Programación Concurrente y de Tiempo Real

Guión de prácticas 4: Creación y control de Hilos en Java

Natalia Partera Jaime
Alumna colaboradora de la asignatura

Índice

1. Hilos y Concurrencia

Cuando ejecutamos un programa, se crea un proceso que es gestionado por el sistema operativo. Este proceso recibe recursos asignados por el sistema operativo, el cual también controla cuándo puede cada proceso hacer uso del procesador. Hasta el momento, los programas que implementábamos constaban de un sólo hilo (*thread*) de ejecución, correspondiente al método `main`. Estos programas comenzaban su ejecución al principio del método `main`, continuaban ejecutando las instrucciones y finalizaban su ejecución cuando llegaban al final de `main`.

De la misma manera en que varios procesos pueden ser ejecutados concurrentemente, es posible fraccionar un programa para que sus partes se ejecuten concurrentemente. Para lograrlo, se asignan hilos a cada tarea del programa que puede ser ejecutada concurrentemente. No todos los lenguajes de programación soportan la concurrencia en sus programas. Java sí que soporta varios hilos de ejecución simultáneos, por eso, los programas de Java pueden crear dentro de sí mismos varias secuencias de ejecución concurrentes.

En contraste con los procesos concurrentes, que son independientes entre sí, los hilos de un mismo proceso comparten recursos como el espacio de direcciones virtuales y los recursos del sistema operativo. De este modo, todos los hilos de un proceso tienen acceso a los datos y procedimientos del proceso. Sin embargo, cada hilo posee su propio contador de programa y pila de llamadas a métodos.

En ocasiones puede que implementemos procesos con hilos concurrentes para optimizar el uso de la CPU, o porque el problema se modele mejor de manera concurrente o porque no admita otra solución razonable (como por ejemplo, un programa en un servidor que tenga que responder a los clientes). Sin embargo, al utilizar la concurrencia multihilo aparecen los problemas habituales de la concurrencia: puede que varios hilos necesiten comunicarse entre ellos porque colaboren para un determinado fin, o puede que estén compitiendo por un recurso común o del sistema. Para asegurar el correcto funcionamiento de los hilos es necesario utilizar mecanismos de **comunicación** y **sincronización** entre hilos, evitando así problemas de exclusión mutua, condición de sincronización, esquema de prioridades o interbloqueos.

En Java existen varias formas para implementar la concurrencia: heredar de la clase `Thread`, implementar la interfaz `Runnable` o utilizar objetos de la clase `ThreadPoolExecutor` con la ayuda de los métodos de la clase `Executors`, entre otros. A continuación veremos detalladamente estas formas.

2. Implementar la Concurrencia en Java

En Java, los hilos se representan mediante la clase `java.lang.Thread`. Así que, para implementar la concurrencia en Java, hay que crear objetos de la clase `Thread` o de una clase que extienda a la clase `Thread`.

2.1. Usando la clase `Thread`

La clase `Thread`, perteneciente al paquete `java.lang`, sirve para representar hilos. Podemos crear hilos fácilmente extendiendo con la clase de la que queremos crear hilos a la clase `Thread`:

```
class MiHilo extends Thread
```

Esta clase posee algunos métodos especiales que nos ayudan a controlar los hilos. A continuación explicamos los más usados:

```
public class Thread extends Object implements Runnable {
  public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target, String name);
  ...
  public long getId();
  public boolean isAlive();
  public void join();
  public void run();
  public void start();
  ...
}
```

- `public Thread()`: constructor, reserva memoria para un nuevo objeto de la clase `Thread`.
- `public Thread(String name)`: constructor, reserva memoria para un nuevo objeto de la clase `Thread`. El hilo se llamará como se indique en la variable `name`.
- `public Thread(Runnable target)`: constructor, reserva memoria para un nuevo objeto de la clase `Thread`. En la llamada al método `run()` se ejecutará dicho método del objeto `target`.
- `public Thread(Runnable target, String name)`: constructor, reserva memoria para un nuevo objeto de la clase `Thread` que se llamará como se indique en `name` y ejecutará el método `run()` del objeto `target` cuando `run()` sea invocado.
- `public long getId()`: método observador que devuelve el identificador del objeto `Thread`
- `public boolean isAlive()`: método observador que comprueba si el hilo sigue vivo.
- `public void join()`: método que impide que el programa principal continue con su ejecución hasta que no termine la ejecución del hilo desde el que se llama. Introduce una condición de espera en el programa principal.
- `public void run()`: método que contiene el código del hilo. En este método se programa el funcionamiento del hilo, lo que queremos que se ejecute en cada hilo.
- `public void start()`: método que causa el inicio de la ejecución del hilo.

Para ilustrar el funcionamiento de un programa donde se lanzan hilos y se introduce condición de espera, utilizando el método `join()`, para continuar con el programa principal, observe el siguiente gráfico:

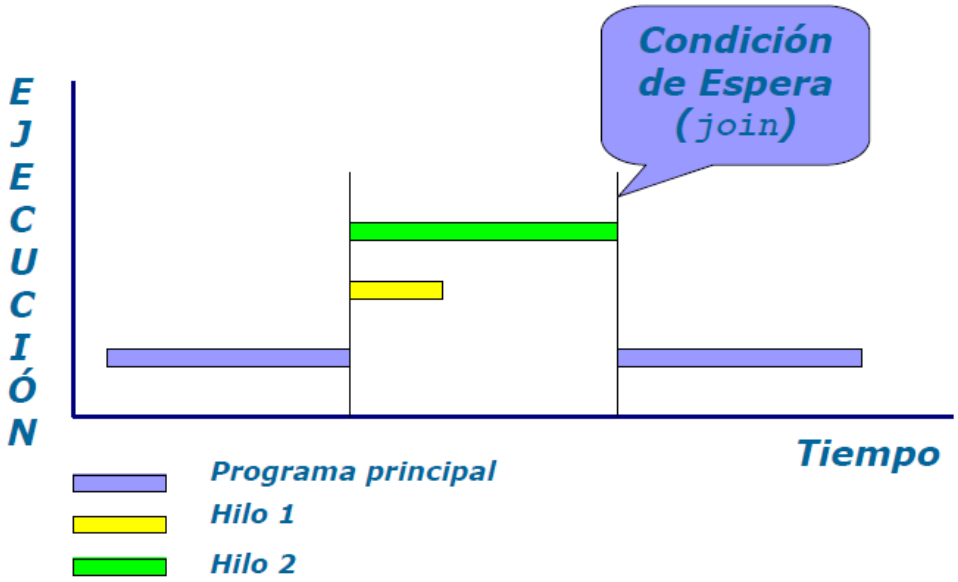

Figura 1: Gráfico de la secuencia temporal de co-rutina con hilos.

Veamos un ejemplo. Lo primero es extender la clase de la que queremos crear hilos. En esta clase implementamos el método `run()` de la clase `Thread`, que contendrá las instrucciones que ejecutará el hilo:

```
/**
 * Clase que representa a una palabra y comprueba si es palíndromo.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.lang.*;

public class Palabra extends Thread {
  String palabra;

  Palabra() {}
  Palabra(String pal) {
    palabra = pal;
  }

  //El método run() comprobará si la palabra es un palíndromo.
  public void run() {
    boolean palindromo = true;
    int i = 0;
    int j = palabra.length() - 1;

    while(palindromo && i < j) {
      if(palabra.charAt(i) == palabra.charAt(j)) {
```

```
        ++i;
        --j;
      }
      else
        palindromo = false;
    }
    if(palindromo)
      System.out.println("La palabra " + palabra + " es un palíndromo.");
    else
      System.out.println("La palabra " + palabra + " no es un palíndromo.");
  }
}
```

A continuación creamos un programa de prueba que cree varios hilos y los ejecute. Para ello usaremos los métodos `start()` y `join()` anteriormente explicados de la clase `Thread`:

```
/**
 * Programa en Java que lanza varios hilos comprobando si varias palabras son
 * palíndromos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaPalabra {
  public static void main (String[] args) throws InterruptedException {
    Palabra pal1 = new Palabra("casa");
    Palabra pal2 = new Palabra("ala");
    Palabra pal3 = new Palabra("Oso");
    //A continuación lanzamos los hilos
    pal1.start();
    pal2.start();
    pal3.start();
    //Esperamos a que terminen los hilos
    pal1.join();
    pal2.join();
    pal3.join();
    System.out.println("Hilos terminados.");
  }
}
```

También es posible crear el hilo y lanzarlo en una sola instrucción. Cuando el hilo termine, acabará el programa de forma natural. Veamos un ejemplo de creación de un nuevo hilo según este método para el ejemplo anterior:

```
new Palabra("mirar").start();
```

Ejercicio 1 Cree una clase que almacene dos números y de la que puedan crearse hilos. En su ejecución, los hilos deberán hacer las cuatro operaciones básicas (suma, resta, multiplicación y división) y mostrar

su resultado por pantalla. Cree un programa de prueba donde se lancen 3 hilos. Compruebe su resultado con el programa del apartado 5.1.

2.2. Usando la interfaz `Runnable`

Si implementamos hilos haciendo que nuestra clase herede de la clase `Thread`, entonces ya no podrá nuestra clase heredar de otra más. Así pues, otra forma de crear hilos es usando la interfaz `Runnable`, también del paquete `java.lang`. Para ello, lo que debemos hacer es implementar la interfaz `Runnable` en la clase que define los hilos, y sobreescribir el método `run()` para definir el comportamiento que queremos que tengan los hilos. Luego, en el programa de prueba, creamos los objetos de la clase anterior. La interfaz `Runnable` expresa que los objetos que la implementen puedan ejecutarse concurrentemente. Pero estos objetos no pueden ejecutarse de forma autónoma, ya que no son hilos. Así que es necesario que estos objetos sean pasados como parámetros a los constructores de la clase `Thread`, uno por objeto.

La interfaz `Runnable` tan sólo tiene el método `run()` que hemos indicado anteriormente. Su signatura es como el de la clase `Thread`, y su funcionamiento también es similar: cuando un objeto que implementa la interfaz `Runnable` es usado para crear un hilo, el método `run()` es invocado al comenzar la ejecución del hilo.

```
public interface Runnable
{
  public void run();
}
```

Veamos un ejemplo. Primero veamos la clase que implementa la interfaz `Runnable`:

```
/**
 * Clase que almacena un número y cuenta 50 a partir de él.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.lang.*;

public class Cuenta50 implements Runnable {
  int numero;

  Cuenta50() {}
  Cuenta50(int num) {
    numero = num;
  }

  public void run() {
    for(int i = 0; i < 50; ++i) {
      System.out.println(numero + " + " + i + " = " + (numero + i) );
    }
  }
}
```

Ahora el programa de prueba en el que se crean estos objetos y se crean hilos basados en ellos. Existen dos alternativas además de las ya vistas para crear los hilos: creando un objeto de la clase anterior y pasándoselo a un constructor de la clase `Thread`, o declarando un objeto de la clase `Runnable` al que se le asigna un objeto de la clase anterior y pasando el objeto `Runnable` al constructor de `Thread`. En el siguiente ejemplo se ilustran todas las maneras, pero recomendamos crear los hilos a partir de objetos `Runnable` al tratarse del estándar, lo cual permite heredar de otras clases.

```
/**
 * Programa en Java que crea y lanza varios hilos utilizando la clase Cuenta50.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaCuenta50 {
  public static void main (String[] args) throws InterruptedException {
    Cuenta50 c1 = new Cuenta50(0);
    Runnable c2 = new Cuenta50(63);
    Runnable c3 = new Cuenta50(102);
    //Creamos los hilos
    Thread hilo1 = new Thread(c1);
    Thread hilo2 = new Thread(c2);
    //A continuación lanzamos los hilos
    hilo1.start();
    hilo2.start();
    //O creamos y lanzamos el hilo en una sola instrucción
    new Thread(c3).start();
    //Esperamos a que terminen los hilos
    hilo1.join();
    hilo2.join();
    System.out.println("Hilos terminados.");
  }
}
```

Es posible que al ejecutar este programa, observe que en la pantalla se entrelazan los resultados de los distintos hilos. Recuerde que este efecto es normal en la programación concurrente.

Ejercicio 2 Cree una clase que implemente la interfaz `Runnable` y almacene un número. Cuando un objeto de la clase sirva para crear y poner en marcha un hilo, debe imprimir sus 10 primeros múltiplos. Implemente también el programa de prueba necesario. Compruebe sus códigos con los del apartado 5.2.

2.3. *Thread Pools*

Existe otra forma de trabajar con hilos de modo que sea la máquina virtual de Java (JVM) quien gestione el ciclo de vida de los hilos. Así, el programador puede dedicarse a otros aspectos de la implementación con mayor dedicación. Pero este método también tiene sus costes. Veamos con más detalles en qué consiste este método.

Suponga que dispone de un conjunto de hilos creados pero que no están realizando ninguna tarea. Cuando necesita que se lleven a cabo ciertas tareas, informa a los hilos y pide que las ejecuten cuando puedan. Los hilos van haciéndose cargo de las tareas en cuanto pueden, las ejecutan y las devuelven cuando estén listas. Una vez que un hilo ha terminado de ejecutar una tarea, vuelve a estar inactivo, esperando a la siguiente. Pues en esto consiste la idea de un *thread pool*[1].

Existen 3 razones para el uso de los *thread pools*. Por una parte, reducen la latencia provocada por la creación de hilos ya que como los hilos son reutilizados, ahorramos el tiempo de creación del hilo que habría que dedicar si hubiera que crear un hilo para cada tarea. En segundo lugar, permiten un mejor diseño del programa al delegar todo el control de los hilos en el propio *thread pool*, dedicándose así mejor el programador a la lógica del programa. Por último, usar *thread pools* en aplicaciones que ejecutan varios hilos simultáneamente hace que se aproveche mejor la capacidad de procesamiento, ya que es posible obtener resultados parciales en menos tiempo del que se necesitaría lanzando un hilo por cada tarea y, además, al haber varios hilos creados en el *thread pool* se garantiza que la ejecución no se queda bloqueada.

Pero, como siempre, esta herramienta también tiene sus desventajas. Los hilos requieren memoria, y el uso de la memoria repercute en el rendimiento del sistema. Además, los *thread pools* introducen una pequeña sobrecarga al sistema operativo, que tiene que encargarse de controlar los hilos. Por otra parte, esta herramienta no es útil cuando no interesa el resultado parcial de cada hilo, sino que el resultado final se forma una vez que se ha completado la ejecución de todos los hilos. Por último, los *thread pools* no son necesarios cuando los recursos de la CPU son adecuados para manejar las necesidades del programa, ni introducen beneficios si el número de procesadores es mayor o igual al número de hilos que se crean.

Los *thread pools* son herramientas de desarrollo multihilo. Java incluye en sus bibliotecas su propia implementación de *thread pools* desde la versión Java SE 5.0. Para hacer uso de esa herramienta, necesitamos conocer, al menos, otras clases o interfaces que intervienen. El siguiente diagrama muestra las clases que explicaremos en este guión.

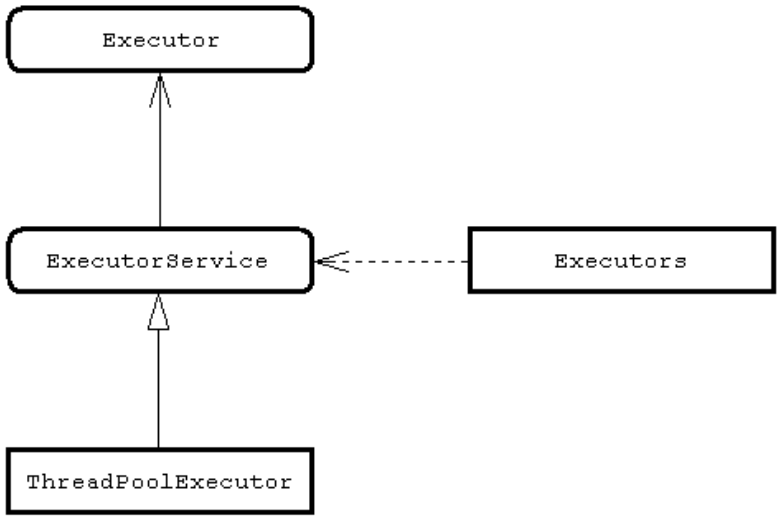

Figura 2: Diagrama de las clases que intervienen en los *thread pools*.

[1] Este término no tiene una traducción concreta en castellano y muchas veces se usa el término inglés. En algunos libros puede aparecer traducido como "conjunto de hilos", o puede traducirlo usando un diccionario como "fondo de hilos" o "reserva de hilos". Incluso hay personas que se refieren a un *thread pool* como "*pool* de *threads*". Por eso utilizaremos en este guión el término inglés cada vez que hagamos referencia a este concepto.

2.3.1. `Executor` y `ExecutorService`

Lo primero es saber qué es un ejecutor. El concepto de ejecutor (*executor* en inglés) es la base de la implementación de los *thread pools* en Java. Antes decíamos que cuando teníamos una tarea, pedíamos que se ejecutara. Pues bien, la interfaz `Executor` nos proporciona abstracción al pedir que una tarea sea ejecutada. Nosotros creamos un objeto `Runnable` y lo pasamos como argumento al método `execute()` de la interfaz `Executor`. Con este gesto, hemos pasado al sistema una tarea para que sea ejecutada. El sistema pasará la tarea al ejecutor apropiado. La interfaz `Executor` se encuentra en el paquete `java.util.concurrent`:

```
package java.util.concurrent;
public interface Executor {
  public void execute (Runnable task);
}
```

Hay varios tipos de ejecutores, pero todos están definidos por la interfaz `ExecutorService`. Esta interfaz extiende a `Executor` y añade otros métodos útiles para el control del ejecutor y de las tareas. Esta interfaz se encuentra también en el paquete `java.util.concurrent`. Como el objetivo de este apartado no es explicar a fondo esta interfaz, se muestra a continuación una versión parcial de la interfaz `ExecutorService`. Si necesita más información, consulte la documentación:

```
package java.util.concurrent;
public interface ExecutorService extends Executor {
  void shutdown();
  List shutdownNow();
  boolean isShutdown();
  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException;
  <T> Future<T> submit(Runnable task, T result);

  ...

}
```

2.3.2. Clase `ThreadPoolExecutor`

Y por fin llegamos a la clase que implementa los *thread pools*. La clase que implementa esta herramienta en Java se llama `ThreadPoolExecutor` e implementa la interfaz `ExecutorService`. La clase `ThreadPoolExecutor` modela un *thread pool*, que están en principio inactivos, y que a medida que el programa principal solicita que se realicen determinadas tareas (en forma de objetos `Runnable` como comentamos), éstos se ocupan de las tareas y dejan de estar inactivos. Al crear un objeto de la clase `ThreadPoolExecutor` se especifica cuántos hilos debe tener como mínimo y cuántos como máximo. Las tareas que se van solicitando, se van colocando en algún tipo de lista de espera, y cuando hay algún hilo libre, éste toma la tarea y la ejecuta. La implementación de la lista de espera de tareas también puede variar entre algunos tipos. Lo mismo ocurre con la política de creación de hilos. La clase `ThreadPoolExecutor` pertenece al paquete `java.util.concurrent`. Cabe mencionar que no tiene sentido tener un *thread pool* con un número de hilos menor o igual al número de procesadores. Más adelante comentaremos las ventajas y desventajas de esta herramienta.

Esta clase tiene varios constructores, que permiten elegir la forma en que se crean los hilos y el comportamiento que debe tener en el caso de rechazar que un hilo se añada a la lista de espera. Cuando al constructor de `ThreadPoolExecuor` no se le pasen objetos que manejen estos aspectos, se aplicará el comportamiento definido por defecto. Veamos los constructores posibles:

```
package java.util.concurrent;
public class ThreadPoolExecutor implements ExecutorService {

  public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
    long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue);
  public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
    long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
    RejectedExecutionHandler handler);
  public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
    long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory);
  public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
    long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory, RejectedExecutionHandler handler);

  ...
}
```

El argumento que define la creación de hilos en el constructor de la clase `ThreadPoolExecutor` es el parámetro de la clase `ThreadFactory`. Si no es especificado, los hilos se crearán por defecto sin ninguna característica especial. Por otra parte, el argumento de la clase `RejectedExecutionHandler` es quien indica qué hacer cuando un hilo no pueda ser añadido a la lista de hilos por ejecutar. Por defecto, se lanzará una excepción.

El tamaño del *thread pool* y de la lista de espera también influyen en el comportamiento del objeto de la clase `ThreadPoolExecutor`. Estos valores también son estipulados en el constructor. El primer argumento del constructor es el número mínimo de hilos (*core threads*) que el *thread pool* puede tener, mientras que el segundo argumento es el número máximo de hilos.

Cuando un objeto de la clase `ThreadPoolExecutor` es creado, no crea ningún hilo. Aunque esta clase proporciona un método (`prestartAllCoreThreads()`) que permite, una vez creado el *thread pool*, que se creen todos los hilos y permanezcan inactivos. Según se le van pasando tareas que ejecutar va creando hilos hasta alcanzar el número mínimo de hilos. Las sucesivas tareas a ejecutar provocarán la creación de más hilos o utilizarán alguno inactivo según el número de hilos que haya. Cuando una tarea termina su ejecución, el hilo queda libre y espera o se destruye según el número de tareas que haya en espera.

La clase `ThreadPoolExecutor` proporciona también métodos para interactuar con el tamaño del *thread pool* y con la lista de tareas, entre otros métodos. Veamos algunos de los métodos de la clase:

```
package java.util.concurrent;
public class ThreadPoolExecutor implements ExecutorService {
  ...

  protected void afterExecute(Runnable r, Throwable t);
  protected void beforeExecute(Thread t, Runnable r);
  public void execute(Runnable command);
  public int getActiveCount();
```

```
  public long getCompletedTaskCount();
  public int getCorePoolSize();
  public int getMaximumPoolSize();
  public int getPoolSize();
  public BlockingQueue<Runnable> getQueue();
  public long getTaskCount();
  public int prestartAllCoreThreads();
  public boolean remove(Runnable task);
  public void setCorePoolSize(int corePoolSize);
  public void setMaximumPoolSize(int maximumPoolSize);
  public void shutdown();
  ...
}
```

- `protected void afterExecute(Runnable r, Throwable t)`: método que será llamado después de que se termine la tarea indicada por el objeto `Runnable`.
- `protected void beforeExecute(Thread t, Runnable r)`: método que será invocado por el hilo `t` donde se ejecutará la tarea `r` antes de que ésta sea ejecutada.
- `public void execute(Runnable command)`: añade una tarea al *thread pool* para que sea ejecutada.
- `public int getActiveCount()`: indica el número aproximado de hilos que están en activo en ese momento.
- `public long getCompletedTaskCount()`: indica el número aproximado de tareas que ya han sido completadas.
- `public int getCorePoolSize()`: indica el número mínimo de hilos del *thread pool*.
- `public int getMaximumPoolSize()`: indica el número máximo de hilos del *thread pool*.
- `public int getPoolSize()`: indica el número de hilos que tiene actualmente el *thread pool*.
- `public BlockingQueue<Runnable> getQueue()`: devuelve la lista de tareas usada por este ejecutor.
- `public long getTaskCount()`:
- `public int prestartAllCoreThreads()`: crea hilos hasta llegar al número mínimo de hilos del *thread pool* dejándolos en inactivo. Devuelve el número de hilos creados.
- `public boolean remove(Runnable task)`: elimina la tarea de la lista de tareas del *thread pool*, si está en la lista y no ha empezado aún su ejecución.
- `public void setCorePoolSize(int corePoolSize)`: configura el número mínimo de hilos del *thread pool*.
- `public void setMaximumPoolSize(int maximumPoolSize)`: configura el número máximo de hilos del *thread pool*.
- `public void shutdown()`: inicia un apagado ordenado del *thread pool*. Las tareas que queden pendientes en la lista de tareas no serán ejecutadas, mientras que las que ya están siendo ejecutadas terminan su ejecución antes de parar todos los hilos.

Tener un *thread pool* en nuestro programa es tan sencillo como construir un objeto de la clase `ThreadPoolExecutor` e invocar su método `execute()` pasándole una tarea, cada vez que haya una nueva tarea que ejecutar. A continuación podemos ver un programa de ejemplo que utiliza la clase `Task` que incluimos:

```
/**
 * 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);
  }
}


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

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

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

    System.out.print("Introduzca el número de tareas a ejecutar: ");
    int nTasks = teclado.nextInt();
    System.out.print("Introduzca el número de hilos que tendrá la reserva de " +
      "hilos: ");
    int tpSize = teclado.nextInt();

    //Creamos el thread pool
    ThreadPoolExecutor tpe = new ThreadPoolExecutor(tpSize, tpSize, 50000L,
```

```
      TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());

    //Creamos un vector de tareas de la clase anterior.
    Task[] tasks = new Task[nTasks];
    for(int i = 0; i < nTasks; ++i) {
      tasks[i] = new Task(i);
      System.out.println("Tarea " + i + " a ejecutar.");
      //Pasamos la nueva tarea recién creada al thread pool para que sea ejecutada.
      tpe.execute(tasks[i]);
    }
    tpe.shutdown();
  }
}
```

En este programa de ejemplo solicitamos el número de tareas que queremos ejecutar y el número exacto de hilos que queremos que tenga el *thread pool*. A continuación, en el constructor, creamos un *thread pool* de tamaño fijo dado que el número mínimo y el número máximo de hilos que puede tener es el mismo. Además, los hilos permanecerán vivos 50 segundos tras pasar a inactivo. Si en este tiempo no han encontrado una nueva tarea que ejecutar, los hilos desaparecen. Las tareas se guardan en una lista del tipo `LinkedBlockingQueue`, que es una lista sin límites de capacidad, por tanto todas las tareas serán añadidas a la lista. El tipo de lista de tareas y el número mínimo y máximo de hilos permitidos para el *thread pool* determinan el comportamiento del mismo en los casos extremos. Luego se van creando tareas y mandado a ejecutar. Una vez que se han mandado todas las tareas, se espera a que finalicen antes de eliminar el *thread pool*.

Ejercicio 3 Compile y ejecute el código anterior. Ejecútelo varias veces, dándole distintos valores de entrada. Observe atentamente el resultado de cada ejecución.

2.3.3. Clase `Executors`

Como ha podido comprobar, crear objetos de la clase `ThreadPoolExecutor` puede ser demasiado complejo para nuestros intereses actuales, ya que hay que tener en cuenta qué tipo de lista se le pasa al constructor (aspecto que no hemos detallado aquí), las características que queremos que tengan los hilos que se creen o la política a tener en cuenta con las tareas que sean rechazadas. Para abstraernos de esos detalles, podemos usar los métodos que nos proporciona la clase `Executors`.

La clase `Executors` se encuentra también en el paquete `java.util.concurrent` y proporciona métodos de utilidad para las clases o interfaces definidas antes. Aunque se trata de una clase, no tiene constructores, sino que se trata de una clase de utilidades que contiene métodos para crear objetos predefinidos de otras clases (como `ExecutorService` o `ScheduledExecutorService`, entre otras). Veamos tres métodos de la clase `Executors` que nos permiten crear *thread pools* sin tener que preocuparnos tanto de los detalles de implementación, ya que devuelve objetos con características predefinidas, como siguiendo una plantilla.

- `static ExecutorService newCachedThreadPool()`: crea hilos a medida que recibe tareas hasta que el *thread pool* llega a su tamaño máximo. Luego intenta mantener el tamaño del *thread pool* constante, añadiendo hilos si pierde alguno debido a una excepción no esperada.
- `static ExecutorService newFixedThreadPool(int nThreads)`: cuando el tamaño del *thread pool* excede del tamaño necesario para la demanda de procesamiento actual, elimina los hilos inac-

tivos. Mientras que cuando la demanda aumenta, añade hilos sin límite de tamaño en el *thread pool*.

- `static ExecutorService newSingleThreadExecutor()`: crea un solo hilo para procesar tareas, siendo reemplazado si termina inesperadamente. Se garantiza que las tareas son procesadas secuencialmente según el orden impuesto por la lista de tareas.

Veamos ejemplos de cómo usar estas funciones. Para estos ejemplos, usaremos la clase `Task` mencionada anteriormente y crearemos un programa de ejemplo similar a `UsaThreadPoolExecutor`. El primer ejemplo utiliza el método `newCachedThreadPool()`:

```
/**
 * 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);
    ExecutorService exec = Executors.newCachedThreadPool();

    System.out.print("Introduzca el número de tareas a ejecutar: ");
    int nTasks = teclado.nextInt();

    Task[] tasks = new Task[nTasks];
    for(int i = 0; i < nTasks; ++i) {
      tasks[i] = new Task(i);
      System.out.println("Tarea " + i + " a ejecutar.");
      exec.execute(tasks[i]);
    }
    exec.shutdown();
  }
}
```

El siguiente ejemplo usa el método `newFixedThreadPool()`:

```
/**
 * Programa de prueba para ilustrar el uso de la clase Executors.
 *
 * @author Natalia Partera
 * @version 2.0
 */

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 tareas a ejecutar: ");
    int nTasks = teclado.nextInt();
    System.out.print("Introduzca el número de hilos que tendrá la reserva de " +
      "hilos: ");
    int tpSize = teclado.nextInt();

    ExecutorService exec = Executors.newFixedThreadPool(tpSize);

    Task[] tasks = new Task[nTasks];
    for(int i = 0; i < nTasks; ++i) {
      tasks[i] = new Task(i);
      System.out.println("Tarea " + i + " a ejecutar.");
      exec.execute(tasks[i]);
    }
    exec.shutdown();
  }
}
```

El último ejemplo usa el método `newSingleThreadExecutor()`:

```
/**
 * Programa de prueba para ilustrar el uso de la clase Executors.
 *
 * @author Natalia Partera
 * @version 3.0
 */

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

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

    System.out.print("Introduzca el número de tareas a ejecutar: ");
    int nTasks = teclado.nextInt();

    Task[] tasks = new Task[nTasks];
    for(int i = 0; i < nTasks; ++i) {
      tasks[i] = new Task(i);
      System.out.println("Tarea " + i + " a ejecutar.");
      exec.execute(tasks[i]);
    }
```

```
    exec.shutdown();
  }
}
```

Ejercicio 4 Cree una clase `Tarea` que cuando sea ejecutada imprima 15 veces su nombre. Cree un programa donde se cree un vector de tareas y se ejecuten en hilos. Utilice las funciones explicadas de la clase `Executors` para almacenar los hilos. Cree una versión del programa para cada función. Compare sus versiones con el código que se incluye en 5.3.

---

3. Comparativa de los Métodos de Multithreading

El primer método que hemos visto para lanzar hilos es haciendo que nuestra clase herede de `Thread`. Este método tiene una gran desventaja, y es que al heredar de `Thread`, nuestra clase ya no puede heredar de ninguna otra. La alternativa es que la clase implemente la interfaz `Runnable`.

Si recordamos, la interfaz `Runnable` tan sólo posee el método `run()` y además la clase `Thread` implementa la interfaz `Runnable`. Así pues, usemos `Thread` o `Runnable` con nuestra clase, siempre debemos implementar el método `run()`.

Los objetos que implementan la interfaz `Runnable` deben ser lanzados explícitamente. Para ello, se crea un objeto `Thread` cuyo parámetro es el objeto `Runnable`. A continuación se llama al método `start()` del objeto `Thread`.

En conclusión, para lanzar hilos sueltos la técnica recomendada es hacer que la clase deseada implemente la interfaz `Runnable`.

En cuando a los *thread pools*, existen varias formas de crearlos y varias clases que aportan funcionalidades. Podemos crear *thread pools* más personalizados con la clase `ThreadPoolExecutor`. El inconveniente es que es más difícil de utilizar y que el programador debe controlar todo el ciclo de vida del *thread pool*.

Otra forma más sencilla de crear *threads pools* es con los métodos de la clase `Executors`. Utilizando esta clase, el programador puede desentenderse de tener que controlar el ciclo de vida del *thread pool*. Además la clase `Executors` proporciona otros métodos de utilidad que no hemos visto en este documento.

Independientemente de la creación del *thread pool*, para mandar una tarea a que sea ejecutada se utiliza el método `execute()` de la interfaz `Executor`. Interfaz que tanto `ThreadPoolExecutor` como `Executors` implementan, y que recibe un atributo `Runnable`.

Para finalizar, lo más sencillo para crear un *thread pool* es utilizar los métodos de la clase `Executors`. Además, implementar hilos con `Runnable` como dijimos anteriormente, nos resulta muy útil al implementar un *thread pool* con `Executors`, ya que el método que manda una tarea a ejecutar recibe un objeto `Runnable`.

4. Ejercicios

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

Ejercicio 5 Rehaga la clase `Palabra` que aparece como ejemplo en el apartado 2.1 implementándola como `Runnable`. Cree un programa de prueba en el que lance algunas instancias de la clase. Compare sus resultados con los que aparecen en el apartado 5.4.

Ejercicio 6 Modifique el programa de prueba del ejercicio anterior para que cree un *thread pool* usando alguno de los métodos de la clase `Executors`. Compare el programa con el que puede ver en el apartado 5.5.

Ejercicio 7 Basándose en el ejercicio 1, implemente una clase cuyo constructor reciba dos números y realice operaciones básicas entre estos números. Implemente un *thread pool* usando la función `newFixedThreadPool()`. Cuando lo termine, puede comparar su código con el que se haya en 5.6.

Ejercicio 8 Basándose en el ejemplo del apartado 2.2, implemente una clase que dado un número, le sume 49 indicándolo número por número. Implemente un *thread pool* usando la función `newCachedThreadPool()`. Compruebe sus códigos con los que encontrará en el apartado 5.7.

Ejercicio 9 Basándose en el ejercicio 2, implemente una clase cuyo constructor reciba un número y al ejecutarse muestre los 10 primeros múltiplos de dicho número. Implemente un *thread pool* usando la función `newSingleThreadExecutor()`. Una vez finalizado, compárelo con el código del apartado 5.8.

---

5. Soluciones de los ejercicios

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

5.1. Ejercicio 1

Compare su clase que almacena números con la siguiente:

```
/**
 * Clase que representa almacena un par de números y realiza operaciones básicas
 * entre ellos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.lang.*;

public class OpBasicas extends Thread {
  int x, y;

  OpBasicas() {}
  OpBasicas(int num1, int num2) {
    x = num1;
    y = num2;
  }

  public void run() {
    System.out.println(x + " + " + y + " = " + (x + y) );
    System.out.println(x + " - " + y + " = " + (x - y) );
    System.out.println(x + " * " + y + " = " + (x * y) );
    System.out.println(x + " / " + y + " = " + (x / y) );
  }
}
```

Su programa de prueba debe parecerse al siguiente:

```
/**
 * Programa en Java que lanza varios hilos realizando operaciones básicas entre
 * dos números.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaOpBasicas {
  public static void main (String[] args) throws InterruptedException {
    OpBasicas hilo1 = new OpBasicas(0, 8);
    OpBasicas hilo2 = new OpBasicas(3, 6);
    OpBasicas hilo3 = new OpBasicas(82, 17);
    //A continuación lanzamos los hilos
    hilo1.start();
```

```
    hilo2.start();
    hilo3.start();
    //Esperamos a que terminen los hilos
    hilo1.join();
    hilo2.join();
    hilo3.join();
    System.out.println("Hilos terminados.");
  }
}
```

5.2. Ejercicio 2

A continuación puede ver la clase que implementa `Runnable`:

```
/**
 * Clase que almacena un número y calcula sus 10 primeros múltiplos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

import java.lang.*;

public class Mult10 implements Runnable {
  int numero;

  Mult10() {}
  Mult10(int num) {
    numero = num;
  }

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

Y este es el programa de prueba y creación de hilos para la clase anterior:

```
/**
 * Programa en Java que crea y lanza varios hilos utilizando la clase Mult10.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaMult10 {
  public static void main (String[] args) throws InterruptedException {
    Runnable m1 = new Mult10(12);
    Runnable m2 = new Mult10(5);
```

```
    Runnable m3 = new Mult10(28);
    //Creamos los hilos
    Thread hilo1 = new Thread(m1);
    Thread hilo2 = new Thread(m2);
    //A continuación lanzamos los hilos
    hilo1.start();
    hilo2.start();
    //O creamos y lanzamos el hilo en una sola instrucción
    new Thread(m3).start();
    //Esperamos a que terminen los hilos
    hilo1.join();
    hilo2.join();
    System.out.println("Hilos terminados.");
  }
}
```

5.3. Ejercicio 4

Aquí puede ver la clase `Tarea`:

```
/**
 * Clase Tarea que imprime varias veces su nombre al ser ejecutada.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Tarea extends Thread {
  private int id;

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

  public void run() {
    String name = getName();
    for(int i = 0; i < 15; ++i) {
      System.out.println("Vuelta " + i + " de " + name + " ");
    }
  }
}
```

Este es el programa de prueba usando el método `newCachedThreadPool()`:

```
/**
 * Programa de prueba para la clase Tarea.
 *
 * @author Natalia Partera
 * @version 1.0
 */
```

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

public class UsaTareas
{
  public static void main(String[] args)
  {
    Scanner teclado = new Scanner(System.in);
    ExecutorService exec = Executors.newCachedThreadPool();

    System.out.print("Introduzca el número de tareas a ejecutar: ");
    int numTareas = teclado.nextInt();

    Tarea[] tareas = new Tarea[numTareas];
    for(int i = 0; i < numTareas; ++i) {
      tareas[i] = new Tarea();
      System.out.println("Tarea " + i + " a ejecutar.");
      exec.execute(tareas[i]);
    }
    exec.shutdown();
  }
}
```

El programa de prueba que usa el método `newFixedThreadPool()` es el que sigue:

```
/**
 * Programa de prueba para la clase Tarea.
 *
 * @author Natalia Partera
 * @version 2.0
 */

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

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

    System.out.print("Introduzca el número de tareas a ejecutar: ");
    int numTareas = teclado.nextInt();
    System.out.print("Introduzca el número de hilos que tendrá la reserva de " +
      "hilos: ");
    int tpSize = teclado.nextInt();

    ExecutorService exec = Executors.newFixedThreadPool(tpSize);

    Tarea[] tareas = new Tarea[numTareas];
    for(int i = 0; i < numTareas; ++i) {
```

```
      tareas[i] = new Tarea();
      System.out.println("Tarea " + i + " a ejecutar.");
      exec.execute(tareas[i]);
    }
    exec.shutdown();
  }
}
```

Aquí puede ver el programa de prueba en el que se usa el método `newSingleThreadExecutor()`:

```
/**
 * Programa de prueba para la clase Tarea.
 *
 * @author Natalia Partera
 * @version 3.0
 */

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

public class UsaTareas
{
  public static void main(String[] args)
  {
    Scanner teclado = new Scanner(System.in);
    ExecutorService exec = Executors.newSingleThreadExecutor();

    System.out.print("Introduzca el número de tareas a ejecutar: ");
    int numTareas = teclado.nextInt();

    Tarea[] tareas = new Tarea[numTareas];
    for(int i = 0; i < numTareas; ++i) {
      tareas[i] = new Tarea();
      System.out.println("Tarea " + i + " a ejecutar.");
      exec.execute(tareas[i]);
    }
    exec.shutdown();
  }
}
```

5.4. Ejercicio 5

Esta es la clase `Palabra` tras las modificaciones:

```
/**
 * Clase que representa a una palabra y comprueba si es palíndromo.
 *
 * @author Natalia Partera
 * @version 1.0
 */
```

```
import java.lang.*;

public class PalabraRun implements Runnable {
  String palabra;

  PalabraRun() {}
  PalabraRun(String pal) {
    palabra = pal;
  }

  public void run() {
    boolean palindromo = true;
    int i = 0;
    int j = palabra.length() - 1;

    while(palindromo && i < j) {
      if(palabra.charAt(i) == palabra.charAt(j)) {
        ++i;
        --j;
      }
      else
        palindromo = false;
    }
    if(palindromo)
      System.out.println("La palabra " + palabra + " es un palíndromo.");
    else
      System.out.println("La palabra " + palabra + " no es un palíndromo.");
  }
}
```

Y este es el programa de prueba creado para la clase anterior:

```
/**
 * Programa en Java que lanza varios objetos comprobando si varias palabras son
 * palíndromos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaPalabraRun {
  public static void main (String[] args) throws InterruptedException {
    Runnable pal1 = new Palabra("casa");
    Runnable pal2 = new Palabra("ala");
    Runnable pal3 = new Palabra("Oso");

    new Thread(pal1).start();
    new Thread(pal2).start();
    new Thread(pal3).start();
    System.out.println("Hilos terminados.");
  }
}
```

5.5. Ejercicio 6

Este es el código modificado que usa el método `newSingleThreadExecutor()`:

```
/**
 * Programa en Java que lanza varios objetos comprobando si varias palabras son
 * palíndromos.
 *
 * @author Natalia Partera
 * @version 2.0
 */

import java.util.concurrent.*;

public class UsaPalabraRun {
  public static void main (String[] args) {
    String[] palabras = new String[] {"casa", "ala", "ama", "oso", "leer", "sos"};
    int num = palabras.length;

    PalabraRun pal[] = new PalabraRun[num];
    ExecutorService exec = Executors.newSingleThreadExecutor();

    for(int i = 0; i < num; ++i) {
      pal[i] = new PalabraRun(palabras[i]);
      System.out.println("Palabra " + palabras[i] + " creada.");
      exec.execute(pal[i]);
    }
    exec.shutdown();
  }
}
```

5.6. Ejercicio 7

Esta es la clase `OpBasicas` modificada:

```
/**
 * Clase que representa almacena un par de números y realiza operaciones básicas
 * entre ellos.
 *
 * @author Natalia Partera
 * @version 2.0
 */

import java.lang.*;

public class OpBasicasExec implements Runnable {
  int x, y;

  OpBasicasExec() {}
  OpBasicasExec(int num1, int num2) {
    x = num1;
    y = num2;
```

```
  }

  public void run() {
    System.out.println(x + " + " + y + " = " + (x + y) );
    System.out.println(x + " - " + y + " = " + (x - y) );
    System.out.println(x + " * " + y + " = " + (x * y) );
    System.out.println(x + " / " + y + " = " + (x / y) );
  }
}
```

Y este es el programa de prueba para la clase anterior que utiliza el método `newFixedThreadPool()`:

```
/**
 * Programa en Java que lanza varios hilos realizando operaciones básicas entre
 * dos números.
 *
 * @author Natalia Partera
 * @version 2.0
 */

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

public class UsaOpBasicasExec {
  public static void main (String[] args) {
    Scanner teclado = new Scanner(System.in);
    int x = 3, y = 8;

    System.out.println("Indique cuántas tareas quiere ejecutar: ");
    int numTareas = teclado.nextInt();
    System.out.println("Indique cuántos hilos quiere crear: ");
    int numHilos = teclado.nextInt();

    ExecutorService exec = Executors.newFixedThreadPool(numHilos);

    OpBasicasExec[] tareas = new OpBasicasExec[numTareas];
    for(int i = 0; i < numTareas; ++i) {
      x = 3;
      y = 8;
      tareas[i] = new OpBasicasExec(x * (i+1), y * (i+1));
      System.out.println("Tarea " + i + " creada.");
      exec.execute(tareas[i]);
    }
    exec.shutdown();
  }
}
```

5.7. Ejercicio 8

La clase `Cuenta50` no sufre cambios, y es la misma que se muestra en el apartado 2.2. El programa de prueba cambia un poco. Esta es la nueva versión:

```
/**
 * Programa en Java que crea y lanza varios hilos utilizando la clase Cuenta50.
 *
 * @author Natalia Partera
 * @version 2.0
 */

import java.util.concurrent.*;

public class UsaCuenta50Exec {
  public static void main (String[] args) {
    int[] nums = new int[] {45, 320, 887, 11, 2398, 402, 674, 3057, 1823, 5956};
    int numTareas = nums.length;

    ExecutorService exec = Executors.newCachedThreadPool();

    Cuenta50[] tareas = new Cuenta50[numTareas];
    for(int i = 0; i < numTareas; ++i) {
      tareas[i] = new Cuenta50(nums[i]);
      System.out.println("Tarea " + i + " creada.");
      exec.execute(tareas[i]);
    }
    exec.shutdown();
  }
}
```

5.8. Ejercicio 9

Para realizar este ejercicio, la clase `Mult10` del ejercicio 2 no necesita ser modificada. Por tanto, sólo cambia el programa de prueba:

```
/**
 * Programa en Java que crea y lanza varios hilos utilizando la clase Mult10.
 *
 * @author Natalia Partera
 * @version 2.0
 */

import java.util.concurrent.*;

public class UsaMult10Exec {
  public static void main (String[] args) {

    int[] nums = new int[] {45, 320, 887, 11, 2398, 402, 674, 3057, 1823, 5956};
    int numTareas = nums.length;

    ExecutorService exec = Executors.newSingleThreadExecutor();

    Mult10[] tareas = new Mult10[numTareas];
    for(int i = 0; i < numTareas; ++i) {
      tareas[i] = new Mult10(nums[i]);
```

```
      System.out.println("Tarea " + i + " creada.");
      exec.execute(tareas[i]);
    }
    exec.shutdown();
  }
}
```