



Threads 2 (Coordinació)

MP09 UF2 A2.2



Threads en JAVA, coordinació

Les aplicacions amb diversos fils presenten problemes quan diferents fils accedeixen a les mateixes dades:

- **Exclusió:** només un thread hauria d'executar seccions crítiques en un moment donat
- **Visibilitat:** Els canvis que un fil fa a les dades s'han de veure de manera consistent pels altres fils
- **Sincronia:** Els diferents fils s'han de posar d'acord per no trepitjar-se les modificacions a les dades entre ells

Nota: JAVA ens pot ajudar en aquests punts, però ha de ser el programador el que ha de fer el codi correctament per complir-los



Threads en JAVA, sincronització de les dades

Tenir informació compartida entre diferents fils, implica que aquestes dades es poden sobreescriure o poden llegir en ordre incorrecte. Per evitar-ho hi ha diferents estratègies:

- Funcions '**synchronized**'
- Objecte '**ReentrantLock**', similar a l'anterior amb funcionalitats avançades
- Objecte '**Semaphore**', controlar l'accés a memòria des de múltiples fils

Nota: [Aquest post](#) explica molt bé l'ús de cada mètode

Nota: Els [semàfors](#) són el mètode clàssic i es fa servir a nivell de sistema operatiu i llenguatges de programació com C o C++

Threads en JAVA, synchronized

```
static Object mutex = new Object();  
static int compartida = 0;  
  
public synchronized void augmentaValor () {  
    synchronized(mutex) {  
        compartida = compartida + 1;  
    }  
}
```

Les funcions 'synchronized' poden assegurar que un fil de la mateixa classe executa un tros de codi en un moment donat.

Cal un objecte que permet aquesta exclusió. Habitualment s'anomena 'mutex', perquè fa referència a l'exclusió mútua

Threads en JAVA, synchronized

```
static Object mutex = new Object();  
static int compartida = 0;  
  
public synchronized void augmentaValor () {  
    synchronized(mutex) {  
        compartida = compartida + 1;  
    }  
}
```

Les funcions 'synchronized' poden assegurar que un fil de la mateixa classe executa un tros de codi en un moment donat.

Cal un objecte que permet aquesta exclusió. Habitualment s'anomena 'mutex', perquè fa referència a l'exclusió mútua



Threads en JAVA, ReentrantLock

```
public class Pistola {  
    private boolean enposicio = true;  
    public synchronized void disparar(int cartucho)  
    {  
        while (enposicio == false) {  
            try { wait(); }  
            catch (InterruptedException e) { }  
        }  
        enposicio = false;  
        notifyAll();  
    }  
    public synchronized void apuntar() {  
        while (enposicio == true) {  
            try { wait(); }  
            catch (InterruptedException e) { }  
        }  
        enposicio = true;  
        notifyAll();  
    }  
}
```

Quan hi ha dos '**synchronized**' en una classe, asseguren que els dos mètodes no funcionin al mateix temps.

Habitualment quan dos '**threads**' criden mètodes d'aquesta classe



Threads en JAVA, Semàfors

```
static Semaphore mutex = new Semaphore(1);
static int compartida = 0;

static void augmentaValor () {
    try {
        try {
            mutex.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        compartida = compartida + 1;
    } finally {
        mutex.release();
    }
}
```

Els semàfors controlen l'accés a un recurs a través d'un comptador.

En aquest exemple només es permet 1 procés simultàniament, és un semàfor binari.

Nota: Si volem que 3 processos accedeixin a aquest tros de codi podem crear el semàfor amb:

'new Semaphore(3)'



Threads en JAVA, Semàfors vs ReentrantLock

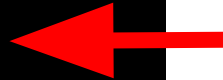
- Els semàfors són un mètode de senyalització, ReentrantLock és un mètode de bloqueig
- Cap thread és propietari d'un semàfor, en canvi l'últim mètode que ha blocat un codi és el propietari del 'ReentrantLock'. Això té implicacions a la hora de programar amb 'ReentrantLock'
- Els semàfors asseguren que un mateix thread no aconseguirà el control un cop rere l'altre bloquejant els altres processos, evitant que el codi sigui fàcil de penjar
- Els semàfors permeten configuracions i codis més complexes amb operacions com 'wait'



Threads en JAVA, memoria compartida (static)

```
class Tmp2 {  
    static int compres = 0;  
    public static void afegirAlCarret (int  
quantitat) {  
        compres = compres + quantitat;  
    }  
}
```

```
Tmp2.compres = 5;      // Sense instància  
Tmp2.afegirAlCarret(3);  
  
Tmp2 tmp = new Tmp2(); // Amb instància  
tmp.compres = 6;  
tmp.afegirAlCarret(4);
```



Per diferents fil·ls d'una mateixa classe, es pot fer servir variables '**static**'.

Les variables i funcions '**static**' pertanyen a la classe i no a la instància de la classe i per tant tots els fil·ls d'aquella mateixa classe hi poden accedir.

Nota: Habitualment es fan servir per donar funcionalitat a les classes sense que calgui fer un nou objecte d'aquella classe (instància)



Threads en JAVA, memoria compartida (static)

Les variables 'static' es creen al posar en funcionament el programa i comparteixen espai de memòria per totes les instàncies de l'objecte.

Les altres variables es creen al fer cada instància de l'objecte (new ...) i tenen un espai de memòria propi per cada instància.



Threads en JAVA, memoria compartida

```
public class DadesCompartides {  
    public static int compartida = 0;  
}
```

```
public class FuncionalitatX extends Thread {  
  
    private DadesCompartides objCompartit;  
  
    public FuncionalitatX() {  
        objCompartit = new DadesCompartides();  
        objCompartit.compartida = 3;  
    }  
    ...  
}
```


Quan fils de diferents classes han d'accedir a un mateix conjunt de dades, podem encapsular les variables estàtiques compartides en un objecte que facin servir totes les classes que volen accedir a aquelles dades.

Nota: Ens hem d'assegurar que cada fil (de cada classe) entén que són dades compartides i només les escriu si pot



Threads en JAVA, memoria compartida (volatile)

```
public class DadesCompartides {  
    public static volatile int compartida = 0;  
}
```



Les variables '**volatile**' avisa'n que aquell valor es farà servir des de diferents fils i que les modificacions des de cada un s'han de fer en ordre d'execució.

El programador ha de sincronitzar igualment les lectures i escriptures, és un avís a nivell de màquina virtual. Té un cost en RAM i pot enlentir l'aplicació, però ajuda a evitar errors si s'executen diversos fils al mateix temps.

No té massa sentit si la sincronització només permet un fil simultàniament

```
public class FuncionalitatX extends Thread {  
  
    private DadesCompartides objCompartit;  
  
    public FuncionalitatX() {  
        objCompartit = new DadesCompartides();  
        objCompartit.compartida = 3  
    }  
    ...  
}
```