

# Control de la Concurrencia en Java (API Alto Nivel)

## Tema 5 - Programación Concurrente y de Tiempo Real

Antonio J. Tomeu<sup>1</sup>   Manuel Francisco<sup>2</sup>

<sup>1</sup>Departamento de Ingeniería Informática  
Universidad de Cádiz

<sup>2</sup>Alumno colaborador de la asignatura  
Universidad de Cádiz

PCTR, 2019

1. API Java 5 de Control de la Concurrency
2. Clases para Gestión de Variables Atómicas
3. Clase `java.concurrent.util.concurrent.Semaphore`
4. Clase `java.concurrent.util.concurrent.CyclicBarrier`
5. Clase `java.concurrent.util.concurrent.locks.*` y la interfaz `Condition`.
6. Colas sincronizadas.

- ▶ Disponible a partir de Java 5.
- ▶ Mejoras respecto a los mecanismos previos de control de la concurrencia.
  - ▶ Generar menos sobrecarga en tiempo de ejecución
  - ▶ Permiten controlar la e. m. y la sincronización a un nivel más fino
  - ▶ Soporta primitivos clásicos como los semáforos o las variables de condición en monitores
  - ▶ Ofrece cerrojos con mejor soporte a granularidad y al ámbito de uso.

# Qué Paquetes Usar

- ▶ Disponible en los paquetes
  - ▶ `java.util.concurrent`
  - ▶ `java.util.concurrent.atomic`
  - ▶ `java.util.concurrent.locks`
- ▶ Cierres para control de e. m.
- ▶ Semáforos, Barreras
- ▶ Colecciones (clases contenedoras) concurrentes de acceso sincronizado.
- ▶ Variables atómicas

- ▶ Conjunto de clases que soportan programación concurrente segura sobre variables simples tratadas de forma atómica
- ▶ Clases: `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray`, `AtomicReference<V>`, etc.
- ▶ OJO: No todos los métodos del API de estas clases soportan **concurrency segura**. Sólo aquellos en los que se indica que la operación se realiza de forma atómica.

## Código 1: codigos\_t5/prueba\_var\_atomic.java

```
1  import java.util.concurrent.atomic.*;
2  import java.util.concurrent.*;
3
4  class Hilo implements Runnable {
5      AtomicInteger cont; //Instancia compartida por todas las tareas
6      int valor;
7      public Hilo(AtomicInteger cont) {
8          this.cont = cont;
9      }
10     public void run() {
11         for (int i = 0; i < 100; i++) valor =
12             this.cont.incrementAndGet();
13     }
14 }
15
16 public class prueba_var_atomic {
17     public static void main(String[] args) throws Exception {
18         AtomicInteger cont = new AtomicInteger(0);
19         ThreadPoolExecutor miPool = new ThreadPoolExecutor(10, 10,
20             60000 L,
21             TimeUnit.MILLISECONDS, new LinkedBlockingQueue < Runnable >
22                 ());
```

```
21     miPool.prestartAllCoreThreads();
22     Hilo[] tareas = new Hilo[100];
23     for (int i = 0; i < 100; i++) {
24         tareas[i] = new Hilo(cont);
25         miPool.execute(tareas[i]);
26     }
27     miPool.shutdown();
28     Thread t = Thread.currentThread();
29     t.sleep(3000);
30     System.out.println(cont.get());
31
32 }
33 }
```

- ▶ Escribir una condición de carrera usando objetos AtomicLong y verificar que se preserve la e. m.



# La clase Semaphore

- ▶ Permite disponer de semáforos de conteo de semántica mínima o igual a la estándar de Dijkstra, o aumentada.
- ▶ Métodos principales:
  - ▶ `Semaphore(long permits)`
  - ▶ `acquire()`, `acquire(int permits)`
  - ▶ `release()`, `release(int permits)`
  - ▶ `tryAcquire()`; varias versiones
  - ▶ `availablePermits()`

## Código 2: codigos\_t5/Bloqueo\_Semaforo.java

```
1  import java.util.concurrent.*;
2
3  public class Bloqueo_Semaforo {
4
5      public static void main(String[] args)
6      throws InterruptedException {
7          Semaphore sem = new Semaphore(2);
8          sem.acquire(2);
9          System.out.println("Semaforo actualizado a valor 0...");
10         System.out.println("y su estado es: " + sem.toString());
11         System.out.println("Ahora intentamos adquirirlo...");
12         sem.tryAcquire();
13         System.out.println("sin bloqueo por no conseguirlo");
14         System.out.println("Ahora intentamos adquirirlo...");
15         sem.tryAcquire(3 L, TimeUnit.SECONDS);
16         System.out.println("tras esperar lo indicado sin
            conseguirlo...");
17         sem.acquire();
18         System.out.println("Aqui no llegaremos nunca...");
19
20     }
21 }
```

# Protocolo de Control de E. M. con Semáforos

```
1  // Thread A
2  public void run() {
3      try {
4          s.acquire();
5      } catch (InterruptedException ex) {}
6      //Bloque de codigo a ejecutar en e.m
7      s.release();
8  }
9  // Thread B
10 public void run() {
11     try {
12         s.acquire();
13     } catch (InterruptedException ex) {}
14     //Bloque de codigo a ejecutar en e.m
15     s.release();
16 }
17 //En el programa principal, hacer siempre
18 Semaphore s = new Semaphore(1);
```

# Ejemplo de Control de E.M con Semáforos I

## Código 3: codigos\_t5/Tarea\_concurrente.java

```
1  import java.util.concurrent.*;
2
3  public class Tarea_concurrente extends Thread {
4      Semaphore s;
5      public Tarea_concurrente(Semaphore param) {
6          s = param;
7      }
8
9      public void run() {
10         for (;;) {
11             try {
12                 s.acquire();
13             } catch (InterruptedException e) {}
14             System.out.println("Hilo " + this.getName() + " entrando a
15                             seccion critica");
16             s.release();
17             System.out.println("Hilo " + this.getName() + " saliendo de
18                             seccion critica");
19         }
20     }
21 }
```

- ▶ Descargue la clase `Tarea_concurente.java` y compile
- ▶ Descargue la clase `Protocolo_em_semaphore.java`, compile y ejecute
- ▶ Diseñe, utilizando semáforos, un código cuyos hilos se interbloqueen: `deadlockSem.java`

# Protocolo de Sincronización Inter-Hilos con Semáforos

```
1  // Thread A
2  public void esperarSenal(semaphore s) {
3      try {
4          s.acquire();
5      } catch (InterruptedException ex) {}
6  }
7  // Thread B
8  public void enviarSenal(Semaphore s) {
9      s.release();
10 }
11 //En el programa principal, hacer siempre
12 Semaphore s = new Semaphore(0);
```

#### Código 4: codigos\_t5/protocolo\_sincronizacion\_semaphore.java

```
1  import java.util.concurrent.*;
2
3  class HiloReceptor extends Thread {
4      Semaphore sem;
5      public HiloReceptor(Semaphore s) {
6          sem = s;
7      }
8      public void run() {
9          System.out.println("Hilo Receptor esta esperando la senal");
10         try {
11             sem.acquire();
12         } catch (InterruptedException e) {}
13         System.out.println("Hilo Receptor ha recibido la senal");
14     }
15 }
16
17
18 class HiloSenalador extends Thread {
19     Semaphore sem;
20     public HiloSenalador(Semaphore s) {
21         sem = s;
22     }
23 }
```

```
24     public void run() {
25         sem.release();
26         System.out.println("Hilo Senalador enviando senal...");
27     }
28 }
29
30 public class protocolo_sincronizacion_semaphore {
31
32     public static void main(String[] args) {
33         int v_inic_sem = 0;
34         Semaphore s = new Semaphore(v_inic_sem);
35
36         new HiloSenalador(s).start();
37         new HiloReceptor(s).start();
38     }
39 }
```



- ▶ Utilizando los semáforos que crea necesarios, provea la sincronización por turno cíclico de tres hilos diferentes. Nombre sus ficheros HA.java, HB.java, ...
- ▶ Implante ahora un lector-escritor con semáforos. Nombre sus ficheros LE1.java, LE2.java, ...

- ▶ Una barrera es un punto de espera a partir del cuál todos los hilos se sincronizan.
- ▶ Ningún hilo pasa por la barrera hasta que todos los hilos esperados llegan a ella
- ▶ Utilidad:
  - ▶ Unificar resultados parciales
  - ▶ Inicio siguiente fase de ejecución simultánea

# Protocolo de Barrera

```
1  // Codigo de Thread
2  public Hilo extends Thread {
3      public Hilo(CyclicBarrier bar) {...}
4      public void run() {
5          try {
6              int i = bar.await();
7          } catch (BrokenBarrierException e) {} catch
              (InterruptedException e) {}
8      //codigo a ejecutar cuando se abre barrera...
9  }
10 }
11 }
12 //En el programa principal, hacer siempre
13 int numHilos = n; //numero de hilo que abren barrera
14 CyclicBarrier Barrera = new CyclicBarrier(numHilos);
15 new Hilo(Barrera).start();
```

## Código 5: codigos\_t5/UsaBarreras.java

```
1  import java.util.concurrent.*;
2
3
4  class Hilo extends Thread {
5      CyclicBarrier barrera = null;
6
7      public Hilo(CyclicBarrier bar) {
8          barrera = bar;
9      }
10
11     public void run() {
12         try {
13             int i = barrera.await();
14         } catch (BrokenBarrierException e) {} catch
            (InterruptedException e) {}
15         System.out.println("El hilo " + this.toString() + " paso la
            barrera...");
16     }
17 }
18
19
20
21
```

```
22
23 public class UsaBarreras {
24     public static void main(String[] args) {
25         int numHilos = 3;
26         CyclicBarrier PasoANivel = new CyclicBarrier(numHilos);
27         new Hilo(PasoANivel).start();
28     }
29 }
```

- ▶ Un vector de 100 enteros es escalado mediante  $f(v[i]) = v[i]^2$
- ▶ Posteriormente se suman los elementos del vector. Escriba un programa en Java multihebrado (utilizando barreras) que realice la tarea: vectSum.java

- ▶ Proporciona clases para establecer sincronización de hilos alternativas a los bloques de código sincronizado y a los monitores.
- ▶ Clases e interfaces de interés:
  - ▶ `ReentrantLock`: proporciona cerrojos de e.m. de semántica equivalente a `synchronized`, pero con un manejo más sencillo y una **mejor granularidad**.
  - ▶ `LockSupport`: proporciona primitivas de bloqueo de hilos que permiten al programador diseñar sus clases de sincronización y cerrojos propios.
  - ▶ `Condition`: es una interfaz 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`.

# El API de la Clase ReentrantLock

- ▶ Proporciona **cerrojos reentrantes de semántica equivalente a synchronized**.
- ▶ Métodos `lock()`-`unlock()`
- ▶ Método `isLocked()`
- ▶ Método `Condition newCondition()` que retorna una variable de condición asociada al cerrojo.



- ▶ 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()`

# Protocolo de Control de E.M. con ReentrantLock II

```
1  class Usuario {
2      private final ReentrantLock cerrojo =
3          new ReentrantLock();
4      // ... public void metodo() {
5          cerrojo.lock();
6          try { // ... cuerpo del metodo en e.m. }
7              finally {
8                  cerrojo.unlock()
9              }
10     }
11 }
```

- ▶ Escribir un protocolo de e.m. con cerrojos ReentrantLock y guardarlo en `eMRL.java`. Escribir una clase que lo use.
- ▶ Descargue otra vez nuestra vieja clase `Cuenta_Banca.java`
- ▶ Decida qué código debe ser sincronizado, y sincronícelo con cerrojos ReentrantLock. La nueva clase será `Cuenta_Banca_RL.java`
- ▶ Escriba un programa multihebrado que use objetos de la clase anterior. Llámelo `Usa_Cuenta_Banca_Sync.java`

- ▶ Proporciona variables de condición
- ▶ Se usa **asociada a un cerrojo**: siendo cerrojo una instancia de la clase `ReentrantLock`
- ▶ `cerrojo.newCondition()` retorna la variable de condición.
- ▶ Operaciones principales: `await()`, `signal()` y `signalAll()`.

## Código 6: codigos\_t5/buffer\_acotado.java

```
1  import java.util.concurrent.locks.*;
2
3  class buffer_acotado {
4      final Lock cerrojo = new ReentrantLock();
5      final Condition noLlena = cerrojo.newCondition();
6      final Condition noVacía = cerrojo.newCondition();
7
8      final Object[] items = new Object[100];
9      int putptr, takeptr, cont;
10
11     public void put(Object x) throws InterruptedException {
12         cerrojo.lock();
13         try {
14             while (cont == items.length)
15                 noLlena.await();
16             items[putptr] = x;
17             if (++putptr == items.length) putptr = 0;
18             ++cont;
19             noVacía.signal();
20         } finally {
21             cerrojo.unlock();
22         }
23     }
```

```
24
25 public Object take() throws InterruptedException {
26     cerrojo.lock();
27     try {
28         while (cont == 0)
29             noVacia.await();
30         Object x = items[takeptr];
31         if (++takeptr == items.length) takeptr = 0;
32         --cont;
33         noLlena.signal();
34         return x;
35     } finally {
36         cerrojo.unlock();
37     }
38 }
39 }
```

- ▶ Descargue y compile la clase `buffer_acotado.java`
- ▶ ¿Qué diferencias tiene con la clase `Buffer.java` (carpeta del tema anterior)?
- ▶ Implemente ahora hilos productores y consumidores que usen el buffer acotado y láncelos. Llame a su código `prod_con_condition.java`

# Clases Contenedoras Sincronizadas

- ▶ A partir de Java 5 se incorporan versiones sincronizadas de las principales clases contenedoras.
- ▶ Disponibles en `java.util.concurrent`
- ▶ Clases: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList` y `CopyOnWriteArraySet`.



- ▶ A partir de Java 5 se incorporan diferentes versiones de colas, todas ellas sincronizadas.
- ▶ Disponibles en `java.util.concurrent`
- ▶ Clases: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue` y `DelayQueue`

- ▶ Son **seguras frente a hilos concurrentes**
- ▶ Proporcionan notificación a hilos según cambia su contenido
- ▶ Son **autosincronizadas**. Pueden proveer sincronización (además de e.m.) por sí mismas
- ▶ Facilitan enormemente la programación de patrones de concurrencia como el P-S

## Código 7: codigos\_t5/Productor.java

```
1  import java.util.concurrent.*;
2
3  public class Productor
4  implements Runnable {
5
6      LinkedBlockingQueue < Integer > data;
7      Thread hilo;
8
9      public Productor(LinkedBlockingQueue < Integer > l) {
10         this.data = l;
11         hilo = new Thread(this);
12         hilo.start();
13     }
14
15     public void run() {
16         try {
17             for (int x = 0;; x++) {
18                 data.put(new Integer(x));
19                 System.out.println("Insertando " + x);
20             }
21         } catch (InterruptedException e) {}
22     }
23 }
```

## Código 8: codigos\_t5/Consumidor.java

```
1  import java.util.concurrent.*;
2
3  public class Consumidor
4  implements Runnable {
5
6      LinkedBlockingQueue < Integer > data;
7      Thread hilo;
8
9      public Consumidor(LinkedBlockingQueue < Integer > l) {
10         this.data = l;
11         hilo = new Thread(this);
12         hilo.start();
13     }
14
15     public void run() {
16         try {
17             for (;;)
18                 System.out.println("Extrayendo " + data.take().intValue());
19         } catch (InterruptedException e) {}
20     }
21 }
```

- ▶ Descargue (subcarpeta colecciones seguras) `Productor.java`, `Consumidor.java` y `ProdConColaBloqueante.java`
- ▶ Ejecute. ¿Cómo explica el comportamiento observado?
- ▶ Modificaciones:
  - ▶ Aumente el número de ranuras de la cola
  - ▶ Active varios productores y un consumidor
  - ▶ Active un productor y varios consumidores

- ▶ Usamos colecciones a través de interfaces (usabilidad del código)
- ▶ Colecciones no sincronizadas no mejora mucho el rendimiento
- ▶ Problemas con esquema tipo productor-consumidor: use colas.
- ▶ Minimice la sincronización explícita
- ▶ Si con una colección retarda sus algoritmos, distribuya los datos y use varias

# En el Próximo Tema...

- ▶ Conceptos de Programación Distribuida
- ▶ El protocolo RMI
- ▶ La interfaz remote y la clase Naming
- ▶ *Stubs* y *Skeleton*
- ▶ Un DNS sencillo: rmiregistry
- ▶ Estructura de una aplicación distribuida
- ▶ Características enlazadas

# Bibliografía



Eckel, B.

*Thinking in Java*

Prentice Hall, 2006



Göetz et al.

*Java Concurrency in Practice*

2006



Oaks & Wong.

*Java Threads*

O'Reilly, 2004