros .
  - No puede hacerlo por un tiempo determinado (no tiene parámetros).
  - Debe estar contenido dentro de un método-
- **Bloques Lock**
  - Puede tener llamadas a lock() en un método y unlock() en otro separado

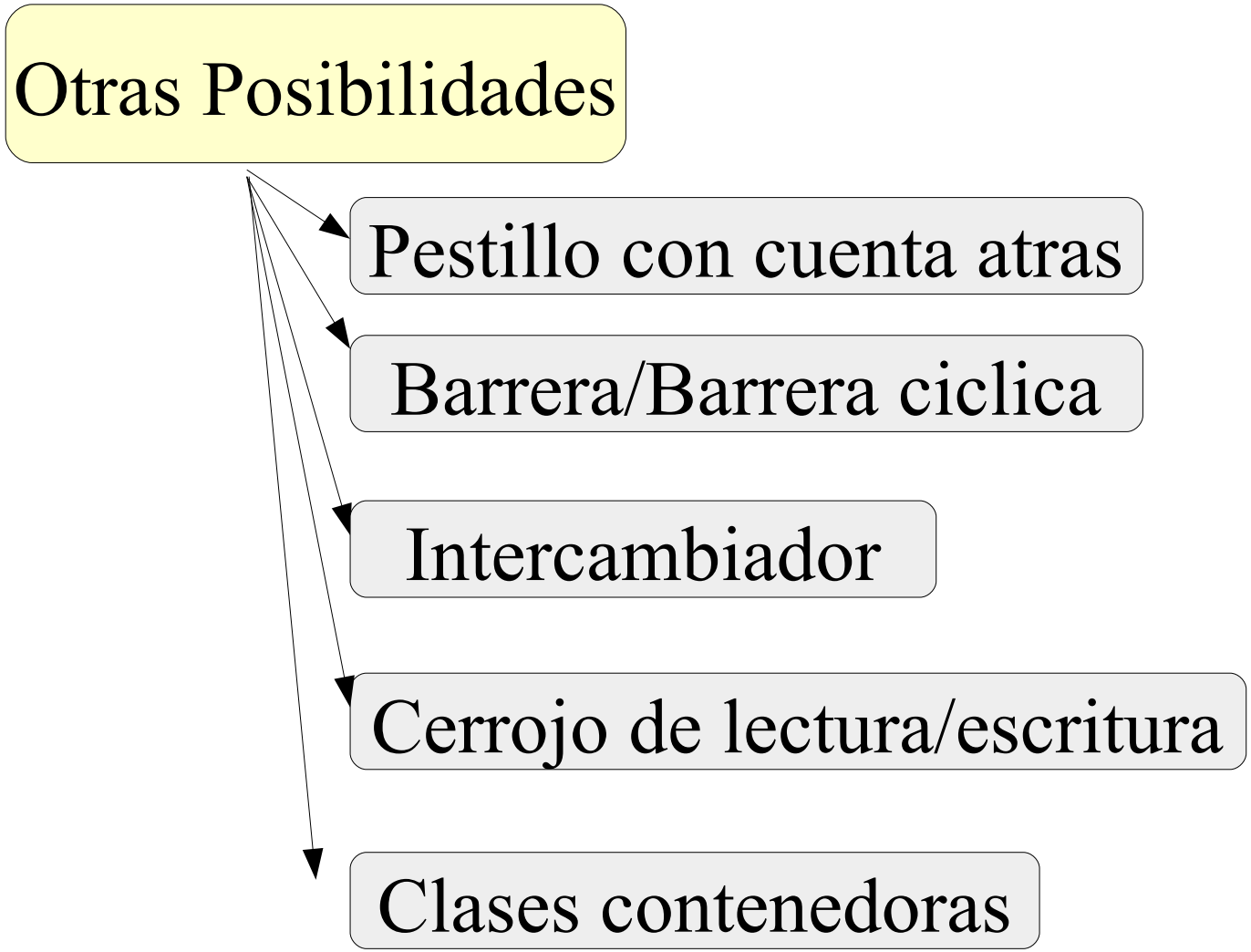
```
import java.util.concurrent.locks.*;
  class buffer_acotado {
      final Lock cerrojo = new ReentrantLock();  //declara y crea un cerrojo
      final Condition noLlena = cerrojo.newCondition();
                //obtiene variables de condicion asociadas a cerrojo
       final Condition noVacia = cerrojo.newCondition();
              //para control de buffer lleno
       final Object[] items = new Object[
            //array de objetos
int putptr, takeptr, cont; //punteros de i
public void put(Object x) throws Interr
          cerrojo.lock();
          …..
```

```
cerrojo.lock();
    try {while (cont == items.length)
       noLlena.await();
        //duerme a hilo productor si buffer lleno
        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
              ++cont;
        noVacia.signal();
                  //despierta al hilo consumidor
esperando
          } finally {cerrojo.unlock();}
}
```

Clase ReentrantLock (implementa interfaz Lock)

- Librerías que uiliza
  - import **java.util.concurrent.locks.Condition;**
  - import **java.util.concurrent.locks.Lock;**
  - import **java.util.concurrent.locks.ReentrantLock;**

- Creación de objetos ReentratLock
  - l = new ReentrantLock(true); //con política equitativa
  - l = new ReentrantLock(false); //con política aleatoria

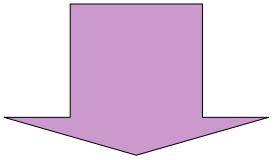


Otras facilidades para concurrencia

Sincronización de varios hilos

Esperar por un hilo

- Se puede utilizar el método ***join()***



```
// Arranque de hilo
hilo.start();
...
// Espera a que termine
hilo.join();
```

- Esperar por muchos hilos nos implicaría hacer muchas llamadas join(), una por cada hilo activo.
- Para que comiencen varios hilos a la vez o para esperar que todos ellos terminen ...

Pestillo con cuenta atrás

- Mecanismo de sincronización que permite que un conjunto de hilos esperen a un contador que debe llegar a 0
- Es un objeto que se inicializa con un contador N que podemos ir decrementando uno a uno.
- El N indica la cantidad de tareas que pondremos a esperar.
- Cuando el contador llega a cero, liberándose todas las tareas que esperan.
- Son útiles cuando un conjunto fijo de hilos deben esperarse para una tarea en común.
- Operaciones:
    - De espera
    - De decremento

Pestillo con cuenta atrás

- En Java se llaman CountDownLatch
- El objeto CountDownLatch libera los hilos al llegar el contador a 0.
- Operaciones:
  - await()
  - await(timeout, UNIT)
  - countDown()

import java.util.concurrent.CountDownLatch;

Pestillo con cuenta atrás

- Constructor

  CountDownLatch (n)

  n indica la cantidad de veces que debe decrementarse el contador (ejecutarse la operación “countDown”) para que los hilos es espera (por un “await”) puedan continuar

- Ejemplo:

  empezar: countDownLatch(1)

  terminar: countDownLatch(n)

Pestillo con cuenta atrás

Ejemplo:

- empezar: CountDownLatch(1)

  $hilo_1$ --------> empezar.await();

  $hilo_n$ --------> empezar.await();

  main --------> empezar.countDown();

- terminar: countDownLatch(n)

  $hilo_1$ --------> terminar.countDown();

  $hilo_n$ --------> terminar.countDown();

  main --------> terminar.await();

- Ver ejemploCountDownLatch.rar

Barreras

- Mecanismo de sincronización que permite que un conjunto de hilos esperen a llegar a un punto de barrera
- Una barrera de N posiciones retiene los primeros N-1 hilos que llegan. Cuando llega el enésimo, permite que salgan todos. La barrera se rompe y nuevos hilos pasan sin esperar.
- Tambien se puede trabajar con Barrera Ciclica

Barreras

- Mecanismo de sincronización que permite que un conjunto de hilos esperen a llegar a un punto de barrera
  - En Java se llaman CyclicBarrier
  - Soporta un comando run() por punto de corrida
  - El objeto barrera libera los hilos al llegar a la cantidad indicada en el constructor.
- Son útiles cuando un conjunto fijo de hilos deben esperarse para una tarea en común y deben sincronizarse repetidamente.
- La barrera es cíclica porque puede ser reutilizada después que los subprocesos en espera se liberan.
- Se puede considerar que un CountDownLatch es una barrera NO ciclica

CyclicBarrier

- Esta clase se instancia pasándole en el constructor cuántos hilos debe sincronizar.

```
//sincroniza 3 hilos
CyclicBarrier barrera = new CyclicBarrier(3);
```

- Los hilos deben llamar al método await() para esperar por los demas hilos que deben llegar a la barrera

```
public void run () {
  try {
//Se queda bloqueado hasta que los 3 hilos hagan esta llamada
     barrera.await();
  } catch (Exception e) { ...  }
  //código del hilo
}
```

import java.util.concurrent.CyclicBarrier;

Utilización de barreras

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

```
CyclicBarrier cb = new CyclicBarrier(n)
```

- **await() :** Espera a que todos los hilos definidos hayan entrado a la barrera
- **int await(long timeout, TimeUnit unit) :** Espera a que todos los hilos definidos hayan entrado a la barrera o que pase el tiempo estipulado
- **int getNumberWaiting() :** Retorna el número de hilos que están esperando en la barrera
- **int getParties():** Retorna el número de hilos que requiere esta barrera .
- **boolean isBroken() :** Pregunta si la barrera está quebrada

  es verdadero cuando uno o mas hilos rompen la barrera por interrupción o timeout, o por un reset, o la acción de la barrera falla debido a una excepción.
- **void reset() :** Restablece la barrera al estado inicial

Barrera

Constructores

```
CyclicBarrier(int cantHilosSincronizados)
```

```
CyclicBarrier
(int cantHilosSinc, Runnable accionesBarrera)
```

- Una barrera es un punto de espera a partir del cuál todos los hilos se sincronizan
  - Ningún hilo pasa la barrera hasta que todos los hilos esperados llegan a ella
  - ***Utilidad: sirve para***
    - Unificar resultados parciales
    - Como inicio a la siguiente fase de ejecución simultánea

Ejemplo con barrera

```
//código del Hilo
public class HiloPrueba implements Runnable
      CyclicBarrier barrera;
     HiloPrueba (CyclicBarrier bar){//constructor
           barrera = bar;
       }
     public void run() {
        try { barrera.await(); };
        catch (BrokenBarrierException e) {}
        catch (InterruptedException e) {}
     //codigo a ejecutar cuando se abre barrera…
       System.out.println("sigue ejecutando");
      }       //del run
} // de la clase
```

```
//programa principal
public void main(...) {
  int nHilos = n;
 //numero de hilo que abren barrera
 CyclicBarrier bar = new
                    CyclicBarrier (nHilos);
 new HiloPrueba(bar).start();
```

Intercambiador (exchanger)

- Actúa como un canal síncrono (buffer sin espacio), pero solo soporta un método, tipo *rendezvous*, que combina los efectos de *agregar* y *sacar* del buffer
- La operación, llamada "*exchange*" toma un argumento que representa el objeto ofrecido por un hilo a otro, y retorna el objeto ofrecido por el otro hilo

Dos agentes A y B se sincronizan en 1 punto. A le pasa un dato a B; B le pasa un dato a A.

Intercambiador

- Clase ***Exchanger*** en Java

import java.util.concurrent.Exchanger;

Clases *Contenedoras*

- Versiones sincronizadas de las principales clases contenedoras
- Disponibles en java.util.concurrent
- Clases:
  - ConcurrentHashMap, CopyOnWriteArrayList
  - ReadWriteLock
- Colas Sincronizadas – Interfaz BlockingQueue
    - *Clases:*
      - *LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue,y DelayQueue*