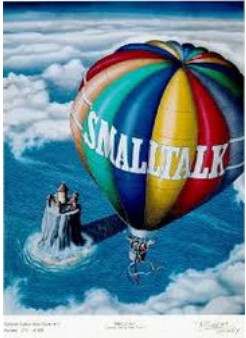




Departamento de Programación  
Facultad de Informática  
Universidad Nacional del Comahue



# Programación Concurrente



*Instrumentos de la  
concurrency*



# Ejemplo contador

```
public class ProcesoI implements
Runnable{
    private Datos unD;
    //crea e inicializa unD

    public ProcesoI(Dato unD){..}

    public void run(){
        //incrementa 10000 veces }
}
```

Classes:

Contador (proceso disparador),  
ProcesoI (hilos), Dato,

```
public class Dato {
    private int dato;
    public Dato(int nro){..}
    public int getDato(){..}
    public void incrementar()
        {dato++; }
}
```

```
public class Contador
{
    public static void main(...){
        Dato unDato;
        ProcesoI p1;
        ProcesoI p2;
        //crea unDato y lo inicializa
        //crea hilos, los ejecuta y luego los finaliza
        // muestra el valor final de unDato
    }
}
```

# Ejemplo contador

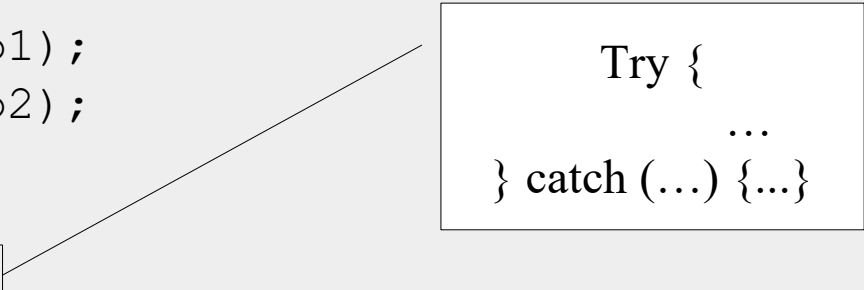
```
public class ProcesoI implements
Runnable{
    private Datos unDato;

    public ProcesoI(Datos unD) {
        unDato = unD;
    }
    public void run() {
        for (int i=1; i<10000; i++){
            unDato.incrementar();
        }
    }
}
```

```
public class Datos {
    private int dato;

    public Datos(int nro) {
        dato = nro;
    }
    public int getDato() {
        return dato;
    }
    public void incrementar()
    {
        dato++;
    }
}
```

```
public class Contador {  
    public static void main(String[] args) {  
        Dato elContador = new Dato(0);  
        ProcesoI p1= new ProcesoI(elContador);  
        ProcesoI p2= new ProcesoI(elContador);  
  
        Thread h1= new Thread(p1);  
        Thread h2= new Thread(p2);  
  
        h1.start(); h2.start();  
        h1.join();  h2.join();  
        System.out.println("en main "+ elContador.getDato());  
    }  
}
```



Try {  
 ...  
} catch (...) {...}

# Código ejecutado varias veces

Resultados diferentes, cercanos al 20000 pero no justo el 20000,

## **Explicación:**

- ambos hilos ejecutan el método *run()*
- la acción de incrementar ( $++$ valor) **no es atómica**
- puede pasar que cuando le toque el turno de ejecución a un hilo, el otro hilo sea interrumpido **justo después de haber recuperado el valor**
- pero antes de modificarlo
- entonces se **pierden incrementos**

```

public class Contador {
    public static void main(String[] args){
        Dato elContador = new Dato(0);
        ProcesoI p1= new ProcesoI(elContador);
        ProcesoI p2= new ProcesoI(elContador);

        Thread h1= new Thread(p1);
        Thread h2= new Thread(p2);

        h1.start(); h2.start();
        h1.join();  h2.join();
        System.out.println("en main "+ elContador.getDato());
    }
}

```

excepcion

```
ProcesoI pp = new ProcesoI (elContador)
```

```

Thread h1= new Thread(pp)
Thread h2 = new Thread(pp)

```



# Cómo resolver el problema?

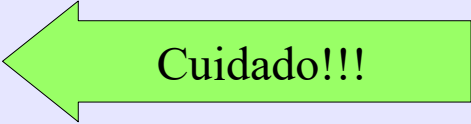
Sincronizar el acceso al recurso (la variable compartida) para lograr la exclusión mutua

```
public class Datos {  
    private int dato;  
  
    public Datos(int nro) {  
        dato = nro;  
    }  
  
    public int synchronized getDato() {  
        return dato;  
    }  
  
    public void synchronized incrementar() {  
        dato++;  
    }  
  
    public void synchronized set(int valor) {  
        Dato= valor;  
    }  
}
```

# Métodos sincronizados

- Sincroniza todo el método

Utiliza el objeto de la clase que posee el método para sincronizar

```
public synchronized void incrementar  
throws InterruptedException {  
  
    dato++;  
    Thread.sleep( ...)  Cuidado!!!  
}
```

Cada objeto en Java tiene un “lock” o llave implícito, disponible para lograr la sincronización



# Synchronized en Java

- Usar synchronized en un método de instancia es lo mismo que poner un bloque de `synchronized(this){}` que contenga todo el código del método.

Es lo mismo:

```
public synchronized void metodo() {  
    // código del método aca  
}
```

```
public void metodo() {  
    synchronized(this) {  
        // código del método aca  
    }  
}
```

- Si el método es de clase entonces es lo mismo pero el bloque de `synchronized` se aplica a la clase.

Ejemplo, si el método está en la clase `MiClase`

```
public static synchronized void metodo() {  
    // código del método aca  
}
```

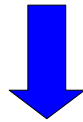
```
public static void metodo() {  
    synchronized(MiClase.class) {  
        // código del método aca  
    }  
}
```

# Sección crítica

- Para entrar a la sección crítica **de manera segura** se debe cumplir:
- **Exclusión mutua**, si un proceso está en su sección crítica, entonces ningún otro proceso puede ejecutar su sección crítica.
- **Progreso**, todos los procesos que no estén en su sección de salida podrán participar en la decisión de quién es el siguiente en ejecutar su sección crítica.
- **Espera limitada**, todo proceso debería poder entrar en algún momento a la sección crítica

# Problemas en lenguajes de alto nivel

Generalmente existen partes de código con **variables compartidas** y que deben ejecutarse en **exclusión mutua**, es decir tomar las operaciones que actúan sobre la variable compartida como **atómicas**.



...  
*contador.incrementar();* Sección crítica

...  
Hay que “controlar” el acceso a la variable (recurso) compartida

# Sección crítica

- El código se divide en las siguientes secciones

SECCION DE ENTRADA

SECCION CRITICA

SECCION DE SALIDA

SECCION RESTANTE

- Existen partes de código con **variables compartidas** y que deben ejecutarse en **exclusión mutua (MUTEX)**

# Sección crítica

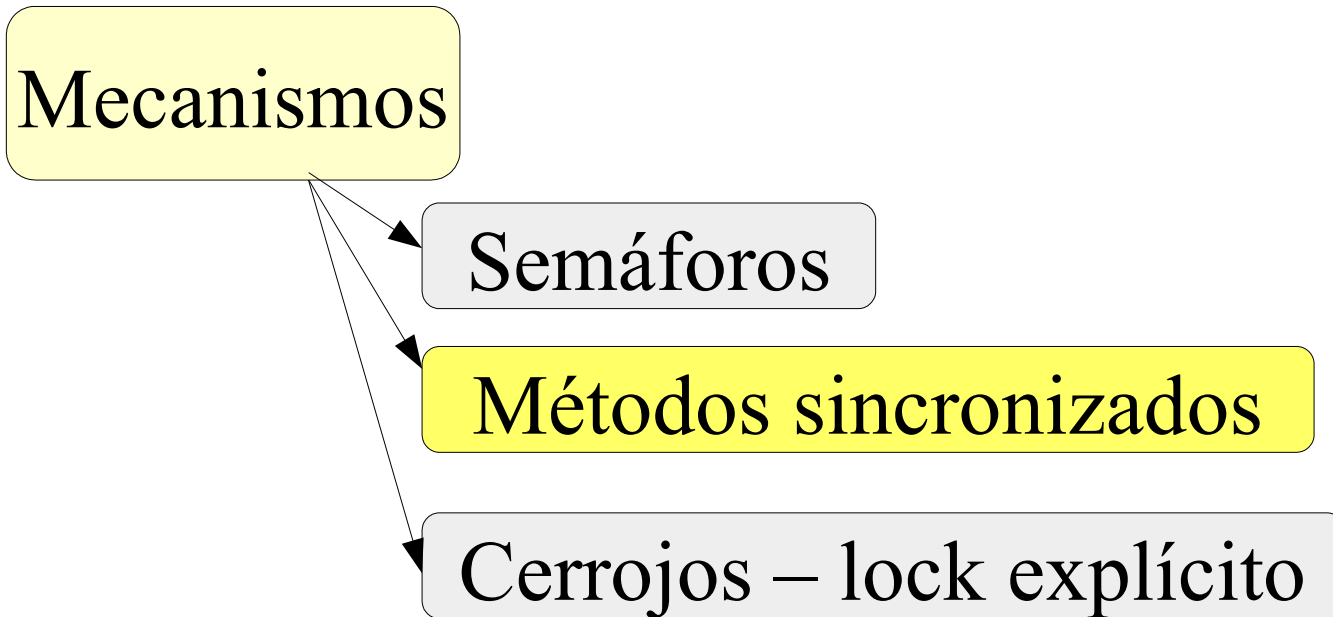
- **Problema:** Garantizar que los procesos involucrados puedan operar sin generar ningún tipo de inconsistencia.

El segmento de código en el que un proceso puede modificar variables compartidas con otros procesos se denomina

## **sección crítica**

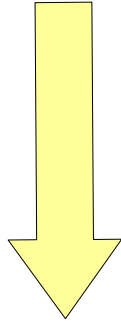
- **Sección de entrada**, se solicita el acceso a la sección crítica.
- **Sección crítica**, en la que se realiza la modificación efectiva de los datos compartidos.
- **Sección de salida**, en la que típicamente se hará explícita la salida de la sección crítica.
- **Sección restante**, que comprende el resto del código fuente.

# Mecanismos para Exclusión Mútua



# Cómo sincronizar?

Usar la palabra **synchronized**

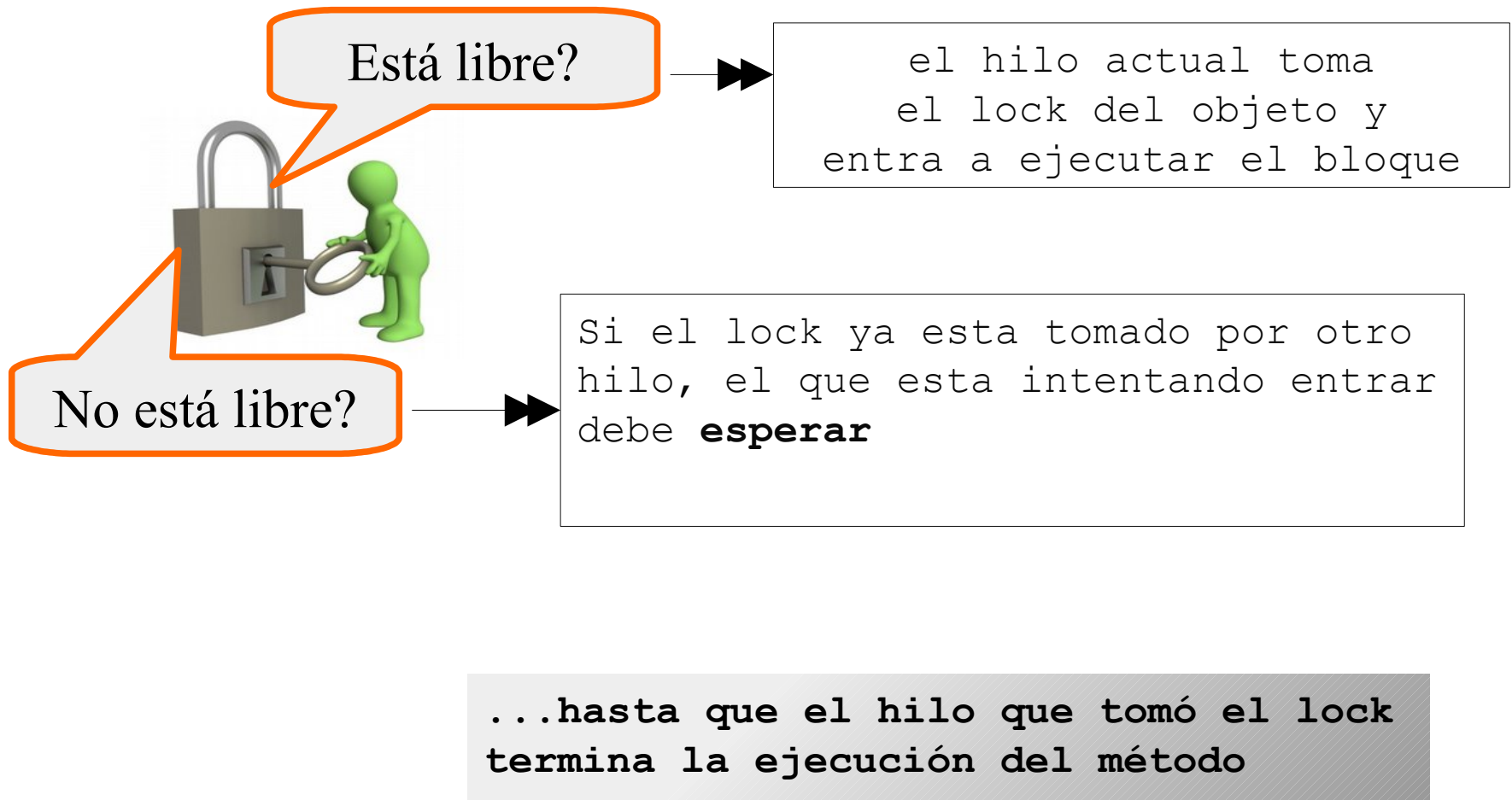


La parte sincronizada  
puede ser ejecutada por  
un sólo hilo por vez



el hilo necesita  
la llave para acceder  
al código sincronizado.

# Cómo sincronizar?





# Exclusion mutua en Java - Synchronized

- Java utiliza **synchronized** para lograr la exclusion mutua sobre los objetos.
- Existen 2 posibilidades para sincronizar objetos
  - El bloque **synchronized**
  - Dentro de la clase del objeto sincronizado **los métodos están declarados como synchronized**
- Cada vez que un hilo intenta ejecutar un método sincronizado sobre un objeto lo puede hacer sólo si no hay algún otro hilo ejecutando un método sincronizado sobre el mismo objeto

# Exclusion mutua en Java - Synchronized

- Un **hilo** que intenta ejecutar un método sincronizado sobre un objeto cuyo lock ya está en poder de otro hilo **es suspendido** y puesto en espera hasta que el lock del objeto es liberado.
- El lock se libera cuando el hilo que lo tiene tomado:  
termina la ejecución del método / ejecuta un return / lanza una excepción.

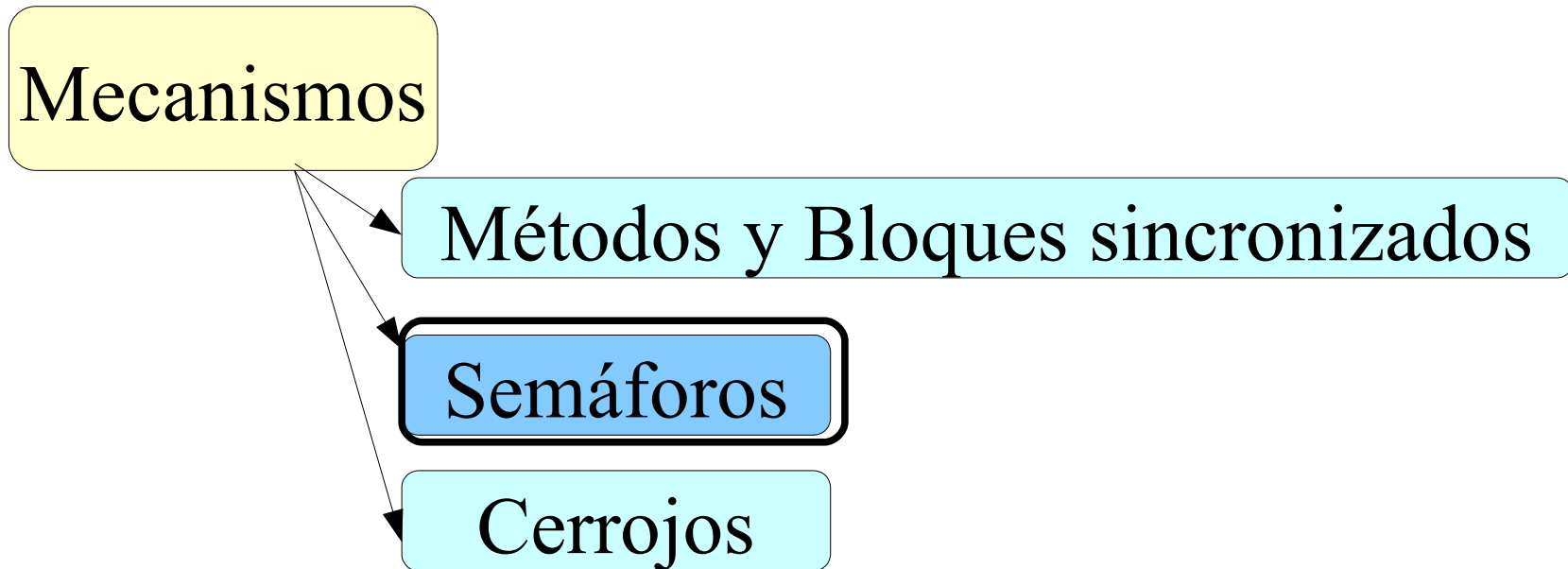
# Exclusión mutua en java – Synchronized

- El mecanismo de sincronización funciona si TODOS los accesos a los *datos delicados* ocurren dentro de métodos sincronizados, es decir con exclusión mútua
- Los datos delicados protegidos por métodos sincronizados deben ser privados

# Exclusion mutua en Java -Synchronized

- Cada instancia de **Object** y sus subclases posee bandera de bloqueo (**lock implícito**)
- Los tipos primitivos (no objetos) solo pueden bloquearse a través de los objetos que los encierran
- No pueden sincronizarse variables individuales
- Los objetos arreglos cuyos elementos son tipos primitivos pueden bloquearse, pero sus elementos NO

# Exclusión mútua



# Mecanismo del semáforo

Los utilizamos para lograr la exclusión mútua. Los procesos COMPITEN por entrar a la sección crítica

Entrada a la SC /adquirir el SEM

**SECCION CRITICA**

Salida de la SC / liberar el SEM

SECCION RESTANTE

*En el algoritmo utilizamos  
las directivas adquirir y liberar,  
pero cada lenguaje  
tiene sus métodos*

ALGORITMO ejemploSem

.....

```
semaforo adquirir
//código sección crítica
semaforo liberar
```

.....

FIN ALGORITMO ejemploSem

# Semáforos, generalidades

Permite que uno de los procesos suspendidos continúe

sólo toman valores 0 o 1.

Semáforo

binarios

clasificación

generales

pueden tomar cualquier valor no negativos

operaciones

liberar

adquirir

el proceso se suspende hasta que se libere el permiso.

# Semáforos, operaciones

- Adquirir

- Si el valor del **semáforo no es nulo** (está abierto o tiene permiso disponible) decrementa el valor del semáforo.
- Si el valor del **semáforo es nulo** (está cerrado o NO tiene permiso disponible), el hilo que lo ejecuta se suspende y se encola en la lista de procesos en espera de un permiso del semáforo.

- liberar



# Semáforos, operaciones

- adquirir
- liberar
  - Si hay **algún proceso en la lista de procesos** del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al “adquirir” que lo suspendió.
  - Si **no hay procesos en espera en la lista** incrementa en 1 el valor del semáforo y queda un permiso disponible.

# Semáforos, en general

- Actúa como **mediador entre un proceso y el entorno** del mismo.
- Proporciona una forma simple de **comunicación sincrónica**.
- Garantiza que la **operaciones** de chequeo del valor del semáforo, y posterior actualización según proceda, sea siempre **segura**
- La inicialización del semáforo **no es una operación segura** por lo que no se debe ejecutar en concurrencia con otro proceso utilizando el mismo semáforo.