Java Threads

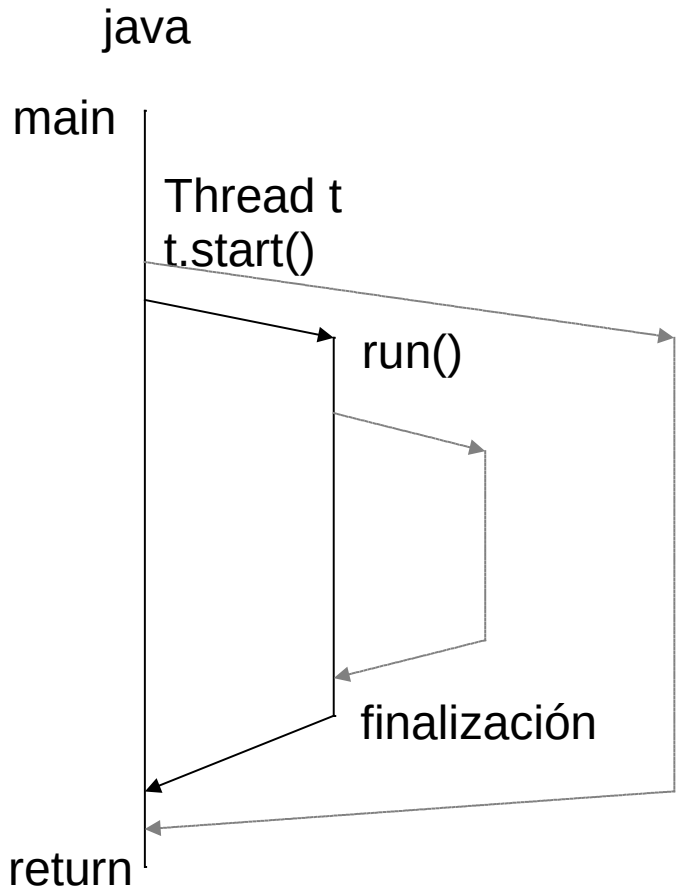
Sistemas Distribuidos
Rodrigo Santamaría

Java Threads

- Hilos
- Sincronización
- Ejercicios
- FAQ

Hilos

- Un hilo (Thread) es un proceso en ejecución dentro de un programa



- Puede haber varios hilos en ejecución simultánea
- Los hilos implementan prioridad y mecanismos de sincronización
- La finalización depende del hilo
- Cualquier clase se puede hacer hilo
  - `extends Thread`
  - `implements Runnable`

Hilos

Thread

```
public class PingPong extends Thread
{
 private String word;
 public PingPong(String s) {word=s;}

 public void run()
  {
   for (int i=0;i<3000;i++)
    {System.out.print(word);
     System.out.flush();}
   }

 public static void main(String[] args)
  {
   Thread tP=new PingPong("P");
   Thread tp=new PingPong("p");
   //tP.setPriority(Thread.MAX_PRIORITY);
   //tp.setPriority(Thread.MIN_PRIORITY);
   tp.start();
   tP.start();
   }
 }
```

- Clase `Thread`
  - Implementa `Runnable`
- `start()→run()`
- `setPriority()`
- `sleep()`
- Hereda de Object
  - `wait()`, `notify()`

OJO**: `run` **no** se ejecuta directamente **nunca*

http://download.oracle.com/javase/tutorial/essential/concurrency

**¿Qué pasaría si ejecutáramos run directamente?*

Hilos

Prioridades

- No hay control por defecto del orden de ejecución de los hilos
  - En el ejemplo anterior de dos hilos jugando al ping pong, tenemos una ejecución como esta:
    - PppppppppPPPPPPPPppppppppPPPPPP
    - Buscamos pPpPpPpPpPpPpPpP
  - Podemos controlar la ejecución mediante prioridades con setPriority()
    - P. ej., si damos a tP prioridad máxima (Thread.MAX_PRIORITY) y a tp prioridad mínima, tendremos (aunque no asegurado)
      - PPPPPPPPPPPPPPpppppppppppppp
- Sin embargo, la prioridad no nos permite un control fino de la ejecución, ni nos asegura un orden concreto
  - Necesidad de mecanismos de sincronización

Hilos

Suspensión

- Se puede interrumpir la ejecución de un hilo temporalmente
  - `Thread.sleep(long millis)`
  - Útil para simular tiempos de procesamiento o timeouts
  - NO para sincronizar hilos, al menos en modelos asíncronos
- La suspensión indefinida (`Thread.suspend`) y la finalización (`Thread.stop`) están en desuso (deprecated)
  - IMPORTANTE: La parada indefinida o terminante de un hilo sólo debe estar controlada por dicho hilo

Java Threads

- Hilos
- Sincronización
- Ejercicios
- FAQ

Sincronización

- Los hilos se comunican generalmente a través de campos y los objetos que tienen esos campos
  - Es una forma de comunicación eficiente
  - Pero puede plantear errores de interferencias entre hilos
- La sincronización es la herramienta para evitar este tipo de problemas, definiendo órdenes estrictos de ejecución

Sincronización

Interferencia entre hilos

```
class Counter
    {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
    }
```

c++ está compuesto de:

1. *Obtener el valor de c*
2. *Incrementar c en 1*
3. *Almacenar el valor de c*

c-- está compuesto de:

1. *Obtener el valor de c*
2. *Decrementar c en 1*
3. *Almacenar el valor de c*

En una situación extrema, dos hilos pueden estropearlo. P. ej., si A y B invocan concurrentemente a `increment` y `decrement`, puede ocurrir lo siguiente:

Hilo A: recuperar c (0)
Hilo B: recuperar c (0)
Hilo A: incrementar c (1)
Hilo B: decrementar c (-1)
Hilo A: almacenar c (1)
Hilo B: almacenar c (-1) > *¡c no vale lo mismo que al comienzo del incremento/decremento!*

Métodos sincronizados

Definición

- Dos invocaciones de ***métodos sincronizados*** del mismo objeto no se pueden mezclar.
  - Cuando un hilo ejecuta un método sincronizado de un objeto, todos los hilos que invoquen métodos sincronizados del objeto se bloquearán hasta que el primer hilo termine con el objeto.
    - Implementación de sección crítica en Java
  - Al terminar un método sincronizado, se garantiza que todos los hilos verán los cambios realizados sobre el objeto.

- Cuando un hilo invoca un método sincronizado, adquiere el ***bloqueo intrínseco*** del objeto correspondiente.
  - Si invoca un método estático sincronizado, adquiere el bloqueo intrínseco de la clase, independiente de los de sus objetos.

Métodos sincronizados

Implementación en Java

- Los métodos sincronizados se identifican mediante la palabra clave **synchronized**
  - Cualquier hilo que acceda a un método sincronizado de un objeto deberá obtener su bloqueo intrínseco de dicho objeto
  - Cualquier hilo que acceda a un método *estático* sincronizado de un objeto deberá obtener el bloqueo intrínseco de su *clase*

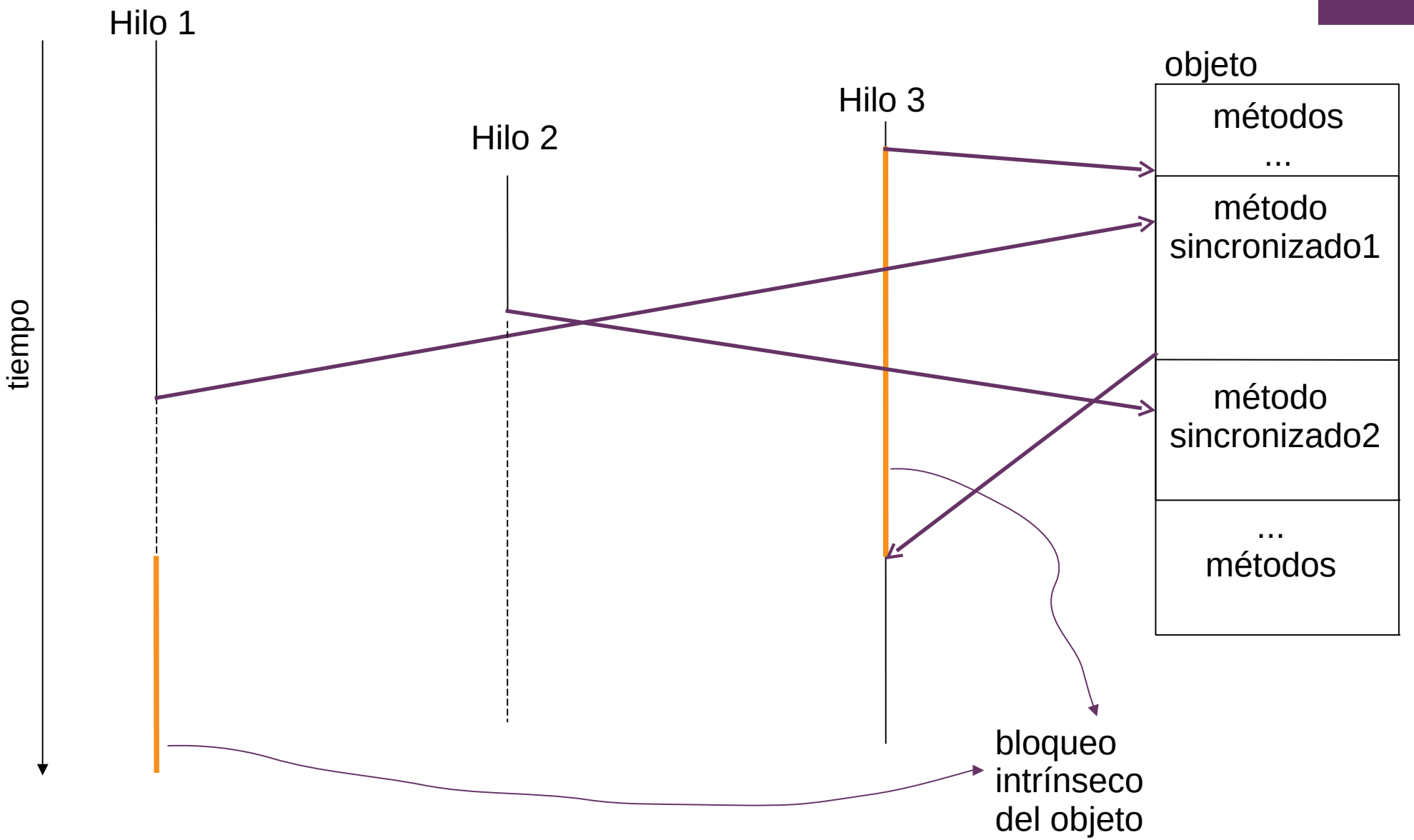
```
class Counter

{
private int c = 0;
public synchronized void increment() { c++; }
public synchronized void decrement() { c--; }
public int value() { return c; }

}
```

Métodos sincronizados

Ejemplo



Sincronización de código

Definición e implementación

- En vez de sincronizar todo un método podemos sincronizar una porción de código
  - Debemos asociar un atributo u objeto sobre el que se requiere el bloqueo intrínseco (testigo)

```
public void addName(String name)  {
        synchronized(this)
           {
           lastName = name;
           nameCount++;
           }
        nameList.add(name);
        }
```

Sincronizamos esta porción de código mediante el bloqueo intrínseco de este mismo objeto (**this**)

Mientras un hilo esté ejecutando este trozo de código, cualquier hilo que intente acceder a un trozo de código sincronizado asociado a este objeto, o a un método sincronizado de este objeto, se bloqueará

Sincronización de código

Testigos

- Podemos usar cualquier objeto como testigo
  - No sólo el objeto (`this`) o su clase (`this.getClass()`)
  - Permite un grado de sincronización más fino que los métodos sincronizados (que son sobre el objeto)
- Si el testigo es un atributo estático, se bloquea su clase

```
public class MsLunch
 {
 private long c1 = 0;
 private long c2 = 0;
 private Object lock1 = new Object();
 private Object lock2 = new Object();

 public void inc1()
  {
  synchronized(lock1) { c1++; }
  }

 public void inc2()
  {
  synchronized(lock2) { c2++; }
  }
}
```

Sincronización

wait y notify

- Los métodos sincronizados permiten definir secciones críticas, pero esto no siempre es suficiente (→ semáforos).
- `objeto.wait()` suspende la ejecución de un hilo hasta que recibe una notificación del objeto sobre el que espera
- El proceso que espera debe tener el bloqueo intrínseco del objeto que invoca a `wait()`
  - Si no, da un error (`IllegalMonitorStateException`)
  - Una vez invocado, el proceso suspende su ejecución y libera el bloqueo
  - `wait(int time)` espera sólo durante un tiempo
- `objeto.notify()`/`objeto.notifyAll()`informa a uno/todos los procesos que están esperando por `objeto` de que puede(n) continuar

Sincronización

```
public class SynchronizedPingPong extends Thread
{
 private String word;
 public SynchronizedPingPong(String s) {word=s;}

 public void run()
  {
   synchronized(getClass())
    {
     for (int i=0;i<3000;i++)
      {
       System.out.print(word);
       System.out.flush();
       getClass().notifyAll();
       try
        {getClass().wait();}
       catch (java.lang.InterruptedException e) {}
       }
     getClass().notifyAll();
     }
   }

 public static void main(String[] args)
   {
   SynchronizedPingPong tP=new SynchronizedPingPong("P");
   SynchronizedPingPong tp=new SynchronizedPingPong("p");
   tp.start();
   tP.start();
   }
}
```

"Para entrar por aquí tenemos que conseguir el bloqueo intrínseco de la clase SynchronizedPingPong"

Ejecuto una iteración

Aviso de que he terminado
Espero

Sincronización

Bloqueos intrínsecos: resumen

Sincronización	Obtiene bloqueo sobre
`synchronized metodo`	Objeto que contiene el método
`static synchronized metodo`	Clase que contiene el método
`synchronized(objeto)`	`objeto`
`synchronized(objeto.getClass())`	Clase instanciada por el objeto
`synchronized(objetoEstático)`	Clase instanciada por el objeto
Espera	**Requiere bloqueo sobre**
`objeto.wait()`	`objeto`
`objeto.getClass().wait()`	Clase instanciada por el objeto
`objetoEstático.wait()`	Clase instanciada por el objeto

Sincronización

Problemas a evitar

- **Espera ocupada:** un proceso espera por un recurso, pero la espera consume CPU
  - `while(!recurso) ;`
  - Se soluciona con el uso de `wait()`
- **Interbloqueo** (deadlock): varios procesos compiten por los mismos recursos pero ninguno los consigue
- **Inanición**: un proceso nunca obtiene los recursos que solicita, aunque no esté interbloqueado

  O los obtiene con mucha menor frecuencia que otros

Sincronización

Semáforos

- Alternativa a `wait/notify`
- `java.util.concurrent.Semaphore`
  - `acquire()` funciona de manera similar a `wait()`
  - `release()` funciona de manera similar a `notify()`
    - Pero no se pierde la notificación si ningún hilo estaba esperando (con `notify` sí)
  - El semáforo puede permitir más de un acceso (*permits*)
    - `acquire/release` pueden adquirir/liberar varios permits.

Listas y Mapas concurrentes

- Existen colecciones listas para funcionar en entornos concurrentes
- `Collections.synchronizedList(List<T> list)` toma una lista y la 'envuelve' en una lista que sólo permite a un hilo acceder concurrentemente (*thread-safe*)
- `ConcurrentHashMap<K,V>` permite definir diccionarios a prueba de hilos (thread-safe)
  - El paquete `java.concurrent` tiene muchos otros tipos de datos a prueba de hilos

Java Threads

- Hilos
- Sincronización
- Ejercicios
- FAQ

Ejercicio

Ping Pong

- Modificar el código de la clase `PingPong` para obtener la siguiente salida: `PpPpPpPpPp…`
- Se necesitará hacer uso de `synchronized`, `wait` y `notify`
  - Cuidado con el interbloqueo y la elección del testigo
  - O, alternativamente, se puede hacer uso de `Semaphore`

Ejercicio

Carrera 4x100

- Implementar una carrera por relevos:
  - Tenemos 4 `Atletas` dispuestos a correr
  - Tenemos una clase *principal* `Carrera`
  - Tenemos un objeto estático `testigo`*
  - Todos los atletas empiezan parados, uno comienza a correr (tarda entre 9 y 11s) y al terminar su carrera pasa el testigo a otro que comienza a correr, y así sucesivamente
  - Pistas:
    - `Thread.sleep` y `Math.random` para simular la carrera
    - `synchronized`, `wait` y `notify` para el paso del testigo
      - O utlizar un `Semaphore` como testigo
    - `System.currentTimeMillis` o `Calendar` para ver tiempos

* ¿Debe estar en la clase Carrera o Atleta?

Ejercicio

Carrera 100m lisos

- Implementar una carrera de 100m lisos:
  - Tenemos 8 `Atletas` dispuestos a correr
    - Cada uno tiene un atributo `dorsal`
  - Tenemos una clase *principal* `Carrera`
    - Indica el pistoletazo de salida y el resultado de la carrera
  - Todos los `Atletas` comienzan pero se quedan parados esperando el pistoletazo de salida
    - Luego comienzan a correr (tardan entre 9 y 11s)
    - Al llegar a meta notifican a la carrera su `dorsal` y terminan
  - La `Carrera` escribe *"preparados"* y espera 1s, luego escribe *"listos"* y espera 1s, finalmente escribe *"ya!"* y notifica a los hilos de los `Atletas`
    - Cada vez que un atleta le notifica su `dorsal`, escribe por pantalla: `dorsal+" tarda "+System.currentTimeMillis()`