1. Threads

1.1. Threads en Java

Los Threads, hilos de ejecución o procesos ligeros, son las unidades más pequeñas de procesamiento que pueden ser programadas por los sistemas operativos, y que permiten a un mismo proceso ejecutar diferentes tareas de forma simultánea. Cada hilo de ejecución ejecuta una tarea concreta y ofrece así al programador la posibilidad de diseñar programas que ejecutan funciones diferentes de forma concurrente.

Esta técnica de programación con hilos se conoce como **multithreading** o **multihilo**, y permite simplificar el diseño de aplicaciones concurrentes y mejorar el rendimiento en la creación de procesos.

Para crear threads en Java podemos optar por dos opciones:

- Crear una clase que herede de la clase **Thread** (java.lang.Thread), o bien
- crear una clase que implemente la interfaz **Runnable** (java.lang.Runnable).

Aunque ambos mecanismos son prácticamente equivalentes, la opción recomendada en la documentación de Java es utilizar la interfaz **Runnable**. Recordemos que Java no soporta herencia múltiple, y que lo más parecido que ofrece son las interfaces. Si creamos una clase que descienda directamente de Thread, no podrá descender de ninguna otra. Así pues, si disponemos de una clase que pertenece a una jerarquía y que además tenga las características y se comporte como un Thread, debemos utilizar la interfaz Runnable.

1.2. Creación de hilos mediante la clase Thread

Como hemos comentado, una de las formas de crear un hilo es creando una clase que descienda de la clase Thread. Además, esta clase implementará un método run() que contendrá el código con la lógica que se ejecutará en hilo.

- El código correspondiente para ello sería:

```
public class MiHilo extends Thread{
   public void run(){
   // lógica interna del hilo
   }
}
```

- Y ahora, en otra clase, inicializamos y ejecutamos el hilo:

```
public class Principal {public static void main(String[] args) {
      // Creamos el nuevo thread
      MiHilo t=new MiHilo();
      // Lo ponemos en ejecución
      t.start();
   }
}
```

IMPORTANTE

Observad que la lógica del hilo se define en el método run(), pero la invocación la realizamos mediante el método start(), que es el que hace posible la ejecución asíncrona de la lógica indicada en el método run(). Si se invocase el método run() directamente, su lógica se ejecutaría de forma síncrona.

1.3. Creación de hilos mediante la interfaz Runnable

Tal y como hemos comentado, la forma recomendable para crear un Thread no es definiendo una subclase de Thread, sino definiendo una clase que implemente la interfaz Runnable. Para ello haremos lo siguiente:

- **Definir una clase** que implemente la interfaz Runnable y sobreescribir el método public void run(), con la lógica interna del hilo:

  ```
  public class ClaseRunnable implements Runnable {

        // Declaración de atributos y métodos

        @Override

        public void run() {

        // lógica interna del hilo

        }

  }
  ```

- **Crear el Thread**. Para ello, creamos una instancia de la clase ClaseRunnable y se la proporcionamos al constructor de la clase Thread para crear el nuevo hilo:

  ```
  ClaseRunnable objetoRunnable=new ClaseRunnable();
  Thread hilo=new Thread(objetoRunnable);
  ```

- **Lanzar el hilo**, invocando al método start():
  ```
  hilo.start();
  ```

IMPORTANTE

La clase Thread admite múltiples constructores. En nuestro caso, estamos utilizando un constructor al que le proporcionamos una clase que implementa la interfaz Runnable, pero también podemos proporcionarle un nombre al Thread, e incluso un nombre de grupo.

A. CREACIÓN DE MÚLTIPLES HILOS

En algunas ocasiones necesitaremos controlar varios hilos de ejecución en nuestra aplicación. En este caso deberemos utilizar estructuras de datos como vectores o listas para almacenarlos.

- **Creación del vector o lista:**
  Thread vector_hilos[]=new Thread[longitud];
  ArrayList<Thread> lista_hilos=new ArrayList<Thread>();
- **Creación de los hilos:**

  Thread hilo=new Thread(objetoRunnable);
  vector_hilos[posicion_i]=hilo;
  lista_hilos.add(hilo);
  // O bien todo en la misma orden:
  vector_hilos[posicion_i]=new Thread(objetoRunnable);
  lista_hilos.add(new Thread(objetoRunnable));

- **Lanzamiento de los hilos:**

  vector_hilos[posicion].start();
  lista_hilos.get(0).start();

B. THREADS CON CLASES ANÓNIMAS

Las clases anónimas en Java nos permiten crear objetos que implementen cierta interfaz sin necesidad de crear una clase específicamente para que implemente dicha interfaz. Esto nos puede ser útil, por ejemplo, cuando solamente vamos a necesitar un objeto de dicha clase que va a ser de uso inmediato. En nuestro caso, esto se traduce en que, si no vamos a necesitar controlar varios hilos, podemos crear directamente un Thread mediante una clase anónima que implemente la interfaz Runnable.

- Esto se podría hacer del siguiente modo:

```
Thread mi_thread = new Thread(new Runnable() {
        @Override
        public void run() {
        // Implementación de la lógica interna
        }
});
mi_thread.start();
```

- Además, a partir de Java 8, podemos utilizar estas clases anónimas mediante expresiones lambda:

```
Thread mi_thread = new Thread(() -> {
        System.out.println(Thread.currentThread().getState());
        // Implementación de la lógica interna
});
mi_thread.start();
```

Con estas dos formas, como vemos, no hemos necesitado crear una clase específica que implemente la interfaz Runnable.

1.4. El método join

Todos los programas multihilo poseen un hilo principal. Cuando creamos un nuevo hilo en nuestro programa, se produce una bifurcación en su ejecución, que separa el hilo original del nuevo hilo.

A partir de esta bifurcación, la ejecución de ambos hilos es independiente, de modo que cada uno puede realizar sus funciones en paralelo. Es posible que, en un momento dado, el hilo principal deba esperar a que finalicen los hilos que ha creado para seguir con su operativa, o bien para finalizar. En este caso deberemos poner al hilo principal en espera, hasta que terminen los hilos que ha creado y se pueda unificar la ejecución de ambos. Esto se consigue mediante el método join().

El método join() deja al hilo que lo ha invocado en estado de espera, hasta que el hilo referenciado termina. El código para ello sería:

```
Thread hilo1=new Thread(objetoRunnable_1);
Thread hilo2=new Thread(objetoRunnable_2);

hilo1.start();
hilo2.start();
//…
hilo1.join();
hilo2.join();
```

Como vemos, se generan dos hilos a partir de dos instancias de la clase ClaseRunnable (objetoRunnable_1 y objetoRunnable_2), se ejecutan y, posteriormente, se espera a que finalicen ambos mediante el join().

Gráficamente podemos ver esta relación del siguiente modo:
Como vemos, el hilo principal lanza los dos hilos en momentos diferentes y, posteriormente, invoca al método join de ambos hilos para esperar su finalización. Dado que el hilo1 ya ha terminado cuando se invoca el join, se unifica inmediatamente con el hilo principal. Aun así, el hilo principal sigue en espera hasta que finaliza el hilo2. Cuando este finaliza, se realiza la unificación con el hilo principal, y sigue la ejecución de este.

2. Métodos sincronizados: synchronized

2.1. Objetos compartidos y sincronización

El desarrollo de aplicaciones multihilo comporta la ejecución de varios hilos de forma concurrente. La forma más habitual de mantener la comunicación entre estos hilos, así como con el hilo principal, es mediante el uso de variables u objetos compartidos, a los cuales todos ellos tienen acceso.

A estos objetos o variables compartidos por varios hilos de ejecución se les conoce como objetos o variables en conflicto y, además, la parte de código que accede a ellos se conoce como sección crítica.

Formalmente, podríamos definir estos términos como:

- **Variable/Objeto en conflicto:** variable u objeto que son compartidos por varios hilos de ejecución.
- **Sección crítica:** se refiere a aquella parte del código fuente de un hilo que realiza acciones sobre una variable u objeto en conflicto.

El problema ahora es poder garantizar que el acceso a estos objetos compartidos se realice de forma segura. Necesitaremos establecer algún mecanismo para que estas operaciones sobre los objetos se realicen de forma atómica, de modo que hasta que no finalice una operación sobre el objeto no empiece otra.

Esta coordinación en la actividad de varios hilos se conoce como **sincronización**.

2.2. Sincronización mediante monitores

En Java, esta sincronización es posible gracias al uso de monitores, un mecanismo que permite controlar el acceso concurrente a objetos en conflicto. Los monitores tienen la capacidad de bloquear un objeto mientras un hilo está accediendo a él, de modo que no permite el acceso por parte de otros hilos. Esto se conoce como **exclusión mutua**. Cuando el hilo que bloqueaba el objeto termina sus operaciones, desbloquea el objeto para que puedan acceder otros objetos. De este modo podemos sincronizar el acceso a un objeto compartido.

Ahora bien, ¿cómo se implementa todo esto? En Java, todos los objetos tienen asociado un monitor, por lo que pueden sincronizarse. Para acceder a este monitor se utiliza la palabra reservada synchronized, que podemos emplear de dos formas:

- Para declarar **métodos como synchronized** en el objeto en conflicto, de modo que se entra en el monitor cuando se acceda a él, asegurando así una exclusión mutua.
- Para declarar solamente una **sección de código como synchronized**, de modo que entramos en el monitor sobre el objeto compartido dentro de dicha sección, asegurando así el acceso exclusivo a este.

Esta segunda opción es la más recomendable, ya que es preferible sincronizar la mínima parte de código que sea posible.

A. MÉTODOS SINCRONIZADOS

Para sincronizar un método utilizamos la palabra reservada synchronized en su declaración:

```
class Compartido {
   private dato_compartido;
   [modificador] synchronized tipoRetorno NombreMetodo(lista_argumentos) {
   // Acceso al dato compartido
   }
}
```

De este modo, cuando el método NombreMetodo es invocado, el hilo que lo invocó entra en el monitor del objeto, quedando este bloqueado. En este momento, ningún otro hilo

que intente acceder al objeto compartido ya sea a través de este método sincronizado u otros también sincronizados, podrá hacerlo, garantizando así la exclusión mutua.

En el momento en el que el hilo que bloqueó el objeto finalice el método sincronizado, el monitor desbloquea el objeto, de modo que ya puede acceder a él otro hilo.

Con este mecanismo, únicamente debemos preocuparnos de marcar como sincronizados los métodos que vayan a contener una sección crítica.

B. BLOQUES SYNCHRONIZED

Además de poder especificar un método en el objeto compartido como synchronized, podemos definir únicamente un bloque de código, del siguiente modo:

```
synchronized(objeto_en_conflicto) {
   // Acceso al objeto
   // (Sección crítica)
}
```

Esto tendría dos posibles usos:

- **Cuando no sea posible definir un método como synchronized**, por ejemplo, porque es una función de librería que no está definida como tal, pero debemos utilizarla de forma sincronizada. Dado que no podemos marcar el método como sincronizado, lo enmarcaríamos en un bloque synchronized:

  ```
  synchronized(objeto_en_conflicto) {
  Función_No_Sincronizada(lista_parametros);
  }
  ```

- **Cuando deseamos reducir el bloqueo únicamente a la sección crítica**. En algunos métodos, la sección crítica será únicamente una parte de este, por lo que no necesitaremos declararlo en su totalidad como synchronized. En este caso encerraríamos en un bloque synchronized a esta sección crítica.

  ```
  class MiClase implements Runnable {
     ModificadorAcceso tipo NombreMetodo(argumentos) {
        // Lógica del método
        synchronized(objeto_en_conflicto) {
           //Sección crítica
        }
     }
     run() {
        //…
        synchronized(objeto_en_conflicto) {
           //Sección crítica
        }
     }
  }
  ```

Como podemos apreciar, podemos incluir también un bloque synchronized directamente dentro del método run(), como en cualquier otro método en el que se utilice run.

C. SYNCHRONIZED Y ESTADOS

En el apartado anterior vimos el ciclo de vida de un hilo, con los diferentes estados que este puede tomar. En él existía el estado **bloqueado**, al que se accedía mediante la espera/bloqueo de un monitor.

Centrándonos en los estados que conciernen al uso de monitores, podemos representarlos en el siguiente diagrama:

- **NEW** -> start() -> **RUNNABLE** -> exit() -> **TERMINATED**
- Desde **RUNNABLE**, un hilo puede pasar a **BLOCKED**.
- Desde **BLOCKED**, un hilo vuelve a **RUNNABLE**.

Espera en monitor: se accedió a una sección crítica controlada por un monitor y bloqueada por otro hilo. (Transición de RUNNABLE a BLOCKED).

Desbloqueo en monitor: el hilo que bloqueaba el monitor ha finalizado su sección crítica (bloque synchronized), dando el control al siguiente hilo bloqueado. (Transición de BLOCKED a RUNNABLE).

Como vemos, cuando un hilo intenta acceder a un método o bloque de código sincronizado y el objeto al que va a acceder está bloqueado por otro hilo, el hilo pasa a un estado bloqueado, pasando a formar parte de una cola de hilos bloqueados a la espera de que se desbloquee el monitor. Esta cola tendrá una estructura **FIFO (First In First Out)**, para garantizar la ejecución de todos los hilos.

3. Coordinación de Threads

3.1. El modelo productor-consumidor

El problema del modelo productor-consumidor es un problema clásico de la sincronización y supone un patrón común a diversos tipos de aplicaciones. El planteamiento inicial de este caso es el siguiente: disponemos de dos hilos, donde uno de ellos produce datos y el otro los consume. El problema se produce cuando el consumidor y el productor no realizan estas tareas de forma coordinada.

Como ejemplo veamos el siguiente programa, en el que dispondremos de un objeto compartido en el que una clase productora escribe valores que son leídos por una clase consumidora.

A. OBJETO COMPARTIDO

La siguiente clase ObjetoCompartido tendrá un atributo entero, que guardará un valor, y un booleano, que indica la disponibilidad de este. También implementa un método get, que devuelve el valor que contiene el objeto y establece a false la disponibilidad de este, puesto que ha sido consumido. El método set establece este valor e indica que está disponible.

```
class ObjetoCompartido {
   int valor;
   boolean disponible = false; // Inicialmente no tenemos valor
   int get() {
      if (this.disponible) {
         this.disponible = false;
         return this.valor;
      } else return -1;
   }

   void set(int vale) {
      this.disponible = true;
      this.valor = vale;
   }
}
```

B. CLASE PRODUCTOR

Por su parte, las clases Productor y Consumidor serán dos clases que implementen la interfaz Runnable y se inicializarán con un objeto compartido (de la clase ObjetoCompartido). Ambas clases en su método run() accederán a dicho objeto compartido, de modo que el Productor escriba (produzca) valores en él y el Consumidor lea (consuma) dichos valores. Concretamente, para ejemplificar esto el productor escribirá mediante un bucle cinco valores, con una pausa entre escrituras de 500 ms, mientras que el consumidor leerá estos cinco valores en un bucle, con pausas de 100 ms.

```
class Productor implements Runnable {
   // Referencia a un objeto compartido
   ObjetoCompartido compartido;

   Productor(ObjetoCompartido compartido) {
      this.compartido = compartido;
   }

   @Override
   public void run() {
      for (int y = 0; y < 5; y++) {
         System.out.println("El productor produce: " + y);
         this.compartido.set(y);
         try {
            Thread.currentThread().sleep(500);
         } catch (InterruptedException e) {
         }
      }
   }
}
```

C. CLASE CONSUMIDOR

```
class Consumidor implements Runnable {
   // Referencia a un objeto compartido
   private ObjetoCompartido compartido;

   Consumidor(ObjetoCompartido compartido) {
      this.compartido = compartido;
   }

   @Override
   public void run() {
      for (int y = 0; y < 5; y++) {
         System.out.println("El consumidor consume: " + this.compartido.get());
         try {
            Thread.currentThread().sleep(100);
         } catch (InterruptedException e) {
         }
      }
   }
}
```

Finalmente, disponemos de una clase principal, ModeloProductorConsumidor que creará un hilo a partir de cada una de estas clases y los lanzará:

```
public class ModeloProductorConsumidor {
   public static void main(String[] args) {
      ObjetoCompartido compartido = new ObjetoCompartido();
      Thread p = new Thread(new Productor(compartido));
      Thread c = new Thread(new Consumidor(compartido));
      p.start();
      c.start();
   }
}
```

Resultado del programa	Resultado esperado
El productor produce: 0 El consumidor consume: -1 El consumidor consume: 0 El consumidor consume: -1 El consumidor consume: -1 El consumidor consume: -1 El productor produce: 1 El productor produce: 2 El productor produce: 3	El productor produce: 0 El consumidor consume: 0 El productor produce: 1 El consumidor consume: 1 ... **Nota:** el comportamiento deseado consistiría en que el hilo consumidor fuese consumiendo los números a medida que el hilo productor los fuese generando.

El productor produce: 4	

En el caso contrario, en el que el productor produjese a mayor velocidad que el consumidor, y dado que en nuestro objeto compartido solamente cabe un objeto, habría valores que se perderían, puesto que se generaría el siguiente valor antes de que fuese leído.

La solución natural a este problema consistirá en que el productor espere a que el consumidor consuma los datos antes de producir nuevos datos. Del mismo modo, el consumidor deberá esperar a que los datos se hayan producido para poder consumirlos.

Por otra parte, cabe decir que este problema admite múltiples variantes y modificaciones en cuanto al objeto compartido, así como respecto al número de productores y consumidores. En este caso contiene únicamente un entero para almacenar datos y un booleano que controla si el dato es válido o no. En otros planteamientos podemos encontrar estructuras de datos más complejas, como listas, pilas o colas para almacenar la información del objeto compartido, así como la existencia de múltiples objetos productores y consumidores.

3.2. Los métodos wait(), notify() y notifyAll()

Para permitir la coordinación entre hilos en el acceso a objetos compartidos disponemos de los métodos wait(), notify() y notifyAll(). Estos métodos solo pueden utilizarse dentro de un bloque de código synchronized.

Método	Descripción
void wait()	Suspende al hilo que lo invocó y libera el monitor hasta que otro hilo invoque al método notify() o notifyAll().
void notify()	Cambia el estado de uno de los hilos suspendidos a ejecución. Si hay varios hilos, escoge a uno de ellos. Se dice que *despierta* a un hilo que estaba suspendido.
void notifyAll()	Despierta a todos los hilos suspendidos. Si estos vuelven a acceder al objeto compartido, deberemos hacerlo también de forma coordinada.

Veamos cómo aplicar estos métodos para coordinar a los productores y consumidores.

Cuando el productor genera un resultado (hace un set), si el objeto compartido todavía contiene un valor que no se ha consumido (por lo tanto, disponible==true), deberá esperar a que alguien lo consuma, por lo tanto, deberá quedar suspendido, mediante una invocación a wait(), hasta que alguien lea el valor y lo despierte.

```
while (disponible == true) {
   try {
```

```
    wait();
  } catch (InterruptedException e) {...}}
```

4. La librería java.util.concurrent

La librería java.util.concurrent ofrece un conjunto muy amplio de clases e interfaces orientadas a gestionar la programación concurrente de una forma más sencilla y óptima.

Entre todas ellas se encuentran la API ExecutorService y la interfaz Callable. La primera se encarga de separar las tareas de creación de threads, su ejecución y su administración, encapsulando la funcionalidad y mejorando el rendimiento, mientras que la interfaz Callable nos resuelve el problema de los Threads con los valores de retorno.

Por otra parte, la librería también presenta la interfaz BlockingQueue y varias clases que la implementan con diferentes estructuras de datos, con la característica común de que los accesos pueden llegar a ser bloqueantes si la lista está vacía o ha llegado al tope de su capacidad. Esto nos será de gran utilidad para abordar diferentes problemas de sincronización sin la necesidad de utilizar monitores en el acceso a objetos compartidos.

4.1. La interfaz Callable

La interfaz Callable es una versión mejorada de la interfaz Runnable, introducida en Java 1.5, que permite retornar un valor al finalizar su ejecución.

Cuando utilizamos la interfaz Runnable, disponemos de un método run() que no acepta parámetros ni devuelve ningún valor. Esto no es un problema, siempre y cuando no necesitemos proporcionar ni obtener datos del Thread. En cambio, cuando hemos necesitado proporcionar valores y obtener resultados de la ejecución de un hilo, hemos optado por la comunicación mediante objetos compartidos.

Por su parte, la interfaz genérica Callable proporciona el método call(), que devuelve un valor de tipo genérico. La sintaxis general de una clase que implemente esta interfaz será la siguiente:

```
public class claseCallable implements Callable<TipoGenerico> {
  // Constructor
  [public claseCallable(args){...}]

  // Método call
  public TipoGenerico call() throws InvalidParamaterException {
    return valor_retorno;
  }
}
```

Como principal diferencia respecto a Runnable, podemos ver que en la definición de la clase debemos indicar ya un tipo de datos genérico, que será el mismo que el del valor de retorno del método call, el cual, como vemos, utiliza ahora un return para devolver dicho valor.

FUTURETASK
Con la anterior definición podríamos utilizar el método call directamente:

```
claseCallable miCallable=new claseCallable(args);
TipoGenerico valor=miCallable.call();
```

Pero con ello no estaríamos ejecutando el método de forma asíncrona. Una primera aproximación a ello sería utilizar directamente la clase FutureTask, introducida en Java 5, y que se puede utilizar para realizar tareas asíncronas, ya que implementa la interfaz Runnable y, por tanto, puede lanzarse como un hilo.

La forma de hacer esto sería mediante la creación de un objeto de tipo FutureTask, de forma genérica, a partir del objeto creado de la clase Callable.

```
claseCallable miCallable=new claseCallable(args);
FutureTask<TipoGenerico> miFutureTask=new
FutureTask<TipoGenerico>(miCallable);
```

Este tipo genérico será el mismo que hayamos definido para nuestra clase Callable. Dado que FutureTask implementa Runnable, podemos crear ahora un Thread a partir de esta clase y lanzarlo:

```
Thread miThread=new Thread(miFutureTask);
miThread.start();
Thread miThread=new Thread(miFutureTask);
miThread.start();
```

Con esto el programa principal espera a que finalicen los hilos que ha generado. Una vez haya finalizado FutureTask, vamos a poder acceder al valor devuelto por el método call mediante el método get:

```
TipoGenerico resultado=miFutureTask.get();
```

Aunque, tal y como hemos visto, es posible utilizar la interfaz Callable de este modo, lo más habitual es utilizarla junto a la API ExecutorService, como veremos a continuación.

4.2. La API ExecutorService

ExecutorService es una API de Java que nos permite simplificar la ejecución de tareas asíncronas. Para ello nos ofrece un conjunto de hilos preparados para asignarles tareas.

La forma más sencilla para crear un ExecutorService es utilizando la clase Executors, la cual proporciona varios métodos de factoría y utilidades para la creación de múltiples API de ejecución asíncrona.

Por ejemplo, el método de factoría newFixedThreadPool crea un servicio de ejecución asíncrona con un conjunto de Threads de longitud fija. Para crear un servicio de este tipo que ponga a nuestra disposición hasta 10 hilos de ejecución haríamos:

```
ExecutorService servicio = Executors.newFixedThreadPool(10);
```

Aparte de este método, disponemos de muchos otros, como newCachedThreadPool(), que genera los hilos a medida que se van necesitando. Podemos consultar la lista de métodos en la documentación oficial de la clase Executors.

La forma de trabajar con esta API es relativamente simple, ya que solamente requiere de instancias de objetos de tipo Runnable o Callable y ella se encarga del resto de las tareas.

Así pues, una vez disponemos del servicio ExecutorService instanciado, tenemos a nuestra disposición diferentes métodos para asignarle tareas. Pero antes de abordar estos métodos vamos a ver la interfaz Future, para entenderlos mejor.

A. FUTURE

La interfaz Future, definida en el paquete java.util.concurrent, representa el resultado de una operación asíncrona. El valor de retorno, del tipo genérico indicado, no lo obtendremos de forma inmediata, sino que se obtendrá en el momento en que finalice la ejecución de la tarea asíncrona. En este momento, el objeto Future tendrá disponible dicho valor de retorno.

La interfaz Future proporcionará pues los mecanismos para saber si ya se dispone del resultado, para esperar a tener resultados y para consultar dichos resultados, así como para cancelar la función asíncrona si todavía no ha terminado.

Veamos en la siguiente tabla los métodos que nos proporciona esta interfaz:

Método	Descripción
boolean isDone()	Devuelve cierto si la tarea ha terminado.
TipoGenérico get()	Retorna el valor devuelto por la tarea, siempre que este esté disponible. En caso de no estar disponible, espera a que lo esté.
TipoGenérico get(long t, TimeUnit u)	Retorna el valor devuelto por la tarea, siempre que este esté disponible. En caso de no estar disponible, espera como mucho el tiempo t indicado en unidades de tiempo u.
boolean cancel(boolean interrupcion)	Intenta cancelar una tarea que está en ejecución. El booleano que le proporcionamos indica si debe interrumpirse para cancelar la tarea.
boolean isCancelled()	Devuelve cierto si la tarea ha sido cancelada antes de completarse.

B. ASIGNACIÓN DE TAREAS AL EXECUTORSERVICE

Ahora que ya conocemos la interfaz Future, vamos a examinar diferentes métodos para asignar tareas en el ExecutorService.

Método	Descripción	Uso
void execute()	Método heredado de la interfaz Executor, que simplemente ejecuta la tarea indicada, pero sin la posibilidad de obtener su resultado o estado.	ExecutorService s=...; s.execute(tareaRunnable);
Future<T> submit()	Envía una tarea, bien de tipo Callable o Runnable al servicio, retornando un tipo Future genérico T.	ExecutorService s=...; Future<T> resultado = s.submit(tarea);
Future<T> invokeAny()	Asigna una colección de tareas al servicio y devuelve el resultado de cualquiera de ellas.	ExecutorService s=...; Future<T> resultado = s.invokeAny(ListaTareas);
List<Future<T>> invokeAll()	Asigna una colección de tareas al servicio, ejecutándolas todas, y devuelve una lista de Futures con los resultados.	ExecutorService s=...; List<Future<String>> resultados = s.invokeAll(ListaTareas);

C. FINALIZACIÓN DEL SERVICIO
Una vez que dejemos de utilizar el servicio y que hayamos obtenido todos los resultados, utilizaremos el método shutdown para finalizar el servicio:

servicio.shutdown();

4.3. Colas concurrentes: BlockingQueue

Antes de centrarnos en las colas concurrentes debemos recordar qué es una cola. Al igual que las listas y las pilas, se trata de una estructura de datos compleja. La principal característica de las colas es que tienen una estructura FIFO (First In - First Out). El concepto es similar al de hacer cola, en el supermercado o en el cajero del banco, donde el primero que llega es el primero en ser atendido, y el resto de clientes van incorporándose al final de la cola.