Programación concurrente en Java.

Breve introducción

Miguel Ángel LATRE

Dept. de Informática e Ingeniería de Sistemas

Universidad Zaragoza

Concurrencia en Java

- Hilos de ejecución
  - Clase `Thread` e interfaz `Runnable`
  - Pausas
  - Interrupciones
  - Citas
- Monitores
  - Exclusión mutua
  - Métodos `wait()`, `notify()` y `notifyAll()`
- Algunas bibliotecas de Java útiles para concurrencia
  - Cierres de exclusión mutua
  - Semáforos, barreras
  - Colecciones (vectores, conjuntos, tablas, colas) concurrentes
  - Variables atómicas

Concurrencia en Java

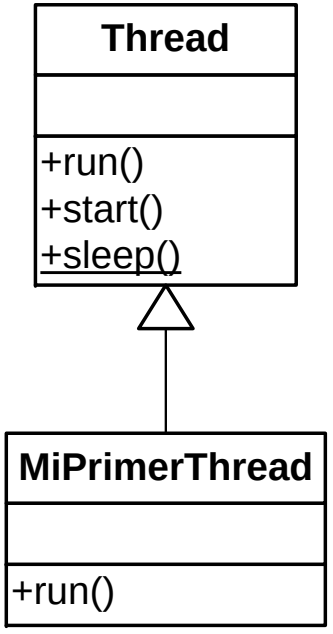
- **Hilos de ejecución**
  - **Clase `Thread` e interfaz `Runnable`**
  - **Pausas**
  - **Interrupciones**
  - **Citas**
- Monitores
  - Exclusión mutua
  - Métodos `wait()`, `notify()` y `notifyAll()`
- Algunas bibliotecas de Java útiles para concurrencia
  - Cierres de exclusión mutua
  - Semáforos, barreras
  - Colecciones (vectores, conjuntos, tablas, colas) concurrentes
  - Variables atómicas

Java

- Lenguaje orientado a objetos
- Incluye elementos para la programación concurrente
  - Clase `Thread` e interfaz `Runnable`
  - Cada objeto es un monitor
    - Métodos `wait()`, `notify()` y `notifyAll()`
  - Incluye una interfaz de programación de aplicaciones (API) de alto nivel
    - Cierres de exclusión mutua
    - Semáforos, barreras
    - Colecciones (vectores, conjuntos, tablas, colas) concurrentes
    - Variables atómicas

Hilos de ejecución

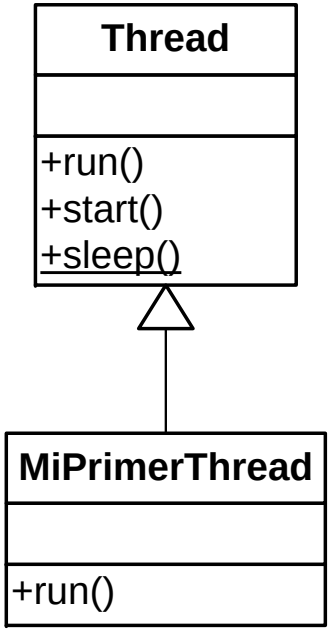
- Cada hilo de ejecución en Java asociado a un objeto de la clase `Thread`
  (http://download.oracle.com/javase/6/docs/api/java/lang/Thread.html)
- Dos posibilidades:
  - Heredando de la clase `Thread`
    - Reescribiendo el método `run()`
  - Implementando el interfaz `Runnable`
    - Ejecutándolo a través de un objeto de la clase `Thread`



Hilos de ejecución. Clase Thread

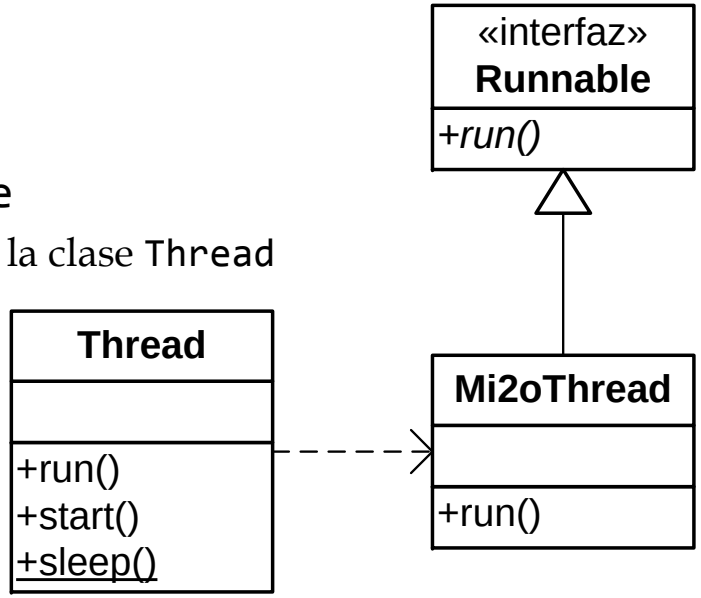
```
class MiPrimerThread extends Thread {
   public void run(){
       System.out.println("Hola mundo");
   }
}
```

```
class Main {
  public static void main(String[] args) {
    MiPrimerThread t = new MiPrimerThread();
    t.start();
  }
}
```



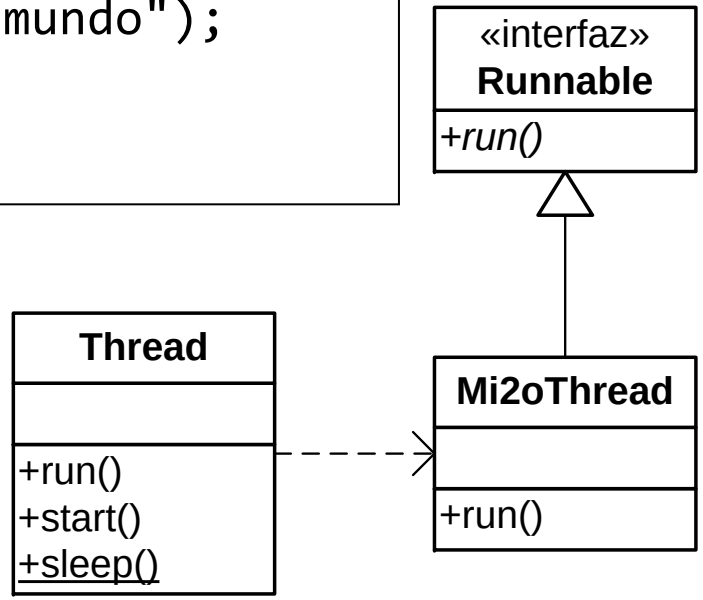
Hilos de ejecución. Interfaz `Runnable`

- Cada hilo de ejecución en Java asociado un objeto de la clase `Thread`
- Dos posibilidades:
  - Heredando de la clase `Thread`
    - Reescribiendo el método `run()`
  - Implementando el interfaz `Runnable`
    - Ejecutándolo a través de un objeto de la clase `Thread`



Hilos de ejecución. Interfaz `Runnable`

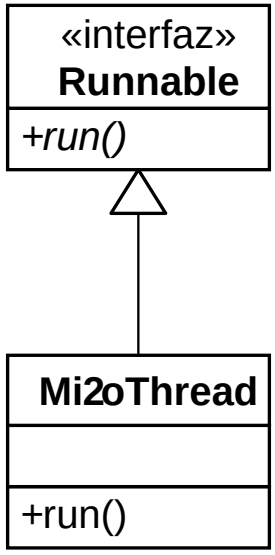
```
class Mi2oThread implements Runnable {
  public void run(){
      System.out.println("Hola mundo");
  }
}
```



Hilos de ejecución. Interfaz `Runnable`

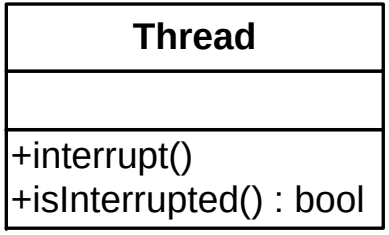
```
class Mi2oThread implements Runnable {
  public void run(){
      System.out.println("Hola mundo");
  }
}
```

```
class Main{
  public static void main(String[] args){
    Thread t = new Thread(new Mi2oThread());
    t.start();
  }
}
```



Hilos de ejecución. Interrupciones

- Una *interrupción* es una indicación a un hilo para que termine o cambie de actividad
- Su significado concreto no está definido por el lenguaje, depende de la aplicación
- Un hilo es interrumpido por otro cuando invoca su método `interrupt()`
- Un hilo puede saber si ha sido interrumpido invocando la función `isInterrupted()`, o porque ha capturado una excepción del tipo `InterruptedException`

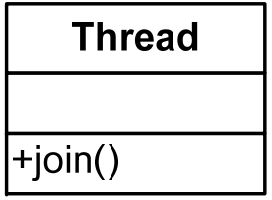


Hilos de ejecución. Pausas

- `Thread.sleep()` suspende temporalmente la ejecución del hilo que lo invoca
- Tiempo se proporciona en milisegundos o milisegundos y nanosegundos
  - `Thread.sleep(1000);`
  - `Thread.sleep(0, 20);`
- No se garantiza la precisión del tiempo de suspensión
- Puede ser inferior al especificado, si el hilo es *interrumpido*
- Puede lanzar la excepción `InterruptedException`
  - Otro hilo interrumpe al que ha invocado a `sleep()` mientras estaba suspendido

Hilos de ejecución. Citas

- El método `join()` permite a un hilo esperar a que otro termine su ejecución
- Es posible especificar un tiempo de espera máximo
  - `t.join();` `//`*Espera a que* `t` *termine*
  - `t.join(1000);` `//`*Espera 1 s como máximo a que* `t` *termine*
  - `t.join(0, 20);` `//`*Espera 20 ns como máximo a que* `t` *termine*
- `join()` responde a una interrupción durante la espera lanzando una `InterruptedException`



Hilos de ejecución. Ejemplo

```
class TareaTonta implements Runnable {

   String mensaje;

   TareaTonta(String mensaje) {
      this.mensaje = mensaje;
   }

   public void run() {

         for(int i=0; i<100; i++) {
            System.out.println(mensaje);
            Thread.sleep(100);
         }



   }
}
```

Hilos de ejecución. Ejemplo

```
class TareaTonta implements Runnable {

   String mensaje;

   TareaTonta(String mensaje) {
      this.mensaje = mensaje;
   }

   public void run() {
      try {
         for(int i=0; i<100; i++) {
            System.out.println(mensaje);
            Thread.sleep(100);
         }
      }
      catch (InterruptedException ignorar) {
      }
   }
}
```

Hilos de ejecución. Ejemplo

```
class TareaTontaMain {
   public static void main(String[] args)
                                    throws InterruptedException {

      Thread t1 = new Thread(new TareaTonta("Soy A"));
      Thread t2 = new Thread(new TareaTonta("Soy B"));
      Thread t3 = new Thread(new TareaTonta("Soy C"));

      t1.start();
      t2.start();
      t3.start();

      t1.join();
      t2.join();
      t3.join();

     System.out.println("Fin");
   }
}
```

Concurrencia en Java

- Hilos de ejecución
  - Clase `Thread` e interfaz `Runnable`
  - Pausas
  - Interrupciones
  - Citas
- **Monitores**
  - **Exclusión mutua**
  - **Métodos `wait()`, `notify()` y `notifyAll()`**
- Algunas bibliotecas de Java útiles para concurrencia
  - Cierres de exclusión mutua
  - Semáforos, barreras
  - Colecciones (vectores, conjuntos, tablas, colas) concurrentes
  - Variables atómicas

Exclusión mutua

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

   public Contador(int valorInicial) {
      c = valorInicial;
   }

   public void incrementar() {
      c++;
   }

   public void decrementar() {
      c--;
   }

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

Exclusión mutua

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

   public Contador(int valorInicial) {
      c = valorInicial;
   }

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

   public synchronized void decrementar() {
      c--;
   }

   public synchronized int valor() {
      return c;
   }
}
```

Métodos sincronizados

- Se marcan a través de la palabra reservada **`synchronized`**
- Se garantiza la exclusión mutua en la ejecución de métodos sincronizados de **un mismo** objeto
- Basado en la adquisición y liberación de un *cierre intrínseco* de cada objeto en Java
- El ámbito *sincronizado* puede ser menor que el de un método → *instrucciones sincronizadas*

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

Monitores en Java

- Todos los objetos en Java son un monitor
  - Tienen una única variable condición implícita
  - Las esperas se realizan invocando el método `wait()`
  - Los anuncios se realizan invocando a los métodos `notify()` o `notifyAll()`
  - `wait()`, `notify()` y `notifyAll()` deben ejecutarse en exclusión mutua (en métodos sincronizados)
    - Lanzan `IllegalMonitorStateException` en caso contrario

Monitores en Java

- `wait()` bloquea al hilo que lo invoca
  - hasta que el hilo sea interrumpido
  - hasta que otro hilo notifique que algún evento ha ocurrido (con `notify()` o `notifyAll()`)
    - Aunque no necesariamente el evento que espera el hilo que invocó **`wait()`**
    - También puede ocurrir que la condición que esperara haya dejado de cumplirse

```
if (!puedoSeguir) {
   wait();
}
```



```
while (!puedoSeguir) {
   wait();
}
```

- Al hacer `wait()`, se libera el cierre intrínseco

Monitores en Java

- `t.notifyAll()` desbloquea todos los hilos que se hayan bloqueado al invocar el método `wait()` del objeto `t`
- `t.notify()` desbloquea sólo a uno de los hilos que se hayan bloqueado al invocar `wait()` del objeto `t`
  - No se puede especificar cuál

Ejemplo. Buffer de un objeto

```
class Buffer {
   private Object dato = null;
   private boolean hayDato = false;

   public synchronized void poner(Object nuevoDato)
                                        throws InterruptedException {
      while(hayDato) {
         wait();
      }
      // !hayDato
      dato = nuevoDato;
      hayDato = true;
      notifyAll();
   }
...
```

Ejemplo. Buffer de un objeto

```
   ...
   public synchronized Object quitar() {
                       throws InterruptedException {
      while(!hayDato) {
         wait();
      }
      // hayDato
      notifyAll();
      hayDato = false;
      return dato;
   }
}
```

Concurrencia en Java

- Hilos de ejecución
  - Clase `Thread` e interfaz `Runnable`
  - Pausas
  - Interrupciones
  - Citas
- Monitores
  - Exclusión mutua
  - Métodos `wait()`, `notify()` y `notifyAll()`
- **Algunas bibliotecas de Java útiles para concurrencia**
  - **Cierres de exclusión mutua**
  - **Semáforos, barreras**
  - **Colecciones (vectores, conjuntos, tablas, colas) concurrentes**
  - **Variables atómicas**

Algunas bibliotecas de Java para concurrencia

- En
  - `java.util.concurrent`
  - `java.util.concurrent.atomic`
  - `java.util.concurrent.locks`
- Cierres de exclusión mutua
- Barreras, semáforos
- Colecciones (vectores, conjuntos, tablas, colas) concurrentes
- Variables atómicas

Cierres de exclusión mutua

- `java.util.concurrent.locks`
- Interfaces `Lock` y `Condition`
  - `Lock` permite gestionar los cierres de forma explícita
    - `lock()`
    - `unlock()`
    - `tryLock()`
    - `newCondition()`
  - `Condition` permite establecer más de una cola asociada a un cierre
    - `await()`
    - `awaitUntil()`
    - `signal()`
    - `signalAll()`

Colecciones concurrentes

- `BlockingQueue` define una estructura FIFO que se bloquea (o espera un tiempo máximo determinado) cuando se intenta añadir un elemento a una cola llena o retirar uno de una cola vacía
- `ConcurrentMap` define una tabla *hash* con operaciones atómicas

Variables atómicas

- `java.util.concurrent.atomic`
- Conjunto de clases que permiten realizar operaciones atómicas con variables de tipos básicos
  - `AtomicBoolean`
  - `AtomicInteger`
  - `AtomicIntegerArray`
  - `AtomicLong`
  - `AtomicLongArray`
  - …

Variables atómicas

- `AtomicInteger`
  - `addAndGet`
  - `compareAndSet`
  - `decrementAndGet`
  - `getAndAdd`
  - `getAndDecrement`
  - `getAndIncrement`
  - `getAndSet`
  - `incrementAndGet`
  - `intValue`
  - `lazySet`
  - `set`

Otras estructuras concurrentes

- CyclicBarrier
  - CyclicBarrier(**int** parties, Runnable barrierAction)
  - await()
  - await(**long** timeout, TimeUnit unit)
  - getNumberWaiting()
  - getParties()
  - isBroken()
  - reset()
  - …

Otras estructuras concurrentes

- Semaphore
  - Semaphore(**int** permits)
  - acquire(), acquire(**int** permits)
  - acquireUninterruptibly(),
    acquireUninterruptibly(**int** permits)
  - availablePermits()
  - drainPermits()
  - getQueueLength()
  - release(), release(**int** permits)
  - tryAcquire(), tryAcquire(**int** permits),
    tryAcquire(**int** permits, **long** timeout, TimeUnit unit),
    tryAcquire(**long** timeout, TimeUnit unit)