



Programación Concurrente - Thread Safety

NOTA 3:

Hay recursos que son liberables en Java (los que implementan la interfaz Closable)

Java ahora nos deja hacer try's definiendo variables, para que se liberen solas despues del bloque try-catch (evitando poner un finally para que cierre manualmente todos esos recursos)

Ejemplo:

```
readFile(){  
    try(Reader reader = new BufferedReader(...){  
        ...  
    } catch() {  
        ...  
    }  
}
```

NOTA 1:

El close de executorService es un metodo default que tiene timeout de UN DIA, NO USARLO

NOTA 2:

Es importante armar pools, y liberar todo lo que no sea necesario

Problemas de la Programación Concurrente

Ejercicio 1 - Thread Safety

- Bajar el archivo thread-safety.zip e incorporarlo a la IDE como un proyecto.
- Copiar de la clase anterior la implementación del Generic Service.
- En ***GenericServiceConcurrencyTest*** implementar:
 - Una instancia del servicio como variable de instancia.
 - La variable ***visitor*** como un Runnable que realice una cantidad de visitas al servicio (la cantidad dada por la variable VISITS_BY_THREAD).
 - ***visit_count_with_thread_start***: que ejecuta una cantidad de threads usando Thread#start para lanzarlas y chequea la respuesta esperada de la cantidad de visitas (cantidad de threads dada por THREAD_COUNT)
 - ***visit_count_with_executor_submit***: que ejecuta THREAD_COUNT threads usando ExecutorService#submit para lanzarlas y chequea la respuesta esperada de la cantidad de visitas.



Problemas de la Programación Concurrente

El modelo de programación concurrente presenta nuevos elementos a tener en cuenta:

- Consistencia / Sincronización
- Disponibilidad (liveness)
- Coordinación

Cada uno de estos elementos surgen a partir de que el modelo trae beneficios pero requiere un poco más de cuidado al utilizarlo.



Consistencia en memoria

Los Threads se comunican principalmente mediante **variables compartidas**, ya sea instancia y estáticas.

Esto se debe a que **los Threads viven dentro del espacio de memoria del proceso principal**, entonces esta "comunicación" de esta manera es natural y hasta es eficiente.

```
public class CommunicatingThreads {
    private boolean done = false;

    public boolean isDone() {
        return done;
    }

    public void setDone() {
        this.done = true;
    }

    public static void main(String[] args) throws InterruptedException {
        final CommunicatingThreads ct = new CommunicatingThreads();
        final ExecutorService executor = Executors.newCachedThreadPool();

        executor.submit(() -> {
            while(!ct.isDone()){
                System.out.println("not done");
            }
            System.out.println("done");
        });
        Thread.sleep(2000L);
        ct.setDone();
    }
}
```



Consistencia en memoria

Más allá de las ventajas

El compartir memoria trae como consecuencia la posibilidad de potenciales “problemas” donde la escritura y/o lectura de esa zona compartida puede dar valores incorrectos.

Estos problemas son denominados problemas de consistencia de memoria.

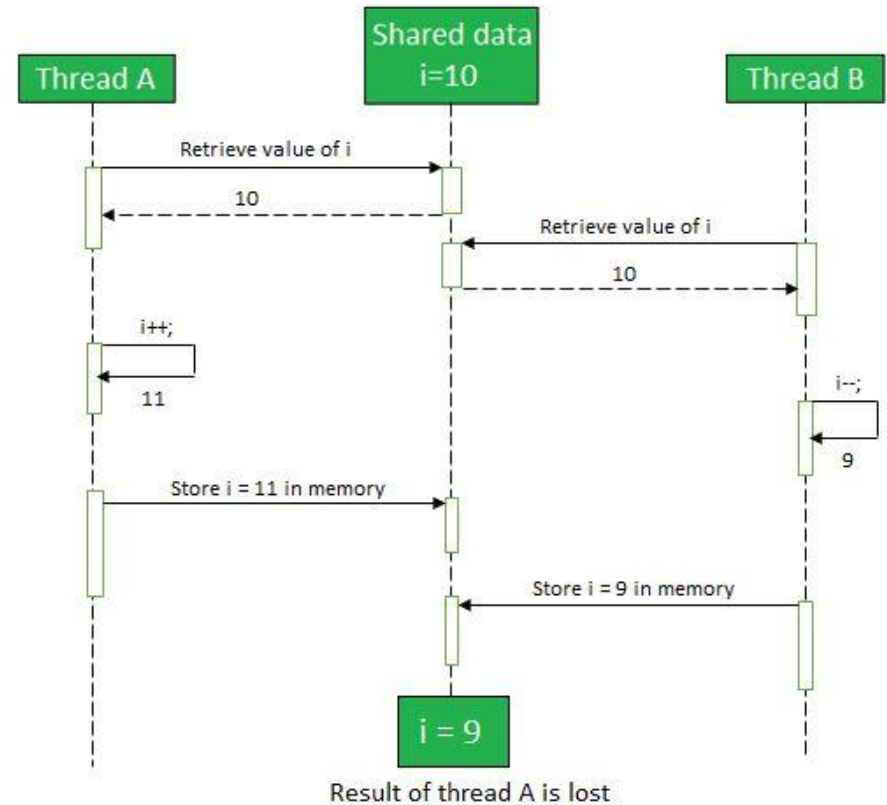


Problemas de consistencia en memoria

Este caso es cuando dos threads quieren escribir. Los dos threads agarran el valor y le hacen una operacion (de suma o resta), pero no detecta que cambio y altera el valor viejo.

Interferencia entre Threads

Se puede dar el caso que **dos threads** se interfieran en sus escrituras, cuando uno escribe en una variable sin detectar que el otro ya la modificó.

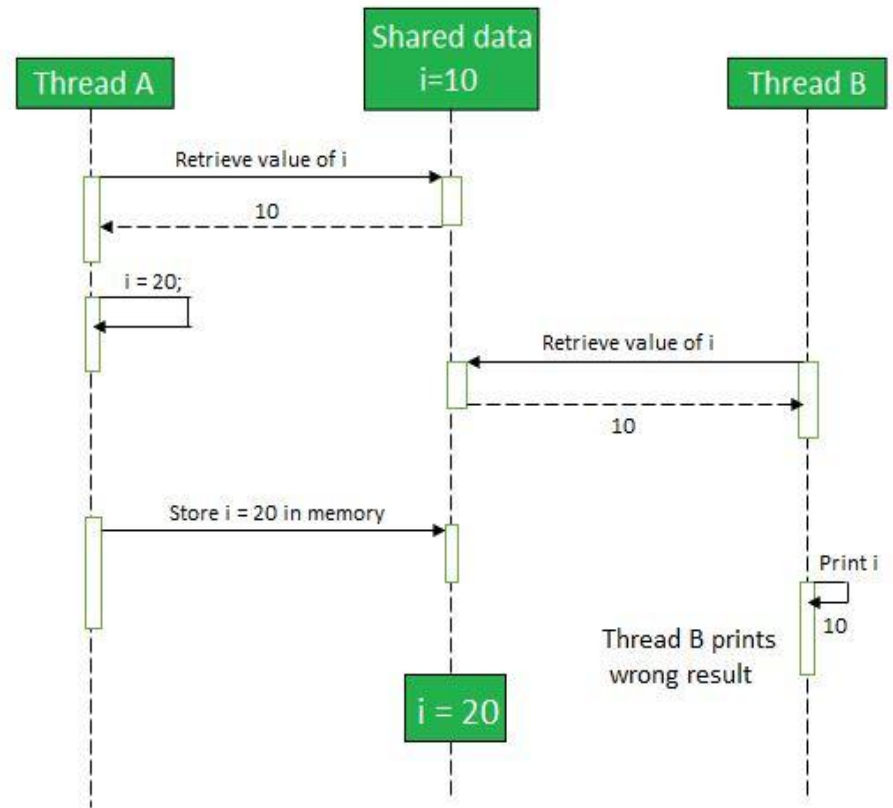


Problemas de consistencia en memoria

Este problema es cuando el thread 1 escribe una variable, el thread 2 lee información vieja. Esto se debe a que en el medio hay distintos caches (como el del procesador, del SO, de Java)

Errores de consistencia de memoria

No hay garantías de que una lectura de una variable en un thread vea lo escrito "aparentemente antes" por otro Thread.





Consistencia en memoria - Happens Before

Estos errores se producen porque hay muy pocos casos donde se garantiza que **una acción se ejecute antes otra.**

Ese tipo de relación se denomina [happens-before](#).

En muchos casos se requiere que el programador se encargue de establecer dicha relación.

No todo tiene esta relacion seteada en True. Esto es porque hay operaciones que son atomicas, y otras que no (por tema de costo de tiempo)



Happens before no es algo que se puede configurar, es un ideal que se intenta conseguir con técnicas de programación Thread Safe

Consistencia en memoria - Happens Before

Para tener en cuenta las garantías de Java en cuanto a happens-before:

- Cada acción en un Thread “happens-before” se ejecuta antes que acciones posteriores, según el programa, en el mismo Thread. “Cada instrucción corre después de la anterior”
- Un unlock (la salida de un bloque o método sincronizado) de un monitor “happens-before” de cualquier lock sobre el mismo monitor.
- Una escritura en una **variable volátil** “happens-before” que cualquier lectura sobre el mismo campo.
- Una llamada al método start de un Thread “happens-before” que cualquier acción en el Thread
- Todas las acciones en un Thread “happens-before” que las acciones en otro Thread que hizo un join sobre el primer Thread.

Las variables volatile originalmente se usaban para indicarle a la JVM que tiene que estar protegida para lecturas.

Sin embargo, hay situaciones en las que volatile falla porque esta no soluciona los problemas de cache de procesador.

La idea es que las escrituras en un volatile siempre se hace antes que la lectura.

NOTA: en ese momento, las volatile solo andaban para int, no para double porque ya habia que leer multiples bytes.

Accesos Atómicos

Una **acción atómica** es aquella acción que no puede detenerse mientras está ocurriendo. O no hay estado intermedio ni interrupciones posibles.

En Java se garantiza que son atómicas: En Java, son atómicas las asignaciones DE REFERENCIAS, las que no son de referencias (como asignar un double) no lo son.

- Lecturas y escrituras de referencias a variables y la mayoría de las primitivas (exceptuando longs y doubles).
- Lecturas y escrituras a variables declaradas como **volatile**.

Es importante aclarar que aunque declarar una variable volatile reduce la posibilidad de errores de consistencias ya que todas las escrituras tienen una relación de happens-before con las escrituras, aún se pueden provocar errores de consistencia en operaciones no atómicas (Ejemplo: ++).

Por lo cual puede ser **necesario sincronizar**.

En JAVA, toda instancia de cualquier objeto viene con un MUTEX incluido, es por eso que en el synchronized puedo usar un Object lock = "lock". En realidad podría usar cualquier objeto. Tener en cuenta que para evitar problemas le ponemos que es FINAL.

Sincronización

La keyword **synchronized** permite generar bloques de sincronización donde se cumple:

- Si un Thread entra en un bloque sincronizado, cualquier otro Thread que intente entrar en cualquier otro bloque que esté sincronizado con la misma instancia queda bloqueado.
- Cuando el Thread sale del bloque, se libera uno de los Threads que están bloqueados y se genera una relación happen-before entre las acciones realizadas por el Thread que sale con las que realizará el Thread que entra.

Hay que proteger tanto las lecturas como las escrituras

NOTA: NO usar strings como locks, porque en java son Singletons, entonces todas las strings "lock" son el mismo puntero.

Esto puede introducir bugs.

```
public class SynchronizdObjectVisitCounter {  
    private final Object lock = "lock";  
    private int c = 0;  
  
    public void addVisit() { Esto lo hace atomico  
        synchronized (lock) {  
            c++; En este caso, los dos bloques  
                estan protegidos por el mismo  
                MUTEX  
        }  
    }  
    public int getVisits() {  
        synchronized (lock) {  
            return c;  
        }  
    }  
    public int peekVisits() {  
        return c;  
    }  
}
```



Sincronización - Internals

Internamente se produce este comportamiento debido a que:

- En Java **cada instancia** de un objeto puede tener **asociado un único lock interno** (monitor lock o mutex).
- Al **acceder a un bloque** sincronizado sobre una instancia efectivamente **se está adquiriendo dicho lock**. Y cualquier **otro que quiera el lock queda bloqueado** hasta que el primero lo libere.
- Los locks son re-entrantes para el Thread que los posee.

re-entrante: puedo poder a pedir el mismo lock



Sincronización - Internals

Cosas a tener en cuenta

- No olvidar que una “Clase” en Java también es una instancia de la clase **Class**, por lo tanto si **las variables de clase son recurso compartido entre threads**, también puede ser necesario sincronizarlas. Esto es porque la clase también es un objeto (que puedo instanciar)
- **No debería lockear un objeto por más tiempo que lo necesario** para permitir que otros threads puedan solicitar el lock.
- **Lamentablemente no hay timeouts para este tipo de sincronización.**

Sincronización de Métodos

Synchronized puede usarse a nivel método, en cuyo caso el lock es sobre **this**.

```
public class SynchronizedMethodVisitCounter
{
    private int c = 0;

    public synchronized void addVisit() {
        c++;
    }

    public synchronized int getVisits() {
        return c;
    }

    public int peekVisits() {
        return c;
    }
}
```

Equivalentes

```
public class SynchronizdThisVisitCounter
{
    private int c = 0;
    public void addVisit() {
        synchronized (this) {
            c++;
        }
    }
    public int getVisits() {
        synchronized (this) {
            return c;
        }
    }
    public int peekVisits() {
        return c;
    }
}
```

Ejercicio 2-3 Sincronización

- Corregir en el ***GenericServiceConcurrencyTest*** el Runnable para que no se provoquen los problemas de memoria detectados durante el ejercicio anterior.

High level Locks

Existen dentro del paquete [java.util.concurrent.locks](#) una serie de objetos que permite realizar locks con otras **semánticas más complejas que `synchronized`**:

- **Lock**: Interfaz básica de un Lock.
- **ReadWriteLock**: Interfaz que mantiene dos locks asociados, uno para lectura y otro para escritura.
- **ReentrantLock**: Lock re-entrante similar a `synchronized` pero que permite extender su funcionalidad
- **ReentrantReadWriteLock**: Combinación entre read-write y reentrant.
- **StampedLock**: Lock más complejo con 3 tipos de operaciones (`read`, `write`, `readOptimistic`).

```
public class BankAccount {  
    final Lock lock = new ReentrantLock();  
    ...  
    static void transfer(BankAccount from, BankAccount  
        to, double amount) {  
        from.lock.lock();  
        from.withdraw(amount);  
        to.lock.lock();  
        to.deposit(amount);  
        to.lock.unlock();  
        from.lock.unlock();  
    }  
    ...  
}
```

readOptimistic: hay un versionado. Por ejemplo: yo leo la version 5 del objeto, si cuando voy a escribir sigue en la version 5 todo OK. Si ya no esta en la version 5, no me deja escribir y tengo que volver a leer.

**Los unlocks generalmente
deben ir en un bloque
finally!!!**

High Level Atomic Classes

El paquete [java.util.concurrent.atomic](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html) presenta clases que sustituyen a primitivas y tienen operaciones atómicas thread safe sin realizar locks:


- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicLongArray
- AtomicReference<V>
- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- LongAdder

```
public class Adder implements Counter {  
    private final LongAdder adder = new LongAdder();  
    @Override  
    public long getCounter() {  
        return adder.longValue();  
    }  
    @Override  
    public void increment() {  
        adder.increment();  
    }  
}
```



Concurrent Collections


El paquete [java.util.concurrent](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html) ofrece una serie de implementaciones de Cola útiles para el manejo de Threads

- **ConcurrentLinkedQueue** y **ConcurrentLinkedDeque**: Ofrecen una cola sin límites de capacidad, thread safe.
 - **BlockingQueue**: es una interfaz que define los métodos **put** y **take** **bloqueantes** y tiene diversas implementaciones como **LinkedBlockingQueue**, **ArrayBlockingQueue**, **SynchronousQueue** (sin capacidad 1 r/w), **PriorityBlockingQueue** (ordenada por prioridad), and **DelayQueue** (elementos que aparecen cuando se vence el delay).
- 



Concurrent Collections

[java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) además ofrece implementaciones de otras colecciones Thread Safe


- **ConcurrentHashMap,**
 - **ConcurrentSkipListMap,**
 - **ConcurrentSkipListSet,**
 - **CopyOnWriteArrayList**
 - **CopyOnWriteArraySet.**
- 



Liveness - Problemas

Al utilizar Threads y locks para coordinar entre los mismos empiezan a aparecer otra clase de problemas.

Porque la sincronización puede provocar que los Threads no terminen por diversos motivos.



Liveness - DeadLock

DeadLock

Este caso ocurre cuando dos Threads quieren adquirir dos locks antes de realizar una acción y cada Thread toma uno de los locks e intenta pedir el otro.

En ese caso ambos se quedan esperando que el otro lock sea liberado, cosa que nunca ocurrirá ya que ellos no liberan el suyo.

```
void transfer(BankAccount from, BankAccount to, double amount) {  
    synchronized (from) {  
        from.withdraw(amount);  
        synchronized (to) {  
            to.deposit(amount);  
        }  
    }  
}
```

El LiveLock tiene pocas posibilidades de ocurrir, porque ambos Threads tienen que dormirse y despertarse al mismo tiempo.

Liveness - LiveLock

Algunas soluciones a esto son esperar una cantidad random de tiempo (para que los dos threads no matcheen), y reinetentar por un numero determinado de veces, luego de eso fallar.

LiveLock

Se da cuando para evitar el deadlock los threads, al no poder tomar el segundo lock, liberan el primer lock, esperan un tiempo y reintentan.

En este caso puede darse que ese ciclo de probar y reintentar se convierta en un ciclo infinito.

```
boolean withdraw(double amount) {  
    if (this.lock.tryLock()) {  
        try {  
            Thread.sleep(101);  
        } catch (InterruptedException e) {  
        }  
        balance -= amount;  
        return true;  
    }  
    return false;  
}
```

Si obtengo el lock ejecuto, si no digo que no pude

Liveness - LiveLock

```
public boolean tryTransfer(BankAccountLiveLock destinationAccount,
double amount) {
    if (this.withdraw(amount)) {
        if (destinationAccount.deposit(amount)) {
            return true;
        } else {
            // destination account busy, refund source account.
            this.deposit(amount);
        }
    }

    return false;
}
```

Si puedo tomar los locks se ejecuta, si no rollback

```
while (!sourceAccount.tryTransfer(destinationAccount, amount))
```

Mientras no pueda hacer la transferencia reintento

En el caso del Deadlock en las transferencias bancarias, una posible solución es tomar siempre los locks en el mismo orden (por ejemplo, siempre se toma primero el lock del número de cuenta menor)

Liveness - Starvation

Starvation

Esto ocurre cuando algún Thread no puede tomar nunca el lock o algún recurso que espera porque siempre está tomado por otros Threads que (probablemente) tardan mucho en liberarlo y/o tienen mejor prioridad.

Por ejemplo, si yo le doy mayor prioridad a las escrituras que a las lecturas y siempre se está escribiendo, los threads que quieras leer podrían nunca recibir el lock.

```
Thread balanceMonitorThread1 = new Thread(new
BalanceMonitor(fooAccount), "BalanceMonitor");

Thread transactionThread1 = new Thread(new
Transaction(fooAccount, barAccount, 250d), "Transaction-1");

Thread transactionThread2 = new Thread(new
Transaction(fooAccount, barAccount, 250d), "Transaction-2");

balanceMonitorThread1.setPriority(Thread.MAX_PRIORITY);
transactionThread1.setPriority(Thread.MIN_PRIORITY);
transactionThread2.setPriority(Thread.MIN_PRIORITY);
```

Coordinación

Para **resolver ciertos problemas** se requiere que los Threads se coordinen. Por ejemplo que un Thread corra luego de que otros threads terminen algún procesamiento:

La **forma nativa** sería mediante el uso del método **wait** para el que espera y **notifyAll** para el que notifica que terminó.

Además de alguna **variable compartida** para el estado de la tarea.


```
void blockThread() {  
    synchronized (lock) {  
        shouldWait = true;  
        while (shouldWait) {  
            try {  
                lock.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
void notifyThread() {  
    synchronized (lock) {  
        shouldWait = false;  
        lock.notify();  
    }  
}
```



Coordinación

Para usos más complejos también existen clases de coordinación:

- **Semaphore:** Hay un número de “tickets” y se van pidiendo hasta que se acaban
 - **Condition:** Permite que un thread espere a otro.
 - **CountDownLatch:** Un thread espera a que n threads indiquen que están en “un punto”.
 - **CyclicBarrier:** N threads se esperan hasta que todos llegan al mismo punto
- 

Coordinación - Semaphore

```
public class LoginQueue {  
  
    private Semaphore semaphore;  
  
    public LoginQueue(int slotLimit) {  
        semaphore = new Semaphore(slotLimit);  
    }  
  
    boolean tryLogin() {  
        return semaphore.tryAcquire();  
    }  
  
    void logout() {  
        semaphore.release();  
    }  
  
    int availableSlots() {  
        return semaphore.availablePermits();  
    }  
}
```

Coordinación - **CountDownLatch**

Esta hecho para que un Thread espere a muchos otros.

```
public boolean callTwiceInSameThread() {  
    CountDownLatch countDownLatch = new CountDownLatch(count);  
    Thread t = new Thread(() -> {  
        countDownLatch.countDown();  
        countDownLatch.countDown();  
    });  
    t.start();  
  
    try {  
        countDownLatch.await();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    return countDownLatch.getCount() == 0;  
}
```

El ejemplo este no es el mejor, ya que muestra un Thread esperando a un solo Thread.

Timeouts

Siempre se recomienda usar timeouts, es la mejor estrategia para evitar Deadlocks no detectados

La mayoría de los métodos bloqueantes de las clases del paquete concurrent proveen versiones que se le puede pasar un **long** y un **TimeUnit**.

Esto es para **proveer un Timeout** para que el bloqueo no sea "para siempre" si es que nunca se libera.


Es recomendable y muy buena práctica utilizar estos métodos y manejar el caso de timeout (Por ejemplo N cantidad de re-tries y luego excepción).

```
final Lock lock = new ReentrantLock();
public int processWithLock(){
    for(int i=0; i < 5; i++){
        if (lock.tryLock(50L, TimeUnit.SECONDS)) {
            int rta = process();
            lock.unlock();
            return rta;
        }
    }
    throw new IllegalStateException("No se pudo obtener lock: vencida la cantidad de re-tries");
}
```



Para finalizar

Para evitar estos problemas hay estrategias como **sincronización, coordinación,** manejo de **objetos inmutables y subdivisión de tareas.**
Varias de estas estrategias las volveremos a ver al pasar a programación distribuida.





CREDITS

Content of the slides:

- » POD - ITBA

Images:

- » POD - ITBA
- » Or obtained from: commons.wikimedia.org

Slides theme credit:

Special thanks to all the people who made and released these awesome resources for free:

- » Presentation template by [SlidesCarnival](https://slidescarnival.com)
 - » Photographs by [Unsplash](https://unsplash.com)
- 