

# Basic Thread Synchronization

**Programación Concurrente  
2017**

**Ing. Ventre, Luis O.**

# Synchronized con un objeto: Detalles

- En la clase anterior, se vió un ejercicio donde el acceso a la venta de tickets de dos salas se sincronizaba a través de la primitiva `synchronized()`; enviando un `object` como llave.
- Veremos a continuación los detalles respecto a posibles errores al enviar diferentes objetos como llaves.
- <https://www.securecoding.cert.org/confluence/display/java/LCK01-J.+Do+not+synchronize+on+objects+that+may+be+reused>

# Synchronized con un objeto: Detalles

- La confusión puede surgir del concepto que casi todos los objetos en java derivan de la clase “object”, por lo cual podrían ser utilizados como “key” para sincronizar un bloque de código.
- Es importante mencionar que en java como en otros lenguajes orientados a objetos un objetivo es optimizar el uso de memoria, la creación por ej. de un **Integer**, no es más que un wrapper, apuntador o manipulador del objeto real al que apunta.

# Synchronized con un objeto: Detalles

- Cuando se sincroniza con un WRAPPER de un tipo primitivo por ejemplo un INTEGER, en realidad se utiliza el KEY (semáforo binario) del objeto valor que está apuntando.
- Cuando en el interior del bloque de código se cambia el valor de ese WRAPPER, se utiliza el key del nuevo valor apuntado.
- Puntualmente en el caso del ejemplo del cine, si se hace la sincronización con los dos valores INTEGER de vacancias, al modificar internamente el valor de vacancias se puede dar (en 1000 ejecuciones 10 veces aprox por ej) que queden ambas llaves apuntando al mismo valor, por lo que pueden acceder ambos hilos al mismo bloque de código y tenemos inconsistencia en los resultados en algunas corridas.

# Synchronized con un objeto: Detalles

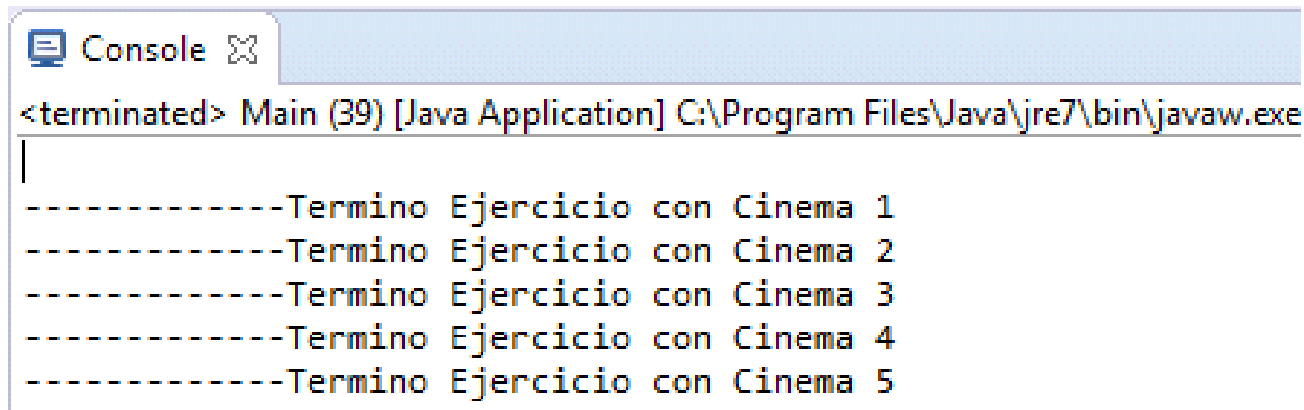
- Veremos los siguientes casos de sincronización:
  - Caso 1  
Del libro donde la sincronización se realiza sobre 2 **objects** creados para tal fin. Resultado OK.
  - Caso 2  
Consultado por alumno en clases, donde se sincroniza con la misma variable VACANCIES (**Integer**) enviada al metodo sincronizado para operar. Resultado ERRONEO.
  - Caso 3  
Donde se sincroniza con un **arreglo** de elementos, utilizando el arreglo como objeto llave. Resultado OK.
  - Caso 4  
Donde se sincroniza con un **elemento del arreglo** del caso anterior. Resultado ERRONEO.
  - Caso 5  
Donde se sincroniza con un **object nuevo**, donde internamente tiene un atributo y metodos para las operaciones. Resultado OK.

# Synchronized con un objeto: Detalles

- Se pueden observar los detalles de la implementación en Eclipse en el proyecto SynchronizedObject.
  - El mismo tiene la clase main, donde se crea un arreglo de interfaces y se le asigna un objeto de cada tipo de los 5 casos enumerados anteriormente.
  - Luego con 2 bucles se ejecutan las veces seleccionadas las corridas para cada caso de los 5 descriptos.
  - En el bucle interno se analiza que los resultados sean correctos, ante una inconsistencia se imprime NO ANDA, en la consola.

# Synchronized con un objeto: Detalles

- A continuación se observa el resultado de ejecutar 10 veces cada grupo de instrucciones de los tickets office...como puede observarse...todo parece funcionar OK!



```
Console [X]
<terminated> Main (39) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
|
-----Termino Ejercicio con Cinema 1
-----Termino Ejercicio con Cinema 2
-----Termino Ejercicio con Cinema 3
-----Termino Ejercicio con Cinema 4
-----Termino Ejercicio con Cinema 5
```

# Synchronized con un objeto: Detalles

- A continuación se observa el resultado de ejecutar 100000 veces cada grupo de instrucciones de los tickets office...  
Cuidado...el testing puede engañarnos...

```
Console X
<terminated> Main (39) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

-----Termino Ejercicio con Cinema 1
NO ANDA
NO ANDA
NO ANDA
NO ANDA
NO ANDA
NO ANDA
NO ANDA
NO ANDA
NO ANDA

-----Termino Ejercicio con Cinema 2
-----Termino Ejercicio con Cinema 3
NO ANDA

-----Termino Ejercicio con Cinema 4
-----Termino Ejercicio con Cinema 5
```



# Using conditions in synchronized code

- En el siguiente ejercicio se comprenderá el uso de primitivas para sincronización wait, notify y notifyAll.
- Problema Productor-consumidor.
  - Existe un buffer de “datos”.
  - Uno o más productores que escriben el buffer.
  - Uno o más consumidores que retiran los datos del buffer.
- Buffer – estructura compartida. ***Debe controlarse el acceso usando mecanismos de sincronización.***
- Pero hay una LIMITACIÓN más. Tamaño del buffer. Condiciones de lleno o vacío limitan el acceso.

# Using conditions in synchronized code

- Java implementa los siguientes métodos para este tipo de situación:
  - wait()
  - notify()
  - notifyAll()Estos métodos pertenecen a la clase Object.
- Un hilo puede llamar al metodo wait(),  
“adentro de un bloque synchronized. “
- Cuando esto sucede la JVM pone a dormir a ese hilo y **libera el objeto llave utilizado en el synchronized.**

Para despertar un hilo debe utilizarse el método notify() o para despertar todos los hilos notifyAll().

# Using conditions in synchronized code

- Cree una clase llamada EventStorage. Con dos atributos uno entero llamado maxSize y una lista (buffer) llamada storage.

```
public class EventStorage {  
  
    private int maxSize;  
    private List<Date> storage;
```

- Implemente el constructor :

```
    public EventStorage() {  
        maxSize=10;  
        storage=new LinkedList<>();  
    }
```

# Using conditions in synchronized code

- Implemente el método SET. Se utilizará por los productores. Primero se chequea que haya lugar en el buffer, sino se ejecuta wait() hasta que haya lugar. Al final se ejecuta notifyAll(), para despertar todos los hilos que estén durmiendo.

```
public synchronized void set() {  
    while (storage.size() == maxSize) {  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
    storage.offer(new Date());  
    System.out.printf("Set: %d", storage.size());  
    notifyAll();  
}
```

# Using conditions in synchronized code

- Implemente el método GET. Se utilizará por los consumidores. Primero se chequea que haya elementos en el buffer, sino se ejecuta wait(). Al final se ejecuta notifyAll(), para despertar todos los hilos que estén durmiendo.

```
public synchronized void get(){
    while (storage.size()==0){
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.printf("Get: %d: %s",storage.
size(), ((LinkedList<?>)storage).poll());
    notifyAll();
}
```

# Using conditions in synchronized code

- Cree una clase Producer y especifique que implementa la interfaz runnable. Declare un objeto Eventstorage e implemente el constructor.

```
public class Producer implements Runnable {  
    private EventStorage storage;  
  
    public Producer(EventStorage storage) {  
        this.storage=storage;  
    }  
}
```

# Using conditions in synchronized code

- Implemente el método run, este llamara 100 veces al método set.

```
@Override
public void run() {
    for (int i=0; i<100; i++) {
        storage.set();
    }
}
```

- Cree la clase llamada Consumer

```
public class Consumer implements Runnable {
```

# Using conditions in synchronized code

- Declare un objeto e implemente el constructor:

```
private EventStorage storage;

public Consumer(EventStorage storage) {
    this.storage=storage;
}
```

- Implemente el método run, el cual ejecuta 100 llamados al get.

```
@Override
public void run() {
    for (int i=0; i<100; i++){
        storage.get();
    }
}
```



# Using conditions in synchronized code

- Cree la clase Main:

```
public class Main {  
  
    public static void main(String[] args) {
```

- Cree un objeto EventStorage, un producer y un hilo para ejecutarlo y un consumer y un hilo. Y arranque los hilos.

```
        EventStorage storage=new EventStorage();  
        Producer producer=new Producer(storage);  
        Thread thread1=new Thread(producer);  
        Consumer consumer=new Consumer(storage);  
        Thread thread2=new Thread(consumer);  
  
        thread2.start();  
        thread1.start();
```

# Using conditions in synchronized code

- La salida del programa:

```
Console X
<terminated> Main (16) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
Set: 1 |
Get: 1: Mon May 15 20:29:43 ART 2017
Set: 1
Set: 2
Get: 2: Mon May 15 20:29:43 ART 2017
Set: 2
Set: 3
Set: 4
Get: 4: Mon May 15 20:29:43 ART 2017
Set: 4
Get: 4: Mon May 15 20:29:43 ART 2017
Set: 4
Set: 5
Set: 6
Get: 6: Mon May 15 20:29:43 ART 2017
Set: 6
Set: 7
Get: 7: Mon May 15 20:29:43 ART 2017
Set: 7
Set: 8
Get: 8: Mon May 15 20:29:43 ART 2017
Set: 8
Set: 9
Get: 9: Mon May 15 20:29:43 ART 2017
Set: 9
Get: 9: Mon May 15 20:29:43 ART 2017
Set: 9
Set: 10
Set: 11
Get: 11: Mon May 15 20:29:43 ART 2017
Set: 11
```

# Synchronizing a block of code with Lock

- Java provee otro mecanismo para la sincronización de acceso a bloques de código.
- Este mecanismo presenta las siguientes ventajas:
  - Permite la estructuración de bloques de manera más flexible.
  - Este mecanismo es mas potente, presenta funcionalidades mejoradas como el `tryLock()`. Pregunto si esta tomado el lock (no me bloqueo).
  - La interface lock permite separación de readers y writers.
  - Mejor performance.

# Synchronizing a block of code with Lock

- En el próximo ejercicio, se aprenderá a utilizar la herramienta LOCK, para sincronizar un bloque de código y se creará una sección crítica usando locks (reentrants locks).
- Clases a utilizar:
  - Printqueue. Implementara una cola de impresión.
  - Job.
  - Main.

# Synchronizing a block of code with Lock

- Clase Main: lanza 10 hilos ejecutando objetos de tipo Job en paralelo. Estos envían trabajos a imprimir al mismo tiempo.

```
*Main.java  Job.java  PrintQueue.java
1 package com.packtpub.java7.concurrency.chapter2.recipe3.core;
2 import com.packtpub.java7.concurrency.chapter2.recipe3.task.Job;
3
4
5 public class Main {
6
7     public static void main (String args[]){
8
9         // Creates the print queue
10        PrintQueue printQueue=new PrintQueue();
11
12        // Creates ten Threads
13        Thread thread[]=new Thread[10];
14        for (int i=0; i<10; i++){
15            thread[i]=new Thread(new Job(printQueue),"Thread "+i);
16        }
17
18        // Starts the Threads
19        for (int i=0; i<10; i++){
20            thread[i].start();
21        }
22    }
23
24 }
25
```

# Synchronizing a block of code with Lock

- Clase Job: simula una tarea que envía un documento a la impresora.

```
*Main.java  *Job.java  PrintQueue.java

1 package com.packtpub.java7.concurrency.chapter2.recipe3.task;
2 public class Job implements Runnable {
3
4     private PrintQueue printQueue;
5
6     public Job(PrintQueue printQueue){
7         this.printQueue=printQueue;
8     }
9
10    @Override
11    public void run() {
12        System.out.printf("%s: Going to print a document\n",Thread.currentThread().getName());
13        printQueue.printJob(new Object());
14        System.out.printf("%s: The document has been printed\n",Thread.currentThread().getName());
15    }
16 }
17
```

# Synchronizing a block of code with Lock

- PrintQueue: Esta clase simula una cola de impresión. Crea un objeto de tipo lock para controlar el acceso.

```
*Main.java  *Job.java  *PrintQueue.java X
1 package com.packtpub.java7.concurrency.chapter2.recipe3.task;
2
3 import java.util.concurrent.locks.Lock;
4
5
6 public class PrintQueue {
7     private final Lock queueLock=new ReentrantLock();
8
9     public void printJob(Object document){
10
11         queueLock.lock();
12
13         try {
14             Long duration=(long)(Math.random()*10000);
15             System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",
16                               Thread.currentThread().getName(),(duration/1000));
17             Thread.sleep(duration);
18         } catch (InterruptedException e) {
19             e.printStackTrace();
20         } finally {
21             queueLock.unlock();
22         }
23     }
24 }
25
```

# Synchronizing a block of code with Lock

- La salida del programa.

 Console 

```
<terminated> Main (17) [Java Application] C:\Program Files\Java\jre7\bi  
Thread 0: Going to print a document  
Thread 7: Going to print a document  
Thread 0: PrintQueue: Printing a Job during 2 seconds  
Thread 6: Going to print a document  
Thread 9: Going to print a document  
Thread 8: Going to print a document  
Thread 3: Going to print a document  
Thread 5: Going to print a document  
Thread 4: Going to print a document  
Thread 2: Going to print a document  
Thread 1: Going to print a document  
Thread 0: The document has been printed  
Thread 7: PrintQueue: Printing a Job during 0 seconds  
Thread 7: The document has been printed  
Thread 6: PrintQueue: Printing a Job during 8 seconds  
Thread 6: The document has been printed  
Thread 9: PrintQueue: Printing a Job during 9 seconds  
Thread 9: The document has been printed  
Thread 8: PrintQueue: Printing a Job during 3 seconds
```



# Locks - Details

- Bloque de código FINALLY. Obliga la ejecución del mismo, aun cuando hay una excepción que haga salir del bloque try catch.
- Cuidado con el orden de los LOCKS, dos hilos tomando locks en orden diferente pueden terminar en deadlock.
- Los locks reentrants, permite el llamado a métodos recursivos.