Asignatura 780014
Programación Avanzada

Tema 5 –HERRAMIENTAS AVANZADAS DE COMUNICACIÓN SINCRONIZADA

Herramientas avanzadas de comunicación sincronizada

- Objetivo del tema:
  - Completar la revisión de mecanismos Java utilizados para coordinar la concurrencia. Estas herramientas son introducidas en la JSR 166 y resuelven casos específicos de coordinación, así como comportamientos generales

Índice

1. Variables Atomic
2. Colas y colecciones
3. Utilidades de sincronización
4. Ejemplos

Atomic Variables

- Las **variables atómicas** proporcionan:
  - Clases para el **manejo atómico** (indivisible) de variables simples
    - Tipos primitivos o referencias
  - Aritmética de **alto rendimiento** y métodos tipo '*compare-and-set*'
  - Programación '**lock-free**' segura sobre variables simples
  - Utilidad en la creación de aplicaciones que necesitan:
    - Comunicación
- Son útiles para:
  - Algoritmos concurrentes de alto rendimiento
  - Contadores y generadores de secuencias de números concurrentes
- Implementadas en el paquete java.util.concurrent.atomic

Atomic Variables

- Las operaciones se traducen en primitivas de hardware que **se ejecutan atómicamente**
- Ejemplos de operaciones de actualización atómicas:
  - `boolean compareAndSet(valorEsperado, nuevoValor);` //Pone atómicamente el *nuevoValor* si el valor actual coincide con el *valorEsperado*, devolviendo true si se cumple
  - `antiguoValor getAndSet(nuevoValor);` //Pone atómicamente el *nuevoValor* y devuelve el *antiguoValor* de la variable
  - En las clases numéricas, además:
    - `addAndGet`, `decrementAndGet`, `getAndAdd`, `getAndDecrement`, `getAndIncrement`, `incrementAndGet`

Atomic Variables

- Ejemplos de clases atomic:
  - `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, y `AtomicReference`
    - Permiten acceso y actualización a **variables** de los tipos relacionados
  - `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray`
    - Permiten acceso y actualización a **elementos** de un array de los tipos relacionados

Atomic Variables

- Ejemplo de secuencia (visto en temas anteriores) con variables atómicas

```
public class Main {
    public static void main(String[] args) {
        AtomicInteger sec = new AtomicInteger(0);
        for(int i=0; i<100; i++) {
            Hilo hilo=new Hilo(i, sec);
            hilo.start();
        }
    }
}
```

```
public class Hilo extends Thread {
    private AtomicInteger sec;
    private int id;
    Hilo(int id, AtomicInteger sec) {
        this.id=id;
        this.sec=sec;
    }
    public void run() {
        System.out.println("Hilo "+id+":
    secuencia="+sec.incrementAndGet());
    }
}
```

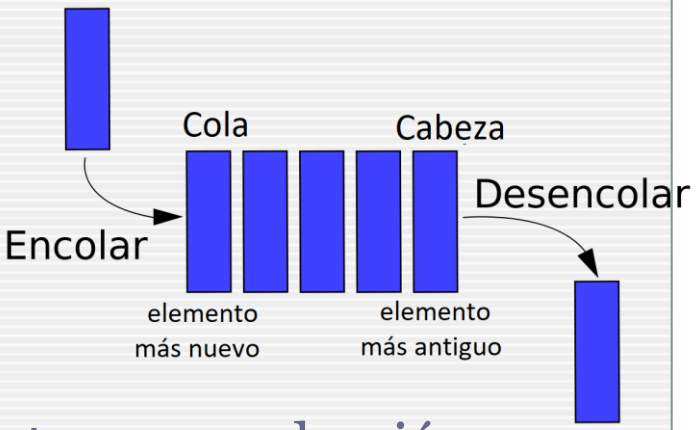
```
run:
Hilo 1: secuencia=1
Hilo 2: secuencia=3
Hilo 0: secuencia=2
Hilo 3: secuencia=4
...
Hilo 96: secuencia=98
Hilo 97: secuencia=99
Hilo 98: secuencia=100
BUILD SUCCESSFUL (total time: 0 seconds)
```

Colas y Colecciones

- JSR166 introdujo dos contenedores de objetos que garantizan su **acceso en exclusión mutua** y pueden usarse para implementar la concurrencia:
  - Colas
  - Colecciones

Colas concurrentes

- La clase **ConcurrentLinkedQueue**
  - Proporciona una cola FIFO:
    - **Sin límite de elementos**
    - Escalable
    - Segura
    - **No bloqueante**



  - Apropiada cuando muchos hilos comparten una colección
  - Métodos:
    - `offer(E e);` //Añade el elemento *e* al final de la cola
    - `contains(E e);` //Dice si el elemento *e* está en la cola
    - `poll();` //Devuelve el primer elemento y lo borra (o null, si vacía)
    - `peek();` //Devuelve el primer elemento sin borrarlo (o null, si vacía)

Colas concurrentes

- Ejemplo productor-consumidor con ConcurrentLinkedQueue

```
public class Main {
    public static void main(String[] args) {
        Queue<String> cola = new ConcurrentLinkedQueue<String>();
        Productor p = new Productor(cola);
        Consumidor c = new Consumidor(cola);

        Thread t1 = new Thread(p);
        Thread t2 = new Thread(c);

        t1.start();
        t2.start();
    }
}
```

Colas concurrentes

```
public class Productor implements Runnable {
    private static int i;
    private final Queue<String> cola;
    private final Random r;
    public Productor(Queue<String> cola) {
         this.cola = cola;   r = new Random();
    }
    public void run() {
         while (true) {
             String m = "Mensaje numero: " + (i++);
             cola.offer(m); //Añade el elemento
             synchronized (cola) {
                 cola.notifyAll();
             }
             System.out.println("Productor: " + m);
             try {
                Thread.sleep(r.nextInt(5000));
             } catch (InterruptedException ex) {}
        }
    }
}
```

```
public class Consumidor implements Runnable {
   private final Queue<String> cola;
   public Consumidor(Queue<String> cola) {
       this.cola = cola;
   }
   public void run() {
      while (true) {
          while (!cola.isEmpty()) { //Mientras haya elementos
             String m = cola.poll(); //Saca el 1º elemento y lo borra
             if (m != null) {
                 System.out.println("Consumidor: " + m);
             }
          }
          try { //Aquí la cola ya está vacía
             synchronized (cola) {
                cola.wait(); //Espera hasta que llegue un elemento
             }
          } catch (InterruptedException ex) {}
      }
   }
}
```

Colas concurrentes

- Interfaz **BlockingQueue**
  - Define 5 versiones de colas **bloqueantes** donde meter y sacar elementos:
    - LinkedBlockingQueue
    - ArrayBlockingQueue
    - SynchronousQueue
    - PriorityBlockingQueue
    - DelayQueue
  - Permiten esperas por cola vacía y cola llena (cuando tienen límite de elementos)
  - Sirven para resolver muchos problemas de concurrencia de intercambios de mensajes

Colas concurrentes

- LinkedBlockingQueue
  - Capacidad **limitada opcionalmente**
  - Basadas en **nodos enlazados**
  - Se provocan **bloqueos** por cola llena (si hay un máximo) y cola vacía
- ArrayBlockingQueue
  - Capacidad **limitada** (una vez creada, no se puede cambiar)
  - Están construidas sobre **arrays**
  - Se provocan **bloqueos** por cola llena y cola vacía
- SynchronousQueue
  - **Sin capacidad** (cero elementos)
  - Cada inserción espera por una extracción, y viceversa
  - Hay operaciones habituales que no se pueden hacer: `peek()`, no se puede iterar, no se puede ver si existe un elemento en la cola (`contains`), etc.

Colas concurrentes

- PriorityBlockingQueue
  - **No tiene límite** de elementos (capacidad ilimitada)
  - Ordena los elementos en su **orden natural** (comparación)
  - Se provocan **bloqueos por cola vacía** (por cola llena no, porque no hay límite)
- DelayQueue
  - **No tiene límite** de elementos
  - Los elementos que contiene deben extender la clase Delayed
  - Cada elemento **sólo puede ser extraído** después de **transcurrido su tiempo de retardo**
  - Se provocan **bloqueos por cola vacía o cuando no hay elementos extraíbles** (es decir, aquellos para los cuales haya expirado el tiempo de retardo). Por cola llena no hay bloqueos, porque no hay límite de tamaño

Colas concurrentes

- Ejemplo productor-consumidor con **LinkedBlockingQueue**

```
public class Main {
     public static void main(String args[])
     {
          BlockingQueue q = new LinkedBlockingQueue(10); //Se puede usar otro tipo de cola
          Productor p = new Productor(q);
          Consumidor c1 = new Consumidor(q);
          Consumidor c2 = new Consumidor(q);
          new Thread(p).start();
          new Thread(c1).start();
          new Thread(c2).start();
     }
}
```

BlockingQueue puede ser utilizada de forma segura con varios productores y consumidores

Colas concurrentes

```
public class Productor implements Runnable {
    private final BlockingQueue cola;
    private int i;

    public Productor(BlockingQueue c) {
        cola = c;
        i=0;
    }

    public void run() {
        try {
            while (true) {
                String elem="elem"+(i++);
                System.out.println("Genero "+elem);
                cola.put(elem); //Si cola llena,
                                //bloquea al hilo
            }
        } catch (InterruptedException ex) {}
    }
}
```

```
public class Consumidor implements Runnable {
    private final BlockingQueue cola;

    public Consumidor(BlockingQueue c) {
        cola = c;
    }

    public void run() {
        try {
            while (true) {
                System.out.println("Extraído: "+cola.take()); //Si
                                        //cola vacía, bloquea al hilo
            }
        } catch (InterruptedException ex) {}
    }
}
```

Colecciones sincronizadas

- La clase **Collections** ofrece métodos para obtener versiones **bloqueantes** de las colecciones habituales (en inglés, *synchronized collections*)
  - Por ejemplo:
    - `Map  m = Collections.synchronizedMap(new HashMap());`
    - `List l = Collections.synchronizedList(new ArrayList());`
  - Son thread-safe: sus métodos proporcionan exclusión mutua **bloqueando toda la colección**
    - Esto significa poca vivacidad (ineficiencia)
    - **Si hay muchos hilos no es una solución válida**: necesitamos colecciones con bloqueos de menor granularidad

Colecciones concurrentes

- Java implementa **colecciones concurrentes** (en inglés, *concurrent collections*):
  - Permiten que **varios hilos accedan a la vez** a la colección (no bloquean toda la colección de una vez)
  - Son thread-safe
  - Ejemplos (1/2):
    - **ConcurrentHashMap**
      - Es una tabla Hash en la que las operaciones *get*, *put* y *remove* pueden tener lugar concurrentemente
      - El resultado es una **concurrencia mucho mayor** cuando múltiples hilos necesitan acceder al mismo Map
      - Proporciona métodos de operaciones compuestas seguras (p.e. *putIfAbsent*)

Colecciones concurrentes

- Ejemplos (2/2):
  - **CopyOnWriteArrayList**
    - Es una **variante concurrente de ArrayList**
    - Internamente: crea una copia del array cada vez que se añade o elimina un elemento, pero las iteraciones que se están realizando siguen trabajando con la copia que existía cuando se creó el iterador
  - **CopyOnWriteArraySet**
    - **Versión concurrente de Set**
    - Ídem: crea una copia del objeto Set cada vez que se modifica

Utilidades de sincronización

- Utilidades/herramientas de sincronización (*synchronizers*)
  - **Coordinan y controlan el flujo** de ejecución de uno o más hilos
  - Java 1.5 incluyó en el JSR 166 (java.util.concurrent):
    - `public interface Lock + Condition` //estudiados en tema 3
    - `public class Semaphore extends Object` //estudiados en tema 4
    - `public class CyclicBarrier extends Object`
    - `public class CountDownLatch extends Object`
    - `public class Exchanger<V> extends Object`
  - Cada una de estas herramientas tiene **métodos** que los hilos pueden llamar, y que los **bloquearán o no**, dependiendo del estado y de las reglas de la herramienta

Utilidades de sincronización

- Las 3 nuevas herramientas:
  - Proporcionan buenas soluciones a **problemas de sincronización** comunes, de **propósito especial**
  - Proporcionan mejores formas de pensar en los diseños
    - ¡Pero peores si no se aplican de la manera más "natural"!
  - Sería complicado o tedioso tener que escribir su código nosotros mismos
    - Ahorran mucho tiempo de codificación
    - Favorecen la reutilización

Sincronización con Barrera Cíclica

- public class **CyclicBarrier** extends Object
  - Permite a un conjunto de hilos **esperarse entre ellos** en un determinado punto (barrera) hasta alcanzar un quorum
    - Ejemplo: jugadores que quedan en un campo de fútbol y tienen que esperar hasta que son 22 y pueden jugar
  - La barrera se crea con un valor *n* que determina el número de hilos que tendrá que esperar (quorum)
    - Cuando lleguen todos, la barrera se levanta y los hilos prosiguen
  - Se llaman "**cíclicas**" porque se pueden **reutilizar** después de que los hilos que esperaban hayan sido liberados
    - La barrera se vuelve a inicializar automáticamente al valor *n*
  - Puede (o no) tener asociado un objeto Runnable que arranca **después de** llegar **el último** hilo, pero **antes de liberarlos**

Sincronización con Barrera Cíclica

- Constructores:
  - `CyclicBarrier(int numHilos)` //Nº de hilos que esperarán
  - `CyclicBarrier(int numHilos, Runnable accion)` //Incluye, además, el hilo que se lanzará justo antes de levantar la barrera

- Métodos principales:
  - `int await()` //**Espera** hasta que todos los hilos han ejecutado //*await* sobre la barrera. Devuelve el nº de los que faltan
  - `int getParties()` //Devuelve el número de hilos que se //requieren para traspasar la barrera
  - `int getNumberWaiting()` //Devuelve el número de hilos //que esperan en la barrera

Sincronización con Barrera Cíclica

- La barrera se "levanta" **automáticamente** cuando llega el último hilo
  - Ningún hilo tiene que despertar a los demás
    - No existen métodos en la clase para hacerlo
- OJO: si menos hilos de los esperados hacen *await()*, todos ellos quedarán **bloqueados** indefinidamente
  - No podemos despertarlos, es la barrera la que lo hace
- Se suelen utilizar para que:
  - Unos hilos comiencen a la vez
  - Unos hilos se esperen para terminar (o en un punto determinado para continuar)

Sincronización con Barrera Cíclica

- Ejemplo: 100 hilos comienzan a la vez, se ejecutan y se esperan cuando terminan

```
public class Main {
    public static void main(String[] args) {
        int numeroHilos = 100;
        HiloFinal hiloFinal = new HiloFinal(); //Se ejecutará cuando lleguen todos los hilos a barreraInicio,
                                               //pero antes de levantarse la barrera
        final CyclicBarrier barreraInicio =
                        new CyclicBarrier(numeroHilos + 1, hiloFinal); //+1 para que el hilo principal también espere
        final CyclicBarrier barreraFin = new CyclicBarrier(numeroHilos + 1); //+1 para el hilo principal
        for (int i = 0; i < numeroHilos; i++) {
            Hilo hilo=new Hilo(i, barreraInicio, barreraFin);
            hilo.start();
        }
        try {
            System.out.println("levanto barrera");
            barreraInicio.await();
            barreraFin.await();
            System.out.println("todo acabado");
        } catch (Exception e) { }
    }
}
```

Sincronización con Barrera Cíclica

```
public class Hilo extends Thread {

    private CyclicBarrier barreraInicio;
    private CyclicBarrier barreraFin;
    private int id;

    public Hilo(int id, CyclicBarrier inicio, CyclicBarrier fin) {
        this.id=id;
        barreraInicio=inicio;
        barreraFin=fin;
    }

    @Override
    public void run() {
        try {
            barreraInicio.await();
            System.out.println("hilo "+id+" ejecutándose");
            barreraFin.await();
        } catch (Exception e) { }
    }
}
```

```
public class HiloFinal extends Thread {
    @Override
    public void run() {
        try {
            System.out.println("Se lanza este
hilo cuando llegan todos a la barrera, antes de
levantarse ésta");
        } catch (Exception e) { }
    }
}
```

```
run:
levanto barrera
Se lanza este hilo cuando llegan todos a la
      barrera, antes de levantarse ésta
hilo 0 ejecutándose
hilo 3 ejecutándose
hilo 6 ejecutándose
hilo 2 ejecutándose
hilo 7 ejecutándose
…
hilo 86 ejecutándose
hilo 83 ejecutándose
todo acabado
BUILD SUCCESSFUL (total time: 0 seconds)
```

Sincronización con CountDownLatch

- public class **CountDownLatch** extends Object
  - Permite a un conjunto de hilos **esperar a** que **otro** conjunto **indique** que ha finalizado sus tareas para continuar
    - Suele haber distintos tipos de hilos
  - **No se puede reutilizar**
  - Se inicializa con un **contador** de valor *n*
  - Separa la llegada de los hilos de la espera:
    - Los hilos **esperan** continuar llamando al método *await()*
    - El contador se **decrementa** cuando un hilo ejecuta *countDown()*, pero no los bloquea
  - **Cuando el contador llega a 0**, todos **los hilos** que estaban esperando, **son liberados**
    - Las siguientes llamadas a *await()* no bloquearán a los hilos

Sincronización con CountDownLatch

- Útil cuando el problema se divide en diferentes partes:
  - Los hilos que realizan las tareas hacen *countDown()* cuando terminan
  - Los hilos que coordinan hacen *await()* para esperarlos
- Constructor:
  - `CountDownLatch(int n)` //n = valor inicial del contador
- Métodos principales:
  - `void await()` //**Espera** hasta que el contador valga 0,
    //o hasta que el hilo sea interrumpido
  - `void countDown()` //**Decrementa** el contador y, si éste llega a 0,
    //**libera** a todos los hilos que estuvieran esperando
  - `long getCount()` //Devuelve el valor actual del contador

Sincronización con CountDownLatch

- Ejemplo: 2 hilos de tipo2 esperan a 5 hilos de tipo1

```
public class Main {
    public static void main(String[] args) {
        CountDownLatch cl=new CountDownLatch(5); //Habrá que esperar a 5 hilos
        for(int i=0; i<5; i++) //Creamos y ponemos en ejecución 5 hilos de tipo1
        {
            HiloTipo1 h1=new HiloTipo1(i, cl);
            h1.start();
        }
        for(int i=0; i<2; i++) //Creamos y ponemos en ejecución 2 hilos de tipo2
        {
            HiloTipo2 h2=new HiloTipo2(i, cl);
            h2.start();
        }
    }
}
```

Sincronización con CountDownLatch

```
public class HiloTipo1 extends Thread {

    private int id;
    private CountDownLatch cl;

    public HiloTipo1(int i, CountDownLatch cl) {
        id=i;
        this.cl=cl;
    }

    public void run() {
        try { //Hace la acción que tenga que hacer:
            sleep((long)(1000+(200)*Math.random()));
        } catch (InterruptedException ex) { }
        cl.countDown(); //Cuando termina, decrementa el contador
        System.out.println("Valor contador: " + cl.getCount());
    }
}
```

```
public class HiloTipo2 extends Thread {

    private int id;
    private CountDownLatch cl;

    public HiloTipo2(int i, CountDownLatch cl) {
        id=i;
        this.cl=cl;
    }

    public void run() {
        try {
            cl.await(); //Espera a que lleguen los tipo1
        } catch (InterruptedException ex) { }
        System.out.println("Hilo tipo2 nº "+id+" sigue");
    }
}
```

```
run:
Valor contador: 4
Valor contador: 3
Valor contador: 2
Valor contador: 1
Valor contador: 0
Hilo tipo2 nº 1 sigue
Hilo tipo2 nº 0 sigue
BUILD SUCCESSFUL (total time: 0 seconds)
```

Sincronización con Exchanger

- public class **Exchanger**<V> extends Object
  - Permite comunicación sincronizada:
    - Punto de sincronización para **dos hilos**
    - Pueden intercambiarse un objeto (comunicación)
  - Constructor:
    - `Exchanger()` //Crea un objeto Exchanger
  - Métodos:
    - `V exchange(V x)` //Cada hilo ofrece un objeto con *exchange* y //recibe, a su vez, el objeto ofrecido por el otro hilo de la misma //manera
    - `V exchange(V x, long timeout, TimeUnit unit)` //Ídem, //pero con tiempo límite para la espera

Sincronización con Exchanger

- Ejemplo: productor-consumidor con dos buffers. Cuando el productor llena el suyo, se lo cambia al consumidor, que lo acepta cuando el suyo está vacío

```
public class Main {
    public static void main(String[] args) {
        final int capacidad=10;
        ArrayList<String> buffCons = new ArrayList<String>(capacidad);
        ArrayList<String> buffProd = new ArrayList<String>(capacidad);
        Exchanger<ArrayList> exch = new Exchanger();
        Productor p=new Productor(buffProd, exch, capacidad);
        Consumidor c=new Consumidor(buffCons, exch);
        p.start();
        c.start();
     }
}
```

Sincronización con Exchanger

```
public class Productor extends Thread {
    private final int capacidad=10;
    private Exchanger<ArrayList> exch;
    private ArrayList<String> bufferCons;

    public Productor(ArrayList<String> buff, Exchanger ex, int capacidad) {
        bufferCons=buff;
        exch=ex;
        this.capacidad=capacidad;
    }

    public void run() {
        ArrayList<String> buffer = bufferCons;
        int i=0;
        try {
            while (buffer != null) {
                buffer.add("Elemento "+i); //Inserta el elemento en el buffer
                if (buffer.size()==capacidad) { //Si está lleno
                    buffer = exch.exchange(buffer); //Se intercambia el buffer con el consumidor
                    System.out.println("Productor ha intercambiado. Nº elementos en el buffer ahora: "+buffer.size());
                }
                i++;
            }
        } catch (InterruptedException ex) { }
    }
}
```

Sincronización con Exchanger

```
public class Consumidor extends Thread {
    Exchanger<ArrayList> exch;
    ArrayList<String> bufferProd;

    public Consumidor(ArrayList buff, Exchanger ex) {
        bufferProd=buff;
        exch=ex;
    }

    public void run() {
        ArrayList<String> buffer = bufferProd;
        try {
            while (buffer != null) {
                if(buffer.size()>0)
                    buffer.remove(0); //Saca elemento del buffer
                if (buffer.isEmpty()) { //Cuando el buffer está vacío, está listo para intercambiar
                    buffer = exch.exchange(buffer); //Intercambia el buffer con el productor
                    System.out.println("Consumidor ha intercambiado. Nº elementos en el buffer ahora: "+buffer.size());
                }
            }
        } catch (InterruptedException ex) { }
   }
}
```

Sincronización con Exchanger

- Posible resultado de la ejecución:

```
run:
Productor ha intercambiado. Nº elementos en el buffer ahora: 0
Consumidor ha intercambiado. Nº elementos en el buffer ahora: 10
Productor ha intercambiado. Nº elementos en el buffer ahora: 0
Consumidor ha intercambiado. Nº elementos en el buffer ahora: 10
Consumidor ha intercambiado. Nº elementos en el buffer ahora: 10
Productor ha intercambiado. Nº elementos en el buffer ahora: 0
Productor ha intercambiado. Nº elementos en el buffer ahora: 0
Consumidor ha intercambiado. Nº elementos en el buffer ahora: 10
...
BUILD STOPPED (total time: 1 second)
```

Ejercicios (1/2)

- 1.- Ejecutar los códigos utilizados en esta presentación.
- 2.- Crear un programa utilizando Atomic variables que calcule la suma de los números primos entre 1 y 10.000.000
- 3.- Crear un programa que haga lo mismo que el ejercicio 2, pero con CyclicBarrier. Para ello:
  - Utilizar 5 hilos que sumen intervalos de 2.000.000 cada uno
  - Los hilos no acaban hasta que todos hayan hecho sus sumas
  - Se esperan hasta que todos hayan terminado en una CyclicBarrier
  - Habrá un hilo asociado que haga la suma de los cinco resultados parciales

Ejercicios (2/2)

- 4.- Implementar el mismo programa que en el ejercicio 1, pero con CountDownLatch

- 5.- Implementar un programa con Exchanger en el que un hilo le pase a otro un mensaje para que lo imprima por pantalla, y viceversa