PROGRAMACION CONCURRENTE Y DISTRIBUIDA

IV.2 Tareas periódicas: Timer, TimerTask y ScheduledThreadPoolExecutor classes

J.M. Drake
L.Barros

Notas:

Esperas temporizadas

- Las sentencias básicas de temporización en Java son las esperas temporizadas:
- De la clase **Thread**:

public static void **sleep**(long millis) **throws** InterruptedException;

public static void **sleep**(long millis, int nanos)

throws InterruptedException;

- De la clase **Object**:

public final void **wait**(long timeout) **throws** InterruptedException;

public final void **wait**(long timeout, int nanos)

throws InterruptedEsception;

- La temporización hace referencia a una espera fijada en tiempo relativo

Notas:

Tarea periódica basada en esperas temporizadas (1)

```
public class PeriodicTask {
  Thread thread;
   public interface Task{ public void oper(); }
   public void periodicExecution(final Task task, final int periodIn_ms){
      thread=new Thread( new Runnable(){
           public void run(){
              while(!Thread.interrupted()){
                   task.oper();
                   try{ Thread.sleep(periodIn_ms);
                   }catch(InterruptedException ie){ Thread.currentThread().interrupt();};
              }
           }
      });
      thread.start();
   }
   public void cancel(){Thread.interrupt();}
}
```


Notas:

Tarea periódica basada en esperas temporizadas (2)

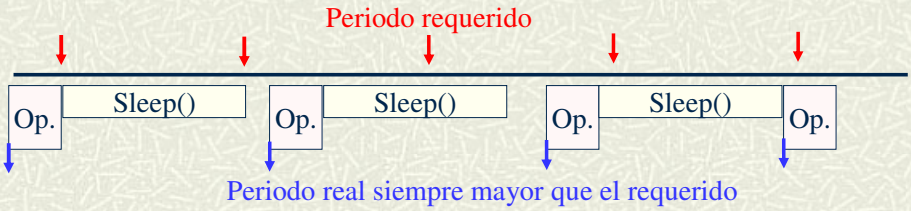
```
public static void main(String[] args) {
    //Define la tarea
    final class Print implements Task{
        public void oper(){
            System.out.println("Nueva Ejecución: "+ System.currentTimeMillis());}
    }
    Print p=new Print();
    // Define la tarea periodica que ejecuta la tarea
    PeriodicTask pt= new PeriodicTask();
    pt.periodicExecution(new Print(), 1000);
    // Espera un tiempo
    try{ Thread.sleep(10000);}
    catch (InterruptedException e){};
    // Cancela la ejecución
    pt.cancel();
}
```

```
Nueva Ejecución: 1194377159484
Nueva Ejecución: 1194377160484
Nueva Ejecución: 1194377161484
Nueva Ejecución: 1194377162484
Nueva Ejecución: 1194377163484
Nueva Ejecución: 1194377164484
Nueva Ejecución: 1194377165500
Nueva Ejecución: 1194377166500
Nueva Ejecución: 1194377167500
Nueva Ejecución: 1194377168500
Nueva Ejecución: 1194377169500
```

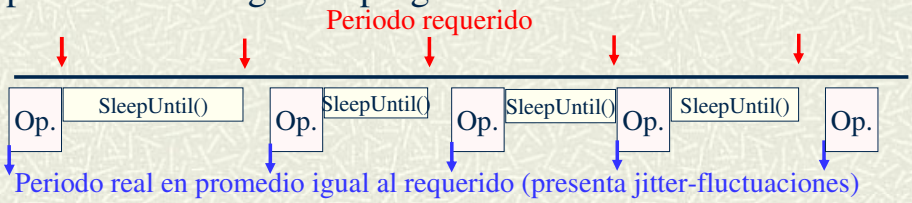

Notas:

Problema de las tareas periodicas basada en sleep

- Si se basa en un tiempo relativo, su periodo es siempre superior al previsto:



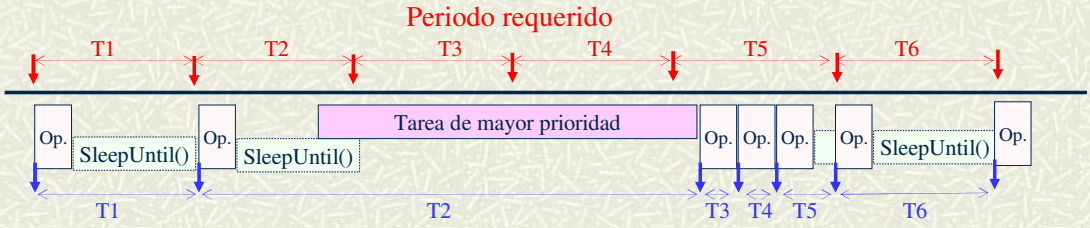
- Las tareas basadas en sleep until un tiempo absoluto tienen un periodo medio igual al programado:



Notas:

Problema de tareas basadas en tiempos absolutos

- Las tareas periódicas basadas en tiempos absolutos pueden presentar altos jitters:



La tarea periódica presenta un alto jitter

Notas:

Tarea periódica basada en plazoabsolutos

```
public class PeriodicTask {
  Thread thread;
  long nextExecution;
   public interface Task{ public void oper(); }
   public void periodicExecution(final Task task, final int periodIn_ms){
       thread=new Thread( new Runnable(){
            public void run(){
                nextExecution= System.currentTimeMillis();
                while(!Thread.interrupted()){
                     nextExecution= nextExecution+periodIn_ms;
                    task.oper();
                    try{ Thread.sleep(nextExecution-System.currentTimeMillis());
                    }catch(InterruptedException ie){ Thread.currentThread().interrupt();};
       }   }   });
       thread.start();
   }.....
```


Notas:

Clase Timer

- Es un recurso activo que permite programar tareas:
  - para que se ejecuten en un tiempo futuro.
  - o para que se ejecuten periódicamente.
- Se basa en un único Thread de soporte, dentro del que se ejecutan todas las tareas que se le programan, secuencialmente. Si la ejecución de una tarea tarda más de lo previsto, las siguientes pueden retrasarse.
- Tiene un diseño muy eficiente, y puede gestionar la ejecución de miles de tareas.
- Es seguro para ser accedido por múltiples thread, sin requerir una sincronización específica.
- No ofrece garantías de tiempo real estricto, ya que se basa en sentencias del tipo Object.wait(long).

Notas:

Constructores de la clase Timer

- Timer()

  //Crea un nuevo Timer

- Timer(boolean isDaemon)

  // Crea un nuevo timer y declara el thread interno com daemon si el parámetro es true.

- Timer(String name)

  // Crea un nuevo timer y le asocia a su thread interno el nombre que se especifica.

- Timer(String name, boolean isDaemon)

  // Crea un nuevo timer con nombre y control de finalización como daemon o no.

Notas:

El Timer desaparece cuando ya no hay threads de usuario

Métodos de la clase Timer (1)

void **cancel**() *//Terminates this timer, discarding any currently scheduled tasks.*

int **purge**() *//Removes all cancelled tasks from this timer's task queue.*

void **schedule**(TimerTask task, Date time)

*// Programa la tarea para la ejecución **una sóla vez** y en el tiempo indicado.*

void **schedule**(TimerTask task, Date firstTime, long period)

*// Programa la tarea para la ejecución a partir de **firstTime** y luego*
*// **periódicamente** cada period ms.*

void **schedule**(TimerTask task, long delay)

*// Programa la tarea para la ejecución **una sóla vez** tras el **retraso** especificado.*

void **schedule**(TimerTask task, long delay, long period)

*// Programa la tarea para la ejecución tras un retraso **delay** y luego*
*// **periódicamente** cada periodo ms.*

Notas:

Métodos de la clase Timer (2)

void **scheduleAtFixedRate**(TimerTask task, Date firstTime, long period)

// Programa la tarea para la ejecución con programación en instante fijo hasta
*// que se cancele , empezando en **firstTime** y ejecutándose cada period ms.*

void **scheduleAtFixedDelay**(TimerTask task, long delay, long period)

// Programa la tarea para la ejecución con programación en instante fijo hasta
*// que se cancele , tras un retraso **delay** y ejecutándose cada period ms.*

Notas:

Ejemplo I: Reloj segundero de pantalla

- Ejemplo: queremos pintar un reloj en pantalla, cada segundo debemos mover o repintar el segundero.
- Solución: a la clase Timer le decimos cuando queremos el aviso (por ejemplo, un aviso cada segundo en el caso del reloj) y ella se encarga de llamar a un método que nosotros hayamos implementado.
- Resultado:El resultado es que ese método (el que pinta el segundero en la pantalla), se llamará cada cierto tiempo (una vez por segundo en el caso del reloj).
  - Scheduled():
    - Problema: es posible que el ordenador esté ocupado haciendo otras cosas, con lo que el aviso (pintado) nos puede llegar con un cierto retraso.
    - Con esta opción el retraso se acumula de una llamada a otra.
    - Si el ordenador está muy atareado y nos da avisos cada 1.1 segundos en vez de cada 1, el primer aviso lo recibimos en el segundo 1.1, el segundo en el 2.2, el tercero en el 3.3, etc, etc.
    - Si hacemos nuestro reloj de esta forma, cogerá adelantos o retrasos importantes en poco tiempo.

Notas:

Ejemplo II: Reloj segundero de pantalla

- **scheduleAtFixedRate()**:
  - Con estos métodos los avisos son relativos al primer aviso, de esta forma, si hay retraso en un aviso, no influye en cuando se produce el siguiente.
  - Igual que antes, si el ordenador está muy ocupado y da avisos cada 1.1 segundos en vez de cada segundo, el primer aviso se recibirá en el segundo 1.1, el segundo en el 2.1, el tercero en el 3.1, etc.
  - El retraso no se va acumulando.
- Está claro que para hacer un reloj, como es nuestro ejemplo, debemos usar la segunda forma (métodos **scheduleAtFixedRate()** en vez de **schedule()**).

Notas:

Clase TimerTask

- Clase abstracta que define una tarea que puede ser planificada para la ejecución en un timer, bien una sóla vez o de forma repetida.
- Constructor:

  protected **TimerTask**() *// Creates a new timer task.*

- Métodos:

  boolean **cancel**() *// Cancels this timer task.*

  abstract void ***run****()* *// The action to be performed by this timer task.*

  long ***scheduledExecutionTime****()*

  // Returns the scheduled execution time of the most recent actual execution of this task.

Notas:

Ejemplo de uso de scheduledExecutionTime() method

```
Public class MyTimerTask extends TimerTask{
    public void run(){
        ...
        if (System.currentTimeMillis-scheduledExecutionTime()>5000){
            // Ya han transcurrido mas de 5 segundos desde que se ejecutó
            // ínicio la ejecución y debe terminarse
            return;
        }
        ...
}
```


Notas:

Formas de finalización de un Timer

- Invocando cancel sobre el timer.
- Haciendo del Timer un *demonio* mediante new Timer(true). Si ma única hebra que abandona el programa es *daemon*, el programa finaliza.
- Una vez terminadas todas las tareas configuradas, se borran todas las referencias del objeto Timer.
- Invocando el método de sistema System.exit, que hace que todo el programa (con todas sus hebras) finalice.

Notas:

Ejemplo de planificación de tarea con Timer

```
import java.util.Timer;
import java.util.TimerTask;
//Simple demo that uses java.util.Timer to schedule task to execute once 5 seconds have passed.

public class Reminder {
        Timer timer;
        public Reminder(int seconds) {
                timer = new Timer();
                timer.schedule(new RemindTask(), seconds*1000);
}

class RemindTask extends TimerTask {
        public void run() {
                System.out.println("Time's up!");
                timer.cancel(); //Terminate the timer thread
        }

}
public static void main(String args[]) {
                System.out.println("About to schedule task.");
                new Reminder(5);
                System.out.println("Task scheduled.");
        }
}
```


Notas:

Las fases que se siguen para esta implementación las podemos enumerar como sigue:

Se crea una hebra instanciando de la clase Timer.

Con el método schedule se configura la planificación, instanciando el objeto TimerTask (new RemindTask()) e introduciendo la temporización que establece un retardo de 5000 ms.

En el método run() se establece la acción a tomar cumplido el retardo.

Ejemplo de la planificación de una tarea periódica I

```
import java.util.TimerTask;
import java.util.date;
public class DisplayMemoria extends TimeTask{
  public void run(){
      System.out.println(new Date() + ": " );
      Runtime rt= Runtime.getRunTime();
      System.out.println(rt.freememory()+" libre,");
      System.out.println(rt.totalMemory() + " total");
      System.out.println();
  }
}
```

```
...
long inicio= System.currentTimeMillis();
Timer temporizador=new Timer(true);//daemon
temporizador.scheduleAtFixedRate(new DisplayMemoria(), 0, 1000);
...
```


Notas:

La arquitectura del entorno de ejecución de programas Java se basa en una máquina virtual, que se ejecuta en el computador para interpretar las instrucciones del programa del usuario descritas en un código intermedio especial, llamado código de máquina virtual Java (Java Byte Code).

La idea principal de esta arquitectura es la de "Escribir una vez, ejecutar en cualquier sitio". El compilador de Java no genera código máquina de un computador concreto, sino un código especial, que luego es interpretado por otro programa, llamado máquina virtual, que existe en cada computador en el que se desea ejecutar el programa Java. De este modo, un programa Java se puede ejecutar indistintamente en cualquier computador que disponga de esa máquina virtual, sin necesidad de recompilarlo.

Adicionalmente, en la arquitectura Java los programas no se enlazan antes de su ejecución, sino que se utiliza un enlazado dinámico. Cuando se hace una llamada a una operación de un módulo (clase) que no está cargado en la máquina virtual, ésta se encarga de buscar ese módulo y cargarlo en ese momento en la máquina virtual.

Desde el programa del usuario se pueden utilizar operaciones "nativas", suministradas por la máquina virtual, escritas generalmente en código máquina, y que pueden acceder a los dispositivos hardware del computador. El resultado es: programas muy portables, muy dinámicos, aunque poco eficientes. Existen también compiladores Java que traducen directamente a lenguaje máquina.

Ejemplo de la planificación de una tarea periódica II

```
import java.util.Timer;
import java.util.TimerTask;
 import java.util.Calendar;
 import java.util.GregorianCalendar;
 import java.util.Date;

public final class FetchMail extends TimerTask {
/** * Implements TimerTask's abstract run method. */
 public void run(){
System.out.println("Fetching mail...");
 }
// PRIVATE
//expressed in milliseconds
private final static long fONCE_PER_DAY =
      1000*60*60*24;
private final static int fONE_DAY = 1;
private final static int fFOUR_AM = 4;
private final static int fZERO_MINUTES = 0;
```

```
private static Date getTomorrowMorning4am(){
Calendar tomorrow = new GregorianCalendar();
      tomorrow.add(Calendar.DATE, fONE_DAY);
Calendar result = new GregorianCalendar(
      tomorrow.get(Calendar.YEAR),
      tomorrow.get(Calendar.MONTH),
      tomorrow.get(Calendar.DATE), fFOUR_AM,
      fZERO_MINUTES );
return result.getTime();
}

/** * Construct and use a TimerTask and Timer. */
public static void main (String... arguments ) {

TimerTask fetchMail = new FetchMail(); //perform the task
      once a day at 4 a.m., starting tomorrow morning
Timer timer = new Timer();
timer.scheduleAtFixedRate(fetchMail,
      getTomorrowMorning4am(), fONCE_PER_DAY);
}
}
```

Notas:

Performed once a day at 4 a.m., starting tomorrow morning.

Prioridades en Timer

- El thread del Timer se crea con la prioridad del thread que lo crea.
- Si se quiere incrementar la prioridad del thread del timer, hay que crear el timer desde un thread con la prioridad deseada.
- Si se quiere que una tarea se realice con prioridad diferente, debe cambiarse la prioridad en el método run(), utilizando los métodos currentThread().setPriority(int newPriority).

Notas:

Problemas de la clase Timer.

- En Java 1.5 la pareja Timer –TimerClass han sido sustituidos por **ScheduledThreadPoolExecutor** por estos problemas:
  - Un Timer tiene un único thread . y puede presentar problemas de planificación de la ejecución de múltiples tareas.
  - Las tareas que ejecuta un Timer deben ser declaradas como extensión de la clase TimerTask, lo que elimina la posibilidad de heredar de otra clase del dominio.
  - La tarea que ejecuta un Timer debe ser implementada por un método run(), el cual se le puede pasar parámetros a través del constructor, pero ni puede retornar valores, ni puede gestionar excepciones.
- ScheduledThreadPoolExecutor resuelve estos tres problemas, por lo que actualmente se puede considerar que los Timer son una construcción obsoleta.

Notas:

ScheduledThreadPoolExecutor

- Permite planificar la ejecución de tareas después de un retraso o periódicamente, utilizando las capacidades de los ThreadPoolExecutor:
  - Usa un threadpool para ejecutar las tareas, y en el constructor se puede establecer el número de thread que se van a utilizar.
  - Las tareas que definen las tareas sólo deben implementar la interfaz Runnable, quedando libre de heredar de cualquier clase del dominio.
  - Trabaja también con objetos que implementa la interfaz Callable, que puede retornar resultados y gestionar excepciones.
- Las tareas se ejecutan tan pronto como son habilitadas, pero sin garantías de tiempo real de cuando se ejecutan.
- Las tareas planificadas para un mismo instante de tiempo son ejecutadas en el orden FIFO en que fueron ordenadas su ejecución.

Notas:

Constructores de la clase ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor (int corePoolSize)
*// Creates a new ScheduledThreadPoolExecutor with the given **core pool size**.*

ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler)
// Creates a new ScheduledThreadPoolExecutor with the given initial parameters.

ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory)
// Creates a new ScheduledThreadPoolExecutor with the given initial parameters.

ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory, RejectedExecutionHandler handler)
// Creates a new ScheduledThreadPoolExecutor with the given initial parameters.

corePoolSize: the number of threads to keep in the pool, even if they are idle.
handler: the handler to use when execution is blocked because the thread bounds and queue capacities are reached.
threadFactory: the factory to use when the executor creates a new thread.

Notas:

Métodos de SchedulerThreadPoolExecutor class (1)

void **execute**(Runnable command)
// Execute command with zero required delay.

*boolean **getContinueExistingPeriodicTasksAfterShutdownPolicy**()*
// Get the policy on whether to continue executing existing periodic tasks even when
// this executor has been shutdown.

*boolean **getExecuteExistingDelayedTasksAfterShutdownPolicy**()*
// Get policy on whether to execute existing delayed tasks even when this executor
// has been shutdown.

*BlockingQueue<Runnable> **getQueue**()*
// Returns the task queue used by this executor

*boolean **remove**(Runnable task)*
// Removes this task from the executor's internal queue if it is present, thus causing it
// not to be run if it has not already started

*<V> ScheduledFuture<V>**schedule**(Callable<V> callable, long delay, TimeUnit unit)*
// Creates and executes a ScheduledFuture that becomes enabled after the given delay

*ScheduledFuture<?>**schedule**(Runnable command, long delay, TimeUnit unit)*
// Creates and executes a one-shot action that becomes enabled after the given delay

Notas:

Métodos de SchedulerThreadPoolExecutor class (2)

ScheduledFuture<?> **scheduleAtFixedRate**(Runnable command,
long initialDelay, long period, TimeUnit unit)
// Creates and executes a periodic action that becomes enabled first after the
// given initial delay, and subsequently with the given period; that is
// executions will commence after initialDelay then initialDelay+period, then
*// initialDelay + 2 * period, and so.*

ScheduledFuture<?> ***scheduleWithFixedDelay****(Runnable command,*
long initialDelay, long delay, TimeUnit unit)
// Creates and executes a periodic action that becomes enabled first after the
// given initial delay, and subsequently with the given delay between the
// termination of one execution and the commencement of the next

void ***setContinueExistingPeriodicTasksAfterShutdownPolicy*** *(boolean value)*
// Set policy on whether to continue executing existing periodic tasks even
// when this executor has been shutdown

void ***setExecuteExistingDelayedTasksAfterShutdownPolicy****(boolean value)*
//Set policy on whether to execute existing delayed tasks even when this
// executor has been shutdown.

Notas:

Métodos de SchedulerThreadPoolExecutor class (3)

void **shutdown**()

// Initiates an orderly shutdown in which previously submitted tasks are executed,
// but no new tasks will be accepteda

List<Runnable> **shutdownNow**()

// Attempts to stop all actively executing tasks, halts the processing of waiting tasks,
// and returns a list of the tasks that were awaiting execution

*<T> Future<T>**submit**(Callable<T> task)*

// Submits a value-returning task for execution and returns a Future representing
// the pending results of the task.

*Future<?> **submit**(Runnable task)*

//Submits a Runnable task for execution and returns a Future representing that task

*<T> Future<T>**submit**(Runnable task, T result)*

// Submits a Runnable task for execution and returns a Future representing that task
// that will upon completion return the given result

Notas:

Interfaz SchedulerFuture<V>

- Es implementada por los objetos que retornan los planificadores de las tareas para su control posterior y para recuperar los resultados que genere:

public interface ScheduledFuture<V>
extends Delayed, Future<V>

- **Métodos que ofrece:**
  - **boolean cancel(boolean mayInterruptIfRunning)**
    // Attempts to cancel execution of this task.
  - **get()**
    // Waits if necessary for the computation to complete, and then retrieves its result.
  - boolean **isCancelled**()
    //Returns true if this task was cancelled before it completed normally.
  - boolean **isDone**()
    // Returns true if this task completed.

Notas:

Métodos de SchedulerThreadPoolExecutor class (3)

Heredados de **ThreadPoolExecutor:**

afterExecute, awaitTermination, beforeExecute,
finalize, getActiveCount,
getCompletedTaskCount, getCorePoolSize, getKeepAliveTime,
getLargestPoolSize, getMaximumPoolSize, getPoolSize,
getRejectedExecutionHandler, getTaskCount,
getThreadFactory, isShutdown, isTerminated,
isTerminating, prestartAllCoreThreads, prestartCoreThread,
purge, setCorePoolSize, setKeepAliveTime,
setMaximumPoolSize, setRejectedExecutionHandler,
setThreadFactory, terminated

Notas:

Ejemplo I:ScheduledExecutorService

```
ScheduledExecutorService sched =Executors.newSingleThreadScheduledExecutor();

public void runTwiceAnHour(Runnable, rTask, int howLong) {

    //Schedule rTask to run every 30 mins
    final ScheduledFuture<?> rTaskFuture = sched.scheduleAtFixedRate(rTask, 0,
    30, MINUTES);

    //Cancel the job after a user specified interval
    sched.schedule(new Runnable {public void run { rTaskFuture.cancel(true); } },
                   howLong,
                   HOURS);
}
```


Notas:

Ejemplo II: ScheduledExecutorService

```
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledFuture;
import static java.util.concurrent.TimeUnit.*;

class BeeperControl {

        private final ScheduledExecutorService scheduler =
        Executors.newScheduledThreadPool(1);

        public void beepForAMinute() {

                final Runnable beeper = new Runnable() {
                public void run() {
                                System.out.println("beep");
                }
        };

final ScheduledFuture<?> future =
        scheduler.scheduleAtFixedRate(beeper, 250, 250,
        MILLISECONDS);

scheduler.schedule(
        new Runnable() {
                public void run(){
                future.cancel(true);
                }
        }, 3, SECONDS
);
```

```
while (!future.isDone()) {
    try {
            Thread.sleep(10);
    } catch(InterruptedException e) {}
}
scheduler.shutdown();
}

public static void main(String[] args)
{
    BeeperControl bc = new
    BeeperControl();
    bc.beepForAMinute();
}

}
```


Notas: