Programación Concurrente y de Tiempo Real

Guión de prácticas 5: Creación y control de Hilos en Java (2): control y planificación

Natalia Partera Jaime
Alumna colaboradora de la asignatura

Índice

1. Introducción

En el guión anterior vimos cómo crear hilos de distintas maneras y algunos métodos para su ejecución y control. En este guión profundizaremos en los métodos para controlar hilos y programar su ejecución.

Antes de ver cómo controlar los hilos, es conveniente recordar cómo es el ciclo de vida de los hilos:

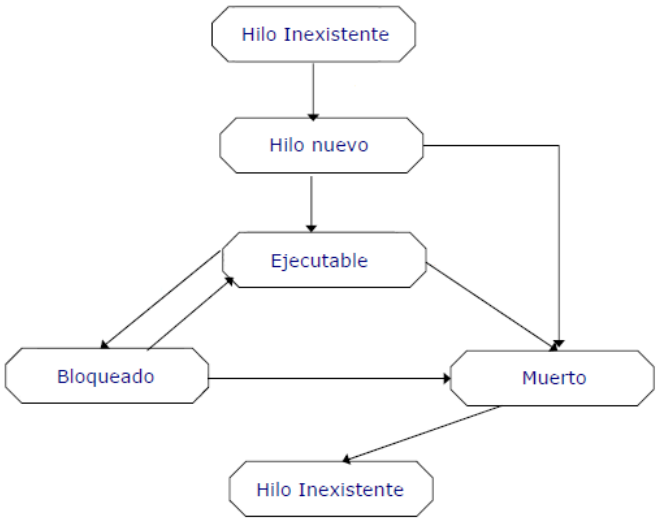

Figura 1: Gráfico del ciclo de vida de los hilos.

Al principio un hilo es creado, luego comienza a ejecutar su acción y en algúm momento muere, ya sea una vez que ha terminado su ejecución o antes. Para controlar estas acciones, cada técnica de creación de hilos puede usar unos métodos u otros. Los vemos a continuación según la técnica de creación de hilos.

2. Clase `Thread`

Cuando creamos los hilos usando la clase `Thread` utilizamos los métodos que la propia clase nos ofrece. Por ejemplo, en el diagrama anterior usaríamos estos métodos:

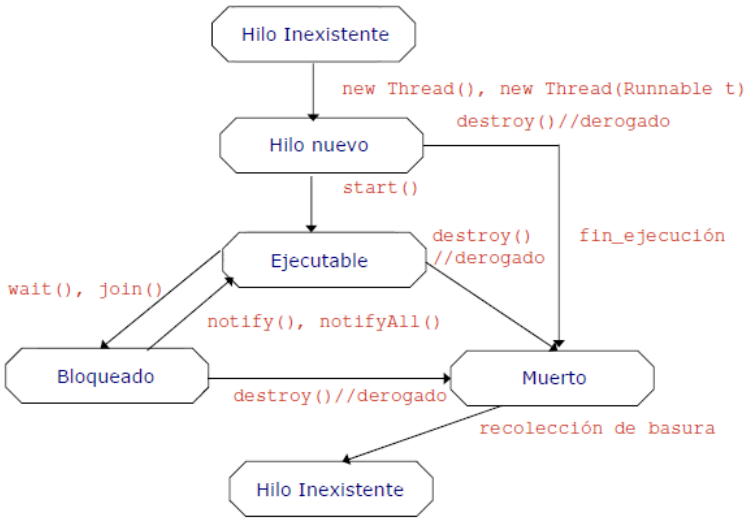

Figura 2: Gráfico del ciclo de vida de los objetos de la clase Thread.

Aunque en el guión anterior ya se explicaron algunos de los métodos más comunes de la clase `Thread`, ampliamos con algunos métodos más. Veamos a continuación un cuadro resumen de dichos métodos ofrecidos por la clase `Thread`:

Método	**Comportamiento**
`void checkAccess()`	Determina si el hilo que está corriendo actualmente tiene permiso para modificar el hilo que invoca al método.
`void destroy()`	**Método derogado**. Este método se usaba para matar un hilo.
`void interrupt()`	Interrumpe al hilo que lo invoca, siempre que sea posible.
`boolean isAlive()`	Indica si el hilo que lo invoca está vivo.
`void join()`	Espera a que muera el hilo que lo invoca.
`void resume()`	**Método derogado**. Este método reanudaba un hilo que había sido suspendido con `suspend()`, haciéndolo pasar de estado bloqueado a estado listo.
`static void sleep(long millis)`	Duerme al hilo que lo invoca, o hace que cese su ejecución, durante el número de milisegundos indicados.
`void start()`	Lanza al hilo que lo invoca para que comience su ejecución.
`void stop()`	**Método derogado**. Este método forzaba al hilo que lo llamaba a que parase su ejecución.
`void suspend()`	**Método derogado**. Este método suspendía al hilo que lo invocaba.
`static void yield()`	Indica al controlador que el hilo que invoca al método está dispuesto a ceder su uso del procesador.

Cuadro 1: Principales métodos de la clase `Thread`.

A pesar de que algunos de los métodos de la tabla anterior están derogados, es conveniente que conozca cómo pueden ser usados. Cabe destacar que el método `destroy()` no tiene implementación, por lo que antes de usarlo debería sobrecargarlo en la clase que herede a `Thread`. A continuación puede ver un ejemplo de uso de métodos derogados:

```
/**
 * Clase Task que representa una tarea que puede ser ejecutada concurrentemente.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.lang.Thread.*;

public class Task extends Thread {
  private int id;

  public Task() {}
  public Task(int id) {
    this.id = id;
  }

  public void run() {
    System.out.println("Ejecutando tarea " + id);
    for(int i = 0; i < 100; ++i) {
      System.out.println("Tarea " + id + ": vuelta" + i);
    }
    System.out.println("Fin de la tarea " + id);
  }
}
```

La clase anterior es utilizada en el siguiente programa que invoca a métodos derogados:

```
/**
 * Programa que utiliza métodos derogados de la clase Thread.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaThreadDerogados extends Thread {

  public static void main(String[] args) throws InterruptedException {
    Task t1 = new Task(1);
    Task t2 = new Task(2);
    Task t3 = new Task(3);

    //Lanzamos t1 y t2
    System.out.println("Lanzamos t1.");
    t1.start();
    System.out.println("Lanzamos t2.");
    t2.start();
```

```
    //Dormimos 10 milisegundos
    System.out.println("Dormimos 10 milisegundos");
    sleep(10);
    //Suspendemos t1
    System.out.println("Suspendemos t1.");
    t1.suspend();
    //Lanzamos t3
    System.out.println("Lanzamos t3.");
    t3.start();
    //Dormimos 10 miliegundos
    System.out.println("Dormimos 10 milisegundos");
    sleep(10);
    //Despertamos a t1
    System.out.println("Despertamos a t1");
    t1.resume();
    //Paramos t3
    System.out.println("Paramos t3.");
    t3.stop();
  }
}
```

Ejercicio 1 Compile las dos clases anteriores y ejecútelas. Observe el comportamiento de los hilos del programa.

A continuación podemos ver otro programa de ejemplo que utiliza sólo métodos de la clase `Thread` que siguen vigentes (no derogados).

```
/**
 * Programa que utiliza métodos válidos de la clase Thread.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.io.*;

public class UsaThreadActuales {

  public static void main(String[] args) throws InterruptedException {
    Task t1 = new Task(1);
    Task t2 = new Task(2);
    Task t3 = new Task(3);

    //Lanzamos t1, t2 y t3
    System.out.println("Lanzamos t1.");
    t1.start();
    System.out.println("Lanzamos t2.");
    t2.start();
    System.out.println("Lanzamos t3.");
```

```
    t3.start();

    while(t3.isAlive()) {
      System.out.println("t3 está vivo, por lo que t1 le cede su uso del
        procesador.");
      t1.yield();
    }
    //Dormimos al hilo t2 durante 2 segundos
    t2.sleep(2000);
    //Interrumpimos al hilo t3 siempre que sea posible
    t3.interrupt();
   //Esperamos a que todos los hilos terminen su ejecución
    t1.join();
    t2.join();
    t3.join();
  }
}
```

Ejercicio 2 Modifique el ejemplo anterior usando sólo los métodos permitidos actualmente de la clase `Thread`. Su programa debe tener 3 hilos, y procurará que el orden de terminación de los hilos sea "tercer hilo - segundo hilo - primer hilo". Juegue con la cesión del uso del procesador para conseguirlo. Cuando lo termine, compruebe su ejercicio con la posible solucion qie se muestra en 7.1.

2.1. Planificación básica de hilos

En la tabla anterior, observamos que con el método `yield()` podemos ceder prioridad de uso del procesador. Pero además de este método, también es posible modificar la prioridad de los hilos para forzar que sean ejecutados en el orden que deseamos.

Hay que tener en cuenta que el concepto de prioridad tiene sentido exclusivamente en el ámbito de la máquina virtual de Java. Aunque la JVM se encarga de mapear los hilos a los hilos del sistema, este mapeo no es riguroso, ya que puede producir inversiones de prioridad.

Por defecto, la prioridad del hilo hijo es igual a la prioridad del hilo padre que lo invocó. La clase `Thread` tiene un esquema de 10 niveles de prioridad, con valores comprendidos entre 1 y 10, donde tiene preferencia el hilo que tenga mayor prioridad. Además, la clase dispone de dos métodos sobre la prioridad: uno observador `public int getPriority()` para obtener la prioridad de un hilo, y otro modificador `public void setPriority(int p)` para modificar la prioridad del hilo. A continuación se muestran los elementos de la clase `Thread` que sirven para el control de la prioridad:

```
package java.lang;

public class Thread implements Runnable {
  public final static int MIN_PRIORITY;
  public final static int NORM_PRIORITY;
  public final static int MAX_PRIORITY;
  public int getPriority();
  public void setPriority(int p);
}
```

Expliquemos con un poco más de detalle los elementos de la clase `Thread` anteriormente mencionados.

- `public final static int MIN_PRIORITY`: constante de la clase `Thread` que indica la mínima prioridad que puede tener un hilo. Su valor es 1.
- `public final static int NORM_PRIORITY`: constante de la clase `Thread` que indica la prioridad media que puede tener un hilo. Su valor es 5.
- `public final static int MAX_PRIORITY`: constante de la clase `Thread` que indica la máxima prioridad que puede tener un hilo. Su valor es 10.
- `public int getPriority()`: método observador que devuelve la prioridad del hilo que lo invoca.
- `public void setPriority(int p)`: método modificador que configura la prioridad del hilo con el valor `p` que se le pasa.

La prioridad que establecemos en los objetos de la clase `Thread` tiene efecto en la JVM, que indica al planificador del sistema operativo qué hilos van primero, pero no se trata de un contrato absoluto porque depende de:

- de la implementación de la JVM,
- del S.O. subyacente,
- del mapeo de prioridad entre la jvm y la prioridad del S.O.

El siguiente ejemplo utiliza la clase `Task` de los ejemplos anteriores e ilustra cómo cambiar las prioridades de los hilos de Java:

```
/**
 * Programa de ejemplo que modifica la prioridad de varios hilos heredados de la
 * clase Thread.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.io.*;

public class Prioridad {

  public static void main(String[] args) throws InterruptedException {
    Task t1 = new Task(1);
    Task t2 = new Task(2);
    Task t3 = new Task(3);

    System.out.println("Observemos la prioridad de los hilos:");
    System.out.println("El hilo t1 tiene prioridad " + t1.getPriority());
    System.out.println("El hilo t2 tiene prioridad " + t2.getPriority());
    System.out.println("El hilo t3 tiene prioridad " + t3.getPriority());

    //Modificamos la prioridad de los hilos
    t1.setPriority(Thread.MIN_PRIORITY);
    t2.setPriority(Thread.NORM_PRIORITY);
```

```
    t3.setPriority(Thread.MAX_PRIORITY);

    //Comprobamos la prioridad de los hilos
    System.out.println("El hilo t1 tiene ahora prioridad " + t1.getPriority());
    System.out.println("El hilo t2 tiene ahora prioridad " + t2.getPriority());
    System.out.println("El hilo t3 tiene ahora prioridad " + t3.getPriority());

    //Lanzamos t1, t2 y t3
    System.out.println("Lanzamos t1.");
    t1.start();
    System.out.println("Lanzamos t2.");
    t2.start();
    System.out.println("Lanzamos t3.");
    t3.start();

   //Esperamos a que todos los hilos terminen su ejecución
    t1.join();
    t2.join();
    t3.join();
  }
}
```

2.2. Mapeo de hilos entre la JVM y el sistema operativo

A continuación explicaremos las características del mapeo de hilos de la JVM a los sistemas Windows y Linux para entender mejor el comportamiento no esperado que puedan tener nuestros programas.

2.2.1. Mapeo entre los hilos de la JVM y los hilos nativos de Win32

En el mapeo entre hilos de la JVM y los hilos nativos de los sistemas basados en Win32, el sistema operativo conoce el número de hilos que usa la JVM. El mapeo se realiza uno a uno, por lo que el secuenciamiento de hilos en Java está sujeto al secuenciamiento de los hilos en el sistema operativo.

Por otra parte, en la JVM se aplican 10 prioridades, mientras que los sistemas basados en Win32 se aplican 7 más otras 5 prioridades de secuenciamiento. La siguiente tabla hace corresponder las prioridades en los hilos en Java con las prioridades de los sistemas operativos basados en Win32.

Prioridad Java	**Prioridad Win32**
0	THREAD.PRIORITY_IDLE
1 (Thread.MIN_PRIORITY)	THREAD.PRIORITY_LOWEST
2	THREAD.PRIORITY_LOWEST
3	THREAD.PRIORITY_BELOW_NORMAL
4	THREAD.PRIORITY_BELOW_NORMAL
5 (Thread.NORM_PRIORITY)	THREAD.PRIORITY_NORMAL
6	THREAD.PRIORITY_ABOVE_NORMAL
7	THREAD.PRIORITY_ABOVE_NORMAL
8	THREAD.PRIORITY_HIGHEST
9	THREAD.PRIORITY_HIGHEST
10 (Thread.MAX_PRIORITY)	THREAD.PRIORITY_TIME_CRITICAL

2.2.2. Mapeo entre los hilos de la JVM y los hilos nativos de Linux

Las últimas versiones del núcleo de Linux implementan *Native Posix Thread Library*. En este mapeo cada hilo de la JVM se aplica a un hilo del núcleo bajo el modelo de Solaris. La prioridad establecida en la clase `Thread` es un factor muy pequeño en el cálculo global del secuenciamiento, ya que el orden que importa es el que tienen los hilos del NPTL. Al equiparar los hilos de la JVM con los de la NPTL, la equivalencia de prioridades cambia:

Prioridad Java	**Prioridad NPTL**
0	19
1 (Thread.MIN_PRIORITY)	4
2	3
3	2
4	1
5 (Thread.NORM_PRIORITY)	0
6	-1
7	-2
8	-3
9	-4
10 (Thread.MAX_PRIORITY)	-5

Ejercicio 3 Observe el ejemplo del apartado 2.1. Compílelo y ejecútelo en su ordenador. Si fuera posible, ejecútelo en un sistema basado en Win32 y en otro basado en Linux. Observe las diferencias. Modifique el programa del ejemplo para que, supuestamente, termine primero el hilo `t2`, luego el hilo `t1` y por último el hilo `t3`. Comprueba su código con el que aparece en el apartado 7.2.

---

3. Clases `Timer` y `TimerTask`

Otro método de gestión es la planificación de tareas basadas en el tiempo. Ahora no decidiremos sobre la ejecución de los hilos sino sobre la realización de tareas. Podemos decidir si estás tareas se realizarán en un tiempo absoluto o relativo, e incluso si se repiten periódicamente. Para esto necesitamos las clases `Timer` y `TimerTask` que están relacionadas.

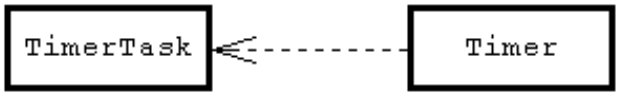

Figura 3: Diagrama que ilustra la relación entre las clases `Timer` y `TimerTask`.

3.1. Representación de tareas con `TimerTask`

La clase `Timer` utiliza a la clase `TimerTask` para la representación de tareas. Las tareas que quieran ser ejecutadas por la clase `Timer` deben pertenecer a otra clase que herede de la clase `TimerTask`. La clase `TimerTask` se encuentra en el paquete `java.util` desde la versión JDK 1.3.

```
public abstract class TimerTask implements Runnable {
  protected TimerTask();
  public abstract void run();
  public boolean cancel();
  public long scheduledExecutionTime();
}
```

Se trata de una clase sencilla con pocos métodos. Al heredar de la clase `TimerTask` es imprescindible implementar el método `run()` con las instrucciones que queremos que tenga nuestra tarea. Los otros métodos no tienen por qué ser sobrescritos cuando se herede de esta clase.

El método `cancel()` se usa para parar la ejecución de la clase. Si una tarea está siendo ejecutada cuando se produzca la llamada a este método, no resulta afectada. Sin embargo, si la tarea aún no ha sido ejecutada, no llega a ejecutarse. O si es periódica, no vuelve a ejecutarse. Este método devuelve `true` si cancela la ejecución de la tarea, ya sea porque aún no se había ejecutado por primera y única vez o en el caso de ejecuciones periódicas que evita la siguiente ejecución. En otro caso, por ejemplo, la tarea ya se estaba ejecutando, devuelve `false`.

El método `scheduledExecutionTime()` se usa para saber cuándo empezó la última invocación de la tarea. Si una tarea está siendo ejecutada, devuelve la hora en la que comenzó a ejecutarse, si no, devuelve la hora de la última invocación que tuvo lugar.

Veamos un ejemplo de cómo implementar una clase que herede de `TimerTask`:

```
/**
 * Código de ejemplo para ilustrar el uso de TimerTask.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class MiTimerTask extends TimerTask {
  private int num;

  MiTimerTask() {
    num = 0;
  }

  MiTimerTask(int n) {
    num = n;
  }

  public void run() {
    for(int i = 0; i < 10; ++i) {
      System.out.println(num);
      ++num;
    }
  }

}
```

Hay que tener en cuenta que la clase que representa la tarea debe heredar de `TimerTask`, por lo que no puede heredar de otras clases.

3.2. Planificación de tareas con `Timer`

Con la clase `Timer` podemos planificar las tareas que hemos creado heredando de `TimerTask`. La clase `Timer` también pertenece al paquete `java.util` desde el JDK 1.3. Con esta clase podemos planificar tareas para que se cumplan dentro de un determinado tiempo, a una hora concreta, o incluso que se repitan periódicamente. Las tareas son colocadas en una lista ordenada y son ejecutadas secuencialmente por un único hilo.

```
public class Timer {
  public Timer();
  public Timer(boolean isDaemon);
  public Timer(String name);
  public Timer(String name, boolean isDaemon);

  public void schedule(TimerTask task, long delay);
  public void schedule(TimerTask task, Date time);
  public void schedule(timerTask task, long delay, long period);
  public void schedule(TimerTask task, Date firstTime, long period);

  public void scheduleAtFixedRate(TimerTask task, long delay, long period);
  public void scheduleAtFixedRate(TimerTask task, Date firstTime, long period);

  public void cancel();
  public int purge();
}
```

Lo primero que observamos en la interfaz es que la clase `Timer` dispone de 4 constructores. Estos constructores nos permiten darle un nombre concreto al hilo, lo cuál puede es muy útil si es necesario monitorizar los hilos existentes durante la corrección. El parámetro booleano nos permite definir si el hilo creado es un hilo demonio (*daemon*). Si el hilo es un *daemon* el programa puede salir cuando todos los hilos de usuario terminen.

Los siguientes métodos de la interfaz son los métodos `schedule()`, que son usados para planificar tareas. Es posible planificar las tareas para que sean ejecutadas después de un tiempo (`delay`), a una hora concreta (`time`), e incluso que se repitan periódicamente cada cierto tiempo (`period`).

Si necesitamos programar una tarea crítica en el tiempo puede que no nos sirva el método `schedule()` para programarla. Anteriormente mencionamos que las tareas se ejecutan todas en un mismo hilo. Por lo que es recomendable que las tareas tengan una ejecución corta. Pero no hay nada que nos lo garantice. Se puede dar el caso de que las tareas se ejecuten más tarde de lo previsto. Además, en tareas que se repiten la siguiente iteración está basada en cuando fue la anterior.

Para solucionar este problema disponemos de dos mecanismos. El primero de ellos es usar los métodos `scheduleAtFixedRate()`, que planifican la siguiente tarea según cuándo se suponía que se tendría que haber ejecutado la iteración anterior. Este método siempre coloca la tarea en el planificador. El otro mecanismo es usar el método `scheduledExecutionTime()` de la clase `TimerTask` y comparar cuándo fue la última iteración con la hora actual. Podemos usar este método desde la propia tarea (en el `run()`), para hacer que se "salte" la ejecución de la iteración si nos conviene. Este método se suele usar junto con el primero para un mejor resultado.

Veamos un ejemplo del uso de la clase `Timer`:

```
/**
 * Código de ejemplo para ilustrar el funcionamiento de la clase Timer.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class MiTimer {

  public static void main(String args[]) {
    Timer miTimer = new Timer();
    MiTimerTask mt1 = new MiTimerTask();
    MiTimerTask mt2 = new MiTimerTask(52);
    MiTimerTask mt3 = new MiTimerTask(77);
    MiTimerTask mt4 = new MiTimerTask(103);

    System.out.println("Pulse Ctrl + Z para detener el programa.");
    miTimer.schedule(mt1, 200);
    miTimer.schedule(mt2, 800);
    miTimer.schedule(mt3, 1500, 3000);
    miTimer.scheduleAtFixedRate(mt4, 2000, 4500);

  }
}
```

Ejercicio 4 Cree un programa que utilice la clase `Timer` para simular el lanzamiento de cohetes. Implemente una clase que simbolice la cuenta atrás. En este ejercicio es importante que la cuenta atrás sea constante. Compruebe su solución con la que se encuentra en el apartado 7.3.

Ejercicio 5 Cree un programa que utilice la clase `Timer` para imprimir por pantalla la hora a cada segundo. Implemente una clase que guarde el valor de la hora y permita imprimirla cuando los milisegundos valgan 0. En este ejercicio es importante que la llamada al método para imprimir la hora se realice cuando se supone que ha pasado un segundo. Compruebe su solución con la que se encuentra en el apartado 7.4.

---

4. Planificación de tareas con *thread pools*

Hay dos maneras de crear *thread pools* que admiten planificación. Pero dependen de algunas de las clases que aparecen en el apartado correspondiente a *thread pools* del guión anterior y a otras que no han sido explicadas. Así que antes de ver los dos mecanismos daremos un repaso a las clases que intervienen. El siguiente diagrama es un resumen de las clases e interfaces que intervienen en los *thread pools*.

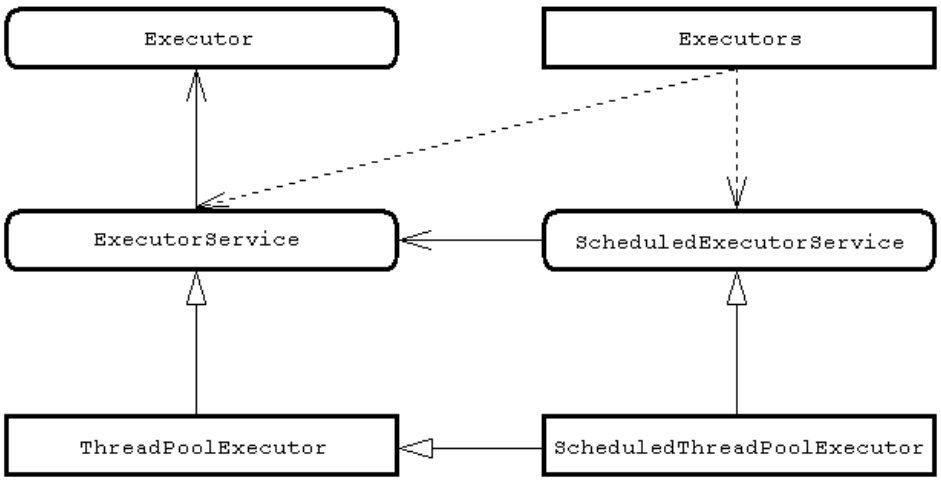

Figura 4: Diagrama de las clases que intervienen en los *thread pools*.

4.1. Interfaz `ScheduledExecutorService`

Un objeto que implemente esta interfaz es como un objeto que implemente la interfaz `ExecutorService` y que además puede planificar tareas para que se ejecuten después de un determinado tiempo de retardo o para que se ejecuten periódicamente.

Esta interfaz hereda de la interfaz `ExecutorService` que se explicó en el guión anterior. Además de heredar todos los métodos de dicha interfaz, implementa algunos nuevos:

```
public interface ScheduledExecutorService extends ExecutorService {
  ...
  ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
  ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long
    period, TimeUnit unit);
  ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay,
    long delay, TimeUnit unit);
}
```

Veamos brevemente qué hacen los métodos anteriores.

- `public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)`: crea una acción y la ejecuta tras el tiempo de retardo indicado.
- `public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,verb long period, TimeUnit unit)`: crea una acción que se ejecuta periódicamente. Su primera ejecución es tras el tiempo inicial de retardo indicado y luego se ejecuta una vez pasado el periodo indicado. Si alguna ejecución de la tarea provoca una excepción, se suprimen las siguientes iteraciones. Si una ejecución tarda más tiempo que el indicado en el periodo, la siguiente ejecución empezará más tarde de lo previsto, pero nunca se ejecutarán concurrentemente.
- `public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`: crea una acción que se ejecuta periódicamente. Su primera ejecución es tras el tiempo inicial de retardo indicado y luego se ejecuta con un retardo

entre el final de una iteración y el principio de la siguiente indicado por el otro retardo. Si alguna ejecución de la tarea provoca una excepción, se suprimen las siguientes iteraciones.

4.2. Clase `ScheduledThreadPoolExecutor`

La clase `ScheduledThreadPoolExecutor` tiene un funcionamiento parecido al de la clase `ThreadPoolExecutor`, solo que permite la planificación de tareas a partir de un cierto momento e incluso su repetición. Realmente esta clase utiliza un *thread pool*, ya que hereda de `ThreadPoolExecutor`, pero en el que el número de hilos es fijo y el tamaño de la cola es infinito. Veamos parte de la interfaz de la clase:

```
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor
implements ScheduledExecutorService {
  public ScheduledThreadPoolExecutor(int corePoolSize);
  public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory);
  public ScheduledThreadPoolExecutor(int corePoolSize,
    RejectedExecutionHandler handler);
  public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory,
    RejectedExecutionHandler handler);

  ...

  //Funciones heredadas de ScheduledExecutorService
  public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
  public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
    long period, TimeUnit unit);
  public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long
    initialDelay, long delay, TimeUnit unit);
  ...

  //Funciones heredadas de ThreadPoolExecutor
  public void execute(Runnable command);
  public void shutdown();
  public List shutdownNow();
  ...

  //Funciones propias
  public boolean getContinueExistingPeriodicTasksAfterShutdownPolicy();
  public boolean getExecuteExistingDelayedTasksAfterShutdownPolicy();
  public void setContinueExistingPeriodicTasksAfterShutdownPolicy(boolean value);
  public void setExecuteExistingDelayedTasksAfterShutdownPolicy(boolean value);

  ...
}
```

Muchas de las funciones arriba indicadas funcionan igual o casi igual que lo hacen en las clases o interfaces de las que proceden. Por eso sólo nos detendremos en las que tienen un comportamiento distinto y significativo o son nuevas. Para completar sus conocimientos, consulte la documentación oficial.

- **Constructores**: la principal diferencia entre los constructores de esta clase y los de la clase `ThreadPoolExecutor` es que siempre tiene un número fijo de hilos, que es el indicado en el constructor. Si se desea este número de hilos se puede cambiar usando las funciones que la clase

`ThreadPoolExecutor` proporcionaba para ello. Otra característica es que la lista de tareas es infinita, por lo que todas las tareas se añaden a la lista para que sean ejecutadas.

- Las funciones heredadas de la interfaz `ScheduledExecutorService` funcionan de la misma manera que en la interfaz antes explicada.
- `public void execute(Runnable command)`: esta función sufre un pequeño cambio importante con respecto a la versión de la clase `ThreadPoolExecutor`. En `ThreadPoolExecutor` esta función garantizaba que la tarea se ejecutaría en algún momento futuro. Sin embargo, en `ScheduledThreadPoolExecutor` la tarea es mandada a ejecutar con retraso 0.
- Las funciones `shutdown()` y `shutdownNow()` funcionan igual que en `ThreadPoolExecutor`, sin embargo, el efecto que tiene en `ScheduledThreadPoolExecutor` es distinto debido a la periocidad de las tareas. Las tareas periódicas seguirían ejecutándose a pesar de usar estas funciones. Para conseguir que `shutdown()` y `shutdownNow()` tengan el mismo efecto que en la clase `Timer`, la clase `ScheduledThreadPoolExecutor` introduce dos poíticas nuevas. Hablaremos sobre ellas en la descripción de los métodos que las modifican.
- Los métodos `boolean getContinueExistingPeriodicTasksAfterShutdownPolicy()` y `setContinueExistingPeriodicTasksAfterShutdownPolicy(boolean value)` modifican la política llamada `ContinueExistingPeriodicTasksAfterShutdownPolicy` que determina si las tareas periódicas se siguen ejecutando tras llamar a los métodos `shutdown()` y `shutdownNow()`.
- Los métodos `boolean getExecuteExistingDelayedTasksAfterShutdownPolicy()` y `void setExecuteExistingDelayedTasksAfterShutdownPolicy(boolean value)` modifican la política llamada `ExecuteExistingDelayedTasksAfterShutdownPolicy` que determina si las tareas que tienen asignado tiempo de retardo se siguen ejecutando tras llamar a los métodos `shutdown()` y `shutdownNow()`.

Cuando las dos políticas propias de la clase`ScheduledThreadPoolExecutor` valen falso, el objeto de la clase se cerrará por completo tras las llamadas a `shutdown()` o `shutdownNow()`.

A continuación podemos ver un ejemplo de uso de la clase `ScheduledThreadPoolExecutor`. Para el ejemplo, usamos la clase `Task` usada en los ejemplos del guión anterior:

```
/**
 * Clase Task que representa una tarea que puede ser ejecutada concurrentemente.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Task implements Runnable {
  private int id;

  public Task() {}
  public Task(int id) {
    this.id = id;
  }

  public void run() {
    System.out.println("Ejecutando tarea " + id);
    for (int i = 0; i < 100; ++i) {}
    System.out.println("Fin de la tarea " + id);
```

```
  }
}
```

Y el uso de la clase `ScheduledThreadPoolExecutor`:

```
/**
 * Programa de prueba para ilustrar el uso de la clase ScheduledThreadPoolExecutor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.*;

public class UsaScheduledThreadPoolExecutor {
  public static void main(String[] args)
  {
    Scanner teclado = new Scanner(System.in);

    System.out.print("Introduzca el número de hilos que tendrá el thread pool: ");
    int numHilos = teclado.nextInt();

    ScheduledThreadPoolExecutor stpe = new ScheduledThreadPoolExecutor(numHilos);

    Task t1 = new Task(1);
    Task t2 = new Task(2);
    Task t3 = new Task(3);
    Task t4 = new Task(4);

    //La tarea t1 se ejecutará inmediatamente.
    stpe.execute(t1);
    //La tarea t2 comenzará dentro de 3 segundos.
    stpe.schedule(t2, 3, TimeUnit.SECONDS);
    //La tarea t3 comenzará dentro de 5 segundos y se repetirá cada 8 segundos.
    stpe.scheduleAtFixedRate(t3, 5, 8, TimeUnit.SECONDS);
    //La tarea t4 comenzará dentro de 15 segundos y se repetirá a un intervalo
    //constante de 3 segundos.
    stpe.scheduleWithFixedDelay(t4, 15, 3, TimeUnit.SECONDS);
  }
}
```

Ejercicio 6 Rehaga el ejercicio 4 sobre el lanzamiento de cohetes usando la clase `ScheduledThreadPoolExecutor`. Implemente el programa de tal forma que no se solapen los lanzamientos. Cuando lo termine, compruebe su solución con la que aparece en el apartado 7.5.

Ejercicio 7 Cree un programa que utilice la clase `ScheduledThreadPoolExecutor` para imprimir por pantalla simultáneamente la hora de 5 ciudades. Implemente una clase que permita imprimir la hora cada

segundo cuando los milisegundos valgan 0. Compruebe su solución con la que se encuentra en el apartado 7.6.

4.3. Clase `Executors`

Como vimos en el guión anterior, la clase `Executors` nos permite crear *thread pools* de una manera más sencilla y genérica. Desde ella, también podemos crear *thread pools* que permiten la ejecución de tareas programadas y periódicas. En este caso, el *thread pool* que devuelve es un objeto de la clase `ScheduledThreadPoolExecutor` o un objeto de otra clase que implemente la interfaz `ScheduledExecutorService`:

```
public class Executors {
  ...
  static ScheduledExecutorService newScheduledThreadPool(int corePoolSize);
  ...
}
```

Este método recibe el tamaño que desea que tenga el *thread pool* y crea un *thread pool* de tamaño fijo. Usar este método o usar el constructor de la clase `ScheduledThreadPoolExecutor` que sólo recibe un parámetro es equivalente.

Veamos cómo crearíamos un `ScheduledThreadPool` usando la clase `Executors`. En el ejemplo utilizaremos la clase `Task` del ejemplo anterior:

```
/**
 * Programa de prueba para ilustrar el uso de la clase Executors.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.*;

public class UsaExecutors {
  public static void main(String[] args)
  {
    Scanner teclado = new Scanner(System.in);

    System.out.print("Introduzca el número de hilos que tendrá el thread pool: ");
    int numHilos = teclado.nextInt();

    ScheduledExecutorService exec = Executors.newScheduledThreadPool(numHilos);

    Task t1 = new Task(1);
    Task t2 = new Task(2);
    Task t3 = new Task(3);
    Task t4 = new Task(4);

    //La tarea t1 se ejecutará inmediatamente.
    exec.execute(t1);
```

```
    //La tarea t2 comenzará dentro de 3 segundos.
    exec.schedule(t2, 3, TimeUnit.SECONDS);
    //La tarea t3 comenzará dentro de 5 segundos y se repetirá cada 8 segundos.
    exec.scheduleAtFixedRate(t3, 5, 8, TimeUnit.SECONDS);
    //La tarea t4 comenzará dentro de 15 segundos y se repetirá a un intervalo
    //constante de 3 segundos.
    exec.scheduleWithFixedDelay(t4, 15, 3, TimeUnit.SECONDS);
  }
}
```

Ejercicio 8 Modifique el ejercicio anterior sobre la hora de 5 ciudades para crear el `ScheduledThreadPool` desde la clase `Executors`. Compruebe su programa principal con el código que encontará en el apartado 7.7.

---

5. Ventajas y desventajas: conclusiones sobre los métodos analizados

En este guión hemos visto dos maneras de planificación en la concurrencia: mediante el control de la prioridad de los hilos que las ejecutan y mediante la planificación de tareas.

La actual especificación de la JVM no establece un modelo de planificación por prioridades. Podemos modificar la prioridad de los hilos en la JVM, pero el comportamiento del mismo programa puede y debe variar al ser ejecutado en diferentes máquinas. En el caso de necesitar que un grupo de tareas sean ejecutadas en un orden estricto, el control de la prioridad de sus hilos no es el método adecuado. Aunque si cada una de las priodidades que establecemos en Java corresponden a un valor distinto de prioridad en el S.O. sí es posible controlar el orden de ejecución de los hilos del S.O. que corresponden a cada uno de los hilos de la JVM.

En conclusión, el control de la ejecución de tareas basado en el control de la prioridad de los hilos en la JVM no es el método aconsejado. Es difícil de controlar y programar, ya que hay que trabajar con prioridades a bajo nivel, y las instrucciones que pueden servir para que los hilos se ejecuten como queremos en un S.O. puede no servir en otro, perdiendo así la propiedad de independencia de la plataforma que ofrece la JVM.

En cuanto a la planificación de tareas, Java proporciona distintos métodos. Podemos usar las clases `Timer` y `TimerTask` o usando un objeto de la clase `ScheduledThreadPoolExecutor` que es *thread pool* que admite planificación de tareas. Construir objetos de esta última clase también se puede hacer de dos formas distintas.

Usar la clase `Timer` tiene como ventajas que podemos planificar una tarea para que sea ejecutada a una hora específica. También es más simple de usar que la clase `ScheduledThreadPoolExecutor`, por lo que es preferible si sólo hay que ejecutar un pequeño número de tareas.

Sin embargo, al usar `Timer` la tarea que queremos que se ejecute debe heredar de la clase `TimerTask`, no pudiendo así heredar de otras clases. Otra desventaja es que con `Timer` todas las tareas son ejecutadas en el mismo hilo, por lo que puede sobrecargarse de trabajo y es posible que la ejecución se atrase. Como consecuencia, no es un método útil para mantener un reloj u otra tarea crítica en el tiempo.

Para evitar los inconvenientes de `Timer` podemos utilizar un *thread pool* de la clase `ScheduledThreadPoolExecutor`. Los objetos de esta clase pueden utilizar más hilos que `Timer`, tantos como se le indique a su constructor. Además, las tareas que se ejecutan en este *thread pool* no tienen que heredar de otras clases, sino que basta con que implementen a la interfaz `Runnable`, por lo que pueden heredar de otras clases si el programador lo precisa.

La desventaja de este sistema respecto a `Timer` es que siempre hay que indicar cuándo se desea que se realicen las tareas indicando un tiempo relativo, no se puede indicar que se ejecute a una hora concreta. Por lo general, no supone una gran desventaja, ya que es fácil de calcular el retraso inicial antes de invocar al constructor.

En cuanto a las formas de crear objetos de esta clase, usar el método de la clase `Executors` puede resultar más sencillo, pero desde los propios constructores de la clase se puede configurar con más precisión su funcionamiento.

6. Ejercicios

En esta sección podrá encontrar algunos ejercicios más para afianzar sus conocimientos.

Ejercicio 9 Realice un programa que simule un teléfono y reprocuzca (escrito por pantalla) el tono de llamada. Haga que el sonido se repita varias veces hasta que lo pare el usuario. Implemente varias llamadas. Compruebe su programa con el que puede encontrar en el apartado 7.8.

---

7. Soluciones de los ejercicios

En esta sección encontrará las soluciones a los ejercicios propuestos a lo largo del guión.

7.1. Ejercicio 2

Para la realización de este ejercicio hemos reutilizado la clase `Task` del ejemplo del apartado 2. A continuación podemos ver la clase:

```
/**
 * Clase Task que representa una tarea que puede ser ejecutada concurrentemente.
 * [Ejemplo]
 * @author Natalia Partera
 * @version 1.0
 */

import java.lang.Thread.*;

public class Task extends Thread {
  private int id;

  public Task() {}
  public Task(int id) {
    this.id = id;
  }

  public void run() {
    System.out.println("Ejecutando tarea " + id);
    for(int i = 0; i < 100; ++i) {
      System.out.println("Tarea " + id + ": vuelta" + i);
    }
    System.out.println("Fin de la tarea " + id);
  }
}
```

El programa que nos pedía el ejercicio viene implementado por la siguiente clase `UsaThreadActuales`, modificación de la misma clase del apartado 2.

```
/**
 * Programa que utiliza métodos válidos de la clase Thread.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.io.*;

public class UsaThreadActuales {

  public static void main(String[] args) throws InterruptedException {
    Task t1 = new Task(1);
    Task t2 = new Task(2);
```

```
    Task t3 = new Task(3);

    //Lanzamos t1, t2 y t3
    System.out.println("Lanzamos t1.");
    t1.start();
    System.out.println("Lanzamos t2.");
    t2.start();
    System.out.println("Lanzamos t3.");
    t3.start();

    //Si t2 o t3 están vivos, t1 cede su uso del procesador
    while(t2.isAlive() || t3.isAlive()) {
      t1.yield();
    }

    //Si t3 está vivo, t2 cede su uso del procesador
    while(t3.isAlive()) {
      t2.yield();
    }
   //Esperamos a que todos los hilos terminen su ejecución
    t1.join();
    t2.join();
    t3.join();
  }
}
```

7.2. Ejercicio 3

Para cambiar la prioridad de los hilos y que terminen, supuestamente, en el orden `t2 - t1 - t3`, el programa `Prioridad` debería sufrir las siguientes modificaciones:

```
/**
 * Programa de ejemplo que modifica la prioridad de varios hilos heredados de la
 * clase Thread.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.io.*;

public class Prioridad {

  public static void main(String[] args) throws InterruptedException {
    Task t1 = new Task(1);
    Task t2 = new Task(2);
    Task t3 = new Task(3);

    System.out.println("Observemos la prioridad de los hilos:");
    System.out.println("El hilo t1 tiene prioridad " + t1.getPriority());
    System.out.println("El hilo t2 tiene prioridad " + t2.getPriority());
```

```
    System.out.println("El hilo t3 tiene prioridad " + t3.getPriority());

    //Modificamos la prioridad de los hilos
    t1.setPriority(Thread.NORM_PRIORITY);
    t2.setPriority(Thread.MAX_PRIORITY);
    t3.setPriority(Thread.MIN_PRIORITY);

    //Comprobamos la prioridad de los hilos
    System.out.println("El hilo t1 tiene ahora prioridad " + t1.getPriority());
    System.out.println("El hilo t2 tiene ahora prioridad " + t2.getPriority());
    System.out.println("El hilo t3 tiene ahora prioridad " + t3.getPriority());

    //Lanzamos t1, t2 y t3
    System.out.println("Lanzamos t1.");
    t1.start();
    System.out.println("Lanzamos t2.");
    t2.start();
    System.out.println("Lanzamos t3.");
    t3.start();

   //Esperamos a que todos los hilos terminen su ejecución
    t1.join();
    t2.join();
    t3.join();
  }
}
```

7.3. Ejercicio 4

A continuación puede observar la clase que simula la cuenta atrás del lanzamiento:

```
/**
 * Clase CuentaAtras que representa la cuenta atrás para el lanzamiento de un
 * cohete.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class CuentaAtras extends TimerTask {
  private int segundos;
  private String cohete;

  CuentaAtras(String nave) {
    segundos = 30;
    cohete = nave;
  }

  CuentaAtras(int seg, String nave) {
```

```
    segundos = seg;
    cohete = nave;
  }

  public void run() {
    if(segundos > 0) {
      System.out.println(cohete + ": " + segundos);
      --segundos;
    }
    else if (segundos == 0) {
      System.out.println("Lanzamiento de la nave " + cohete);
      cancel();
    }
  }

}
```

Y este es el programa que simula el lanzamiento:

```
/**
 * Clase Lanzamiento que simula la cuenta atrás del lanzamiento de un cohete.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class Lanzamiento {

  public static void main(String args[]) {
    Timer lanzamiento = new Timer();
    CuentaAtras apolo = new CuentaAtras(10, "Apolo");
    CuentaAtras soyuz = new CuentaAtras("Soyuz");
    long hora;
    boolean fin = false;

    hora = System.currentTimeMillis();
    lanzamiento.schedule(apolo, 2000, 1000);
    lanzamiento.schedule(soyuz, 15000, 1000);

    while(!fin) {
      if(System.currentTimeMillis() > hora + 60000) {
        System.out.println("Lanzamientos finalizados.");
        lanzamiento.cancel();
        fin = true;
      }
    }
  }
}
```

7.4. Ejercicio 5

Este es la clase que muestra la hora cuando los milisegundos valen 0:

```
/**
 * Clase Minuto que muestra la hora cuando los milisegundos valen 0.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class Minuto extends TimerTask {
  private Calendar hora;

  Minuto() {}

  public void run() {
    hora = Calendar.getInstance();
    while (System.currentTimeMillis() % 1000 != 0) {}
    hora = Calendar.getInstance();
    System.out.println(String.format("%1$tH:%1$tM:%1$tS %1$tL", hora));
  }
}
```

Y este es el programa principal que programa la impresión de la hora a cada segundo:

```
/**
 * Clase reloj que muestra la hora cada segundo.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;
import java.io.*;

public class Reloj {

  public static void main(String args[]) {

    Timer reloj = new Timer();
    Minuto hora = new Minuto();

    reloj.scheduleAtFixedRate(hora, 100, 1000);

  }
}
```

7.5. Ejercicio 6

La clase `CuentaAtras` queda ahora así:

```
/**
 * Clase CuentaAtras que representa la cuenta atrás para el lanzamiento de un
 * cohete.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class CuentaAtras implements Runnable {
  private int segundos;
  private String cohete;

  CuentaAtras(String nave) {
    segundos = 30;
    cohete = nave;
  }

  CuentaAtras(int seg, String nave) {
    segundos = seg;
    cohete = nave;
  }

  public void run() {
    if(segundos > 0) {
      System.out.println(cohete + ": " + segundos);
      --segundos;
    }
    else if (segundos == 0) {
      System.out.println("Lanzamiento de la nave " + cohete);
      --segundos;
    }
  }

}
```

Y el programa principal es ahora el siguiente:

```
/**
 * Clase Lanzamiento que simula la cuenta atrás del lanzamiento de un cohete.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.*;
```

```
public class Lanzamiento {

  public static void main(String args[]) {
    ScheduledThreadPoolExecutor lanzadera = new ScheduledThreadPoolExecutor(2);
    CuentaAtras apolo = new CuentaAtras(10, "Apolo");
    CuentaAtras soyuz = new CuentaAtras("Soyuz");
    long hora;
    boolean fin = false;

    hora = System.currentTimeMillis();
    lanzadera.scheduleWithFixedDelay(apolo, 1, 1, TimeUnit.SECONDS);
    lanzadera.scheduleWithFixedDelay(soyuz, 15, 1, TimeUnit.SECONDS);

    while(!fin) {
      if(System.currentTimeMillis() > hora + 60000) {
        System.out.println("Lanzamientos finalizados.");
        lanzadera.shutdown();
        fin = true;
      }
    }
  }
}
```

7.6. Ejercicio 7

Esta es la nueva versión de la clase `Hora`:

```
/**
 * Clase Hora que muestra la hora cuando los segundos valen 0.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class Hora implements Runnable {
  private Calendar hora;
  private String ciudad;
  private int utc;

  Hora(String ciud, int utc) {
    this.ciudad = ciud;
    this.utc = utc;
  }

  public void run() {
    hora = Calendar.getInstance();
    while (System.currentTimeMillis() % 1000 != 0) {}
    hora = Calendar.getInstance();
```

```
    int offset = hora.get(Calendar.ZONE_OFFSET) + hora.get(Calendar.DST_OFFSET);
    offset = offset / 1000;
    offset = offset / 3600;
    hora.add(Calendar.HOUR_OF_DAY, -offset + utc);
    System.out.println(String.format(ciudad + ": %1$tH:%1$tM:%1$tS", hora));
  }
}
```

Y así es como queda el programa principal:

```
/**
 * Clase relojes que muestra la hora en diferentes ciudades cada minuto.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.concurrent.*;
import java.util.*;

public class Relojes {

  public static void main(String args[]) {

    ScheduledThreadPoolExecutor relojes = new ScheduledThreadPoolExecutor(5);
    Hora madrid = new Hora("Madrid", 2);
    Hora londres = new Hora("Londres", 1);
    Hora paris = new Hora("París", 2);
    Hora berlin = new Hora("Berlín", 2);
    Hora tokio = new Hora("Tokio", 9);

    relojes.scheduleAtFixedRate(madrid, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(londres, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(paris, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(berlin, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(tokio, 0, 1, TimeUnit.SECONDS);

  }
}
```

7.7. Ejercicio 8

Así queda el programa principal al usar la clase `Executors`:

```
/**
 * Clase relojes que muestra la hora en diferentes ciudades cada minuto.
 *
 * @author Natalia Partera
 * @version 2.0
 */
```

```
import java.util.concurrent.*;
import java.util.*;

public class Relojes {

  public static void main(String args[]) {

    ScheduledExecutorService relojes = Executors.newScheduledThreadPool(5);
    Hora madrid = new Hora("Madrid", 2);
    Hora londres = new Hora("Londres", 1);
    Hora paris = new Hora("París", 2);
    Hora berlin = new Hora("Berlín", 2);
    Hora tokio = new Hora("Tokio", 9);

    relojes.scheduleAtFixedRate(madrid, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(londres, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(paris, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(berlin, 0, 1, TimeUnit.SECONDS);
    relojes.scheduleAtFixedRate(tokio, 0, 1, TimeUnit.SECONDS);

  }
}
```

7.8. Ejercicio 9

Para este programa hemos utilizado la clase `Timer`, por lo que las llamadas no podrán pisarse. A continuación puede ver el código de la clase que hereda a `TimerTask`:

```
/**
 * Clase Tono que representa una tono de llamada.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;

public class Tono extends TimerTask {
  private String sonido;

  Tono() {
    sonido = new String("riiing");
  }

  Tono(String son) {
    sonido = new String(son);
  }

  public void run() {
    System.out.println(sonido);
  }
}
```

Y este es el programa principal que hace uso de `Timer`:

```
/**
 * Clase Telefono que utiliza objetos de la clase Tono para avisar de una llamada.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.util.*;
import java.io.*;

public class Telefono {

  public static void main(String args[]) throws IOException {
    Timer telefono = new Timer();
    Tono ring = new Tono();
    Tono ring2 = new Tono("ring ring");
    Tono grito = new Tono("¡Coge el teléfono!");
    Tono bip = new Tono("bip");
    Tono bip2 = new Tono("biip");
    Tono bip3 = new Tono("biiip");
    int fin = 'n';

    System.out.println("Para parar una llamada, introduzca ''q''.");
    System.out.println();

    System.out.println("Llamada 1:");
    telefono.schedule(ring, 3000, 1000);

    do {
      fin = System.in.read();
      System.out.println();
    } while (fin != 'q');
    ring.cancel();

    System.out.println("Llamada 2:");
    telefono.schedule(ring2, 3000, 1000);
    telefono.schedule(grito, 5500, 2000);

    do {
      fin = System.in.read();
      System.out.println();
    } while (fin != 'q');
    ring2.cancel();
    grito.cancel();

    System.out.println("Llamada 3:");
    telefono.schedule(bip, 3000, 1500);
    telefono.schedule(bip2, 5000, 2000);
    telefono.schedule(bip3, 5500, 2000);
```

```
    do {
      fin = System.in.read();
      System.out.println();
    } while (fin != 'q');
    telefono.cancel();
  }

}
```