PROGRAMACIÓN MULTIHILO

DESARROLLO DE APLICACIONES MULTIPLATAFORMA
PROGRAMACIÓN SERVICIOS Y PROCESOS

GBT

CONTENIDO

OBJETIVOS

1. RECONOCER SITUACIONES EN LAS QUE PUEDAN APLICARSE HILOS DE EJECUCIÓN Y DESARROLLAR CLASES QUE PUEDAN EJECUTARSE EN DIFERENTES HILOS DE EJECUCIÓN.
2. CONOCER LOS MECANISMOS DE SINCRONIZACIÓN ENTRE HILOS.
3. DESARROLLAR APLICACIONES CON VARIOS HILOS DE EJECUCIÓN, APLICANDO MECANISMOS DE SINCRONIZACIÓN ENTRE ELLOS.
4. DESARROLLAR APLICACIONES MULTIHILO UTILIZANDO LIBRERÍAS ESPECÍFICAS.

1. THREADS

Los threads, o procesos ligeros, permiten la creación de programas que ejecuten varias tareas de forma simultánea. Esta técnica de programación multihilo hace posible la creación de aplicaciones de tipo multimedia o cliente-servidor, entre otras, que requieren que varios procesos se ejecuten de forma simultánea en un ordenador, como el procesamiento de la imagen o del sonido, así como la atención a peticiones de forma simultánea.

La principal diferencia entre procesos e hilos es que los hilos de un mismo proceso comparten los mismos recursos (memoria, ficheros abiertos...). Así pues, cuando se generan varios hilos en un mismo proceso, es posible que varios de ellos pretendan acceder a datos u objetos del proceso de forma simultánea. Esta situación puede llevarnos a circunstancias no deseadas, por lo que se debe tener especial cuidado en cómo acceden los diferentes hilos a estos datos compartidos.

En esta unidad vamos a aprender a crear y gestionar estos hilos, así como a establecer mecanismos para evitar situaciones inesperadas en el acceso a los recursos compartidos.

1.1. TREADS 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 TREADS

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:

1. 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`
	`}`
	`}`

2. 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);`
3. Lanzar el hilo, invocando al método **start()**:	`hilo.start();`

1.3.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();`

1.3.B-TREADS 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.

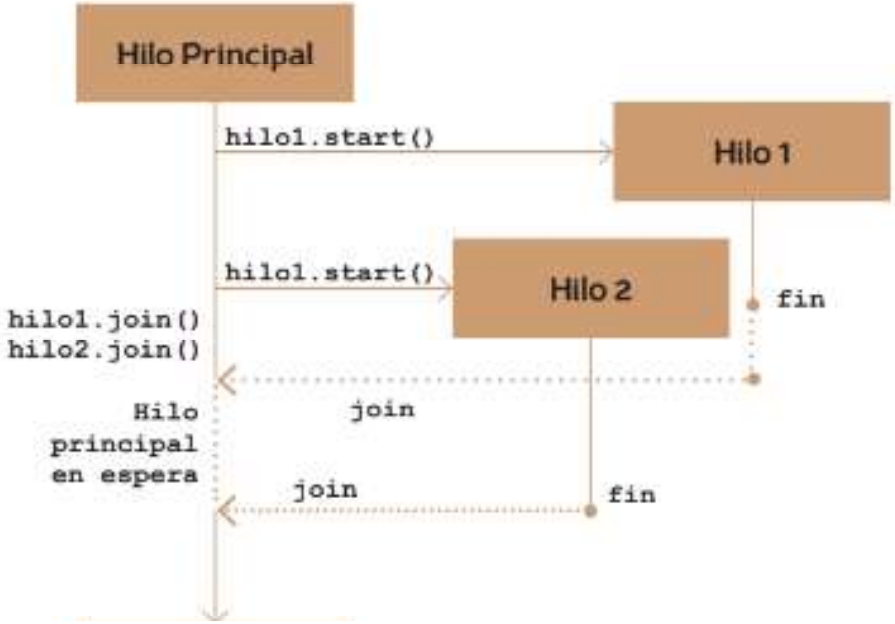
A partir de esta bifurcación, la ejecución de ambos hilos es independiente, de modo que cada uno puede realizar sus funciones en paralelo. Es posible que, en un momento dado, el hilo principal deba esperar a que finalicen los hilos que ha creado para seguir con su operativa, o bien para finalizar. En 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.

Como vemos, el método **`join()`** supone una primera aproximación a la sincronización de procesos, que trataremos más adelante.

IMPORTANTE

El método **`join()`** puede lanzar la excepción de tipo **`InterruptedException`** si el hilo al que se está esperando se interrumpe. En este caso, deberemos capturar y tratar esta excepción con su correspondiente bloque **`try..catch`**, o bien relanzarla con un **`throws`**.

1.4. JOIN() CON TIEMPO DE ESPERA

En ocasiones es posible que el hilo al que el hilo principal está esperando se haya quedado bloqueado. En este caso es posible que el hilo principal quede bloqueado también en esta espera. Para evitar esta situación, se pueden utilizar un par de versiones del método **join** con un temporizador.

`public final void join(long millis) throws InterruptedException`	Establece un tiempo máximo de espera en milisegundos para que finalice el **Thread**. Si se indica 0, espera a que finalice.
`public final void join(long millis, int nanos) throws InterruptedException`	Añade también una cantidad de nanosegundos a este tiempo máximo.

1.5 OTROS MÉTODOS DE LA CLASE THREAD

La clase **Thread** posee gran cantidad de propiedades y métodos que podemos consultar en su documentación oficial. No obstante, vamos a ver algunos métodos que nos pueden ser de utilidad para obtener información sobre los hilos:

Método	Descripción
`long getId()`	Devuelve el identificador del Thread, consistente en un entero largo positivo generado automáticamente en la creación del hilo. Este número es único e inalterable durante el ciclo de vida del Thread.
`void setName(String Nombre)`	Permite darle un nombre al Thread.
`String getName(String Nombre)`	Devuelve el nombre que le hemos asignado al Thread.
`Thread currentThread()`	Devuelve un objeto de la clase **Thread** que representa el hilo de ejecución actual.
`void setPriority(int prioridad)`	Permite establecer la prioridad del hilo, de forma indicativa, puesto que el sistema operativo no está obligado a respetarla. Las prioridades máxima y mínima vienen dadas por las constantes `MAX_PRIORITY` y `MIN_PRIORITY`.

`int getPriority()` `static boolean interrupted()`	Devuelve la prioridad del hilo.
`boolean isInterrupted()`	Devuelven si el hilo actual ha sido interrumpido. El método `interrupted()` es un método estático de la clase `Thread`, mientras que `isInterrupted()` es un método de instancia. Además, `interrupted()` restablece el estado, mientras que `isInterrupted()` no lo hace.
`boolean isAlive()`	Devuelve cierto si el hilo está vivo.
`public Thread.State getState()`	Devuelve el estado del hilo, que puede ser `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED`. Trataremos estos estados en el siguiente apartado.
`String toString()`	Devuelve una cadena con una representación del **Thread**, incluyendo el nombre, la prioridad y el grupo en que este se incluye.

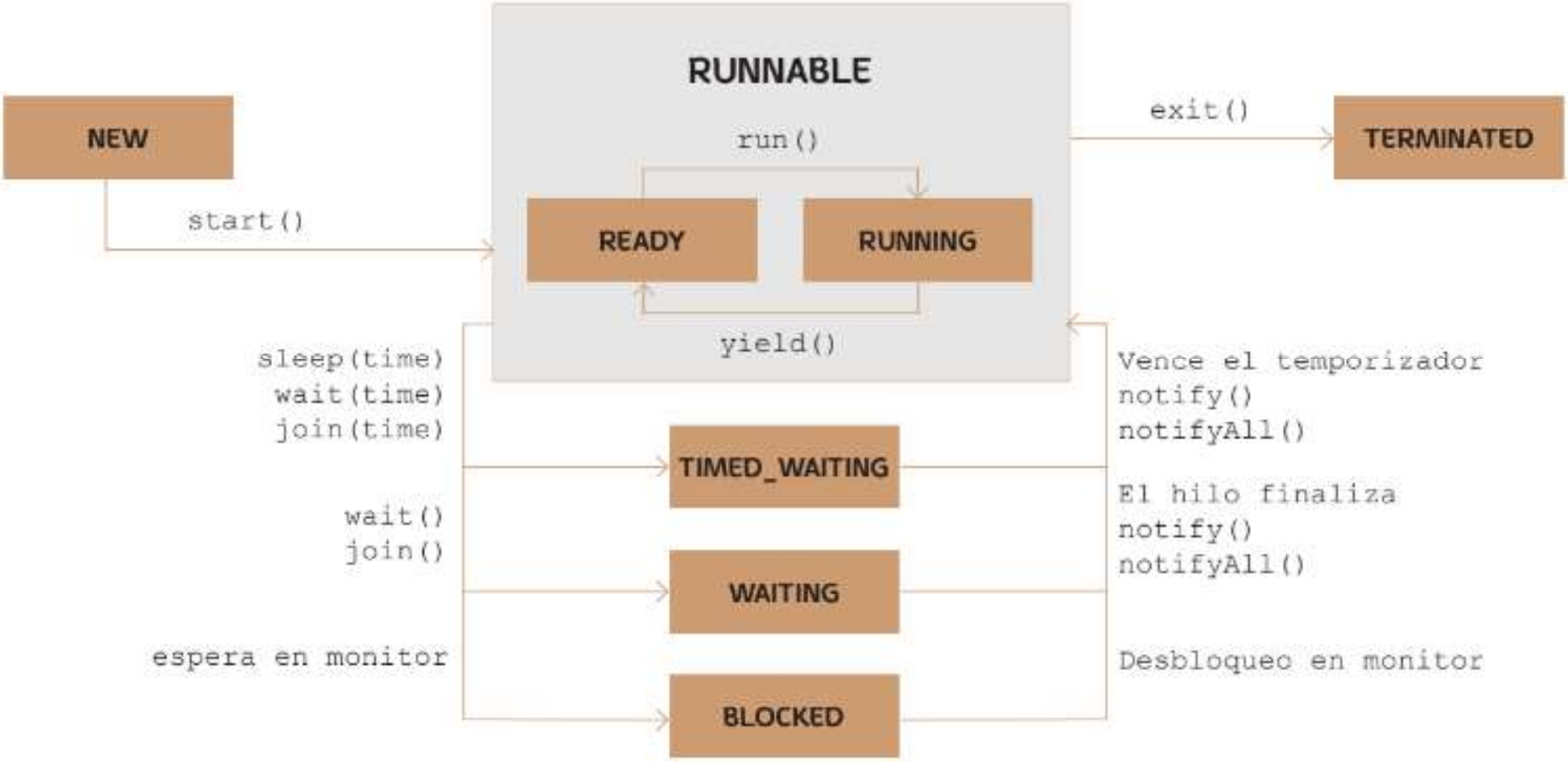
1.6 CICLO DE VIDA DE UN THREAD

Desde que se crea hasta que finaliza, un Thread pasa por diferentes estados. Podríamos definir pues el ciclo de vida de un Thread como el conjunto de estados por los que un Thread pasa desde su creación hasta su destrucción.

La clase **`Thread`** posee una propiedad estática de tipo *enumerado* que define los posibles estados de esta. En un momento dado, estos estados pueden ser:

Estado	Descripción
`NEW`	El hilo se ha creado, pero todavia no ha comenzado su ejecución.
`RUNNABLE`	El hilo se encuentra en ejecución o listo para su ejecución y a la espera de que se le asignen recursos.
`BLOCKED`	El hilo se encuentra bloqueado en un monitor.
`WAITING`	El hilo se encuentra esperando indefinidamente una acción de otro proceso.
`TIMED_WAITING`	El hilo se encuentra esperando un tiempo concreto una acción de otro proceso.
`TERMINATED`	El hilo ha finalizado su ejecución.

Gráficamente podemos ver los diferentes estados en el siguiente diagrama, así como las diversas transiciones entre ellos:



Veamos algunos métodos más de la clase **`Thread`** que nos permiten realizar cambios en el estado de un hilo. En los apartados siguientes profundizaremos en el resto.

Método	Descripción
`void Thread.sleep(long ms)`	Permite dejar el proceso en suspensión durante un tiempo concreto.
`void join()`	Espera la finalización de un hilo de ejecución, volviendo el control al hilo principal que lanzó el hilo secundario, con la posibilidad de escoger el tiempo de espera en milisegundos, como ya hemos visto.
`void yield()`	Vuelve a un proceso en el estado de **listo para ejecutar** (**`READY`** en el esquema), de forma que permite que pasen a ejecutarse otros hilos de la misma prioridad, aunque su estado sigue siendo **`RUNNABLE`**.
`void interrupt()`	Permite enviar una interrupción a un hilo, activando en este el **`flag interrupted`**, permitiéndole así gestionar de forma limpia su finalización. Estados como **`BLOCKED`**, **`WAITING`** o **`TIMED_WAITING`** gestionan esta interrupción de forma automática. El resto deberán gestionarla de forma adecuada mediante cláusulas **`try..cath`**.

Caso Práctico 1

Descarga el código del aula virtual CasoPractico 1

En él se define la clase MiRunnable, a partir de la cual se pueden crear hilos. Además, en su método run() muestra información de los hilos en sí. La clase CasoPractico1, que contiene el método main(), por su parte, crea tres objetos de la clase MiRunnable y los lanza en paralelo.

Compila y ejecuta el ejemplo y analiza los resultados, prestando especial atención a aspectos como:

- El orden en que se muestran los resultados.
- ¿Hay alguna diferencia entre la referencia this y hiloActual? En qué se diferencia a nivel de ejecución this.nombre y hiloActual.getName()?

Después de compilar el ejemplo con javac CasoPractico1.java, lo lanzamos un par de veces y obtenemos las siguientes salidas:

```
$ java CasoPractico1
Hola Mundo de los threads. Soy Joan:
        Mi id de thread es 13
        Mi nombre de thread es Thread-1
        Mi prioridad es 5
        Mi estado es RUNNABLE

Hola Mundo de los threads. Soy Jose:
        Mi id de thread es 12
        Mi nombre de thread es Thread-0
        Mi prioridad es 5
        Mi estado es RUNNABLE

Hola Mundo de los threads. Soy Vicente:
        Mi id de thread es 14
        Mi nombre de thread es Thread-2
        Mi prioridad es 5
        Mi estado es RUNNABLE

$ java CasoPractico1
Hola Mundo de los threads. Soy Vicente:
        Mi id de thread es 14
        Mi nombre de thread es Thread-2
        Mi prioridad es 5
        Mi estado es RUNNABLE

Hola Mundo de los threads. Soy Joan:
        Mi id de thread es 13
        Mi nombre de thread es Thread-1
        Mi prioridad es 5
        Mi estado es RUNNABLE

Hola Mundo de los threads. Soy Jose:
        Mi id de thread es 12
        Mi nombre de thread es Thread-0
        Mi prioridad es 5
        Mi estado es RUNNABLE
```

Como vemos, y respondiendo a la primera cuestión que se plantea, el resultado de la ejecución no es exactamente el mismo, ya que, en cada ejecución, los hilos se ejecutan en un orden. Esto es normal e inherente a la propia programación multihilo. Cuando lanzamos el método start() sobre cada hilo, estos pasan al estado de Preparados para ejecutar, dentro del estado RUNNABLE, y el sistema es el que decide qué hilos pasan de Preparados a RUNNING, independientemente de en qué orden se lanzó el start().

Por otra parte, respecto a la diferencia entre this y CurrentThread, cabe remarcar que this hace referencia al propio objeto de tipo Runnable, mientras que currentThread hace referencia al hilo que hemos creado a partir de ese Runnable. Por este motivo, no tiene nada que ver el atributo nombre de la clase MiRunnable, que es el que asignamos en la creación del objeto, con el nombre del Thread, que obtenemos con hiloActual.getName(), y que asigna automáticamente el sistema.

Tarea 0 Aula Virtual

Programa en Java que muestre los distintos estados de un hilo, el programa ilustra el ciclo de vida de un hilo (Thread) y que permite observar los diferentes estados por los que pasa, desde su creación hasta su terminación.
Se utiliza el método join() para sincronizar la ejecución del hilo principal con el hilo secundario.

Redactar una breve conclusión que resuma el comportamiento observado y la importancia de los métodos join() y getState() en la gestión de hilos.

Tarea I Aula Virtual

2. MÉTODOS SINCRONIZADOS: SYNCHRONIZED

La programación de aplicaciones multihilo implica la generación de diferentes hilos de ejecución para abordar la resolución de un problema de forma concurrente. En este caso, puede que surja la necesidad de mantener objetos accesibles por todos los hilos, que además sirvan como mecanismo de comunicación entre ellos.

Cuando un objeto o variable es compartido por varios hilos de ejecución, debemos ser especialmente cautos en su tratamiento y acceder a él de manera coordinada, de modo que las operaciones que un hilo realiza sobre uno de estos objetos no se vean alteradas por otras. Pensemos por ejemplo en la escritura de un fichero, que suele ser un proceso más largo. Si dos hilos, por ejemplo, desean escribir en el mismo fichero, deberán hacerlo de manera que las escrituras de uno no interfieran en las del otro. Lo más sencillo es que un hilo espere a que el otro termine de escribir para poder escribir él.

En este apartado vamos a introducir los conceptos necesarios para contextualizar estas situaciones y darles una solución desde el punto de vista de la programación, mediante los mecanismos que Java nos proporciona para ello.

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.

2.2.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.

2.2.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.

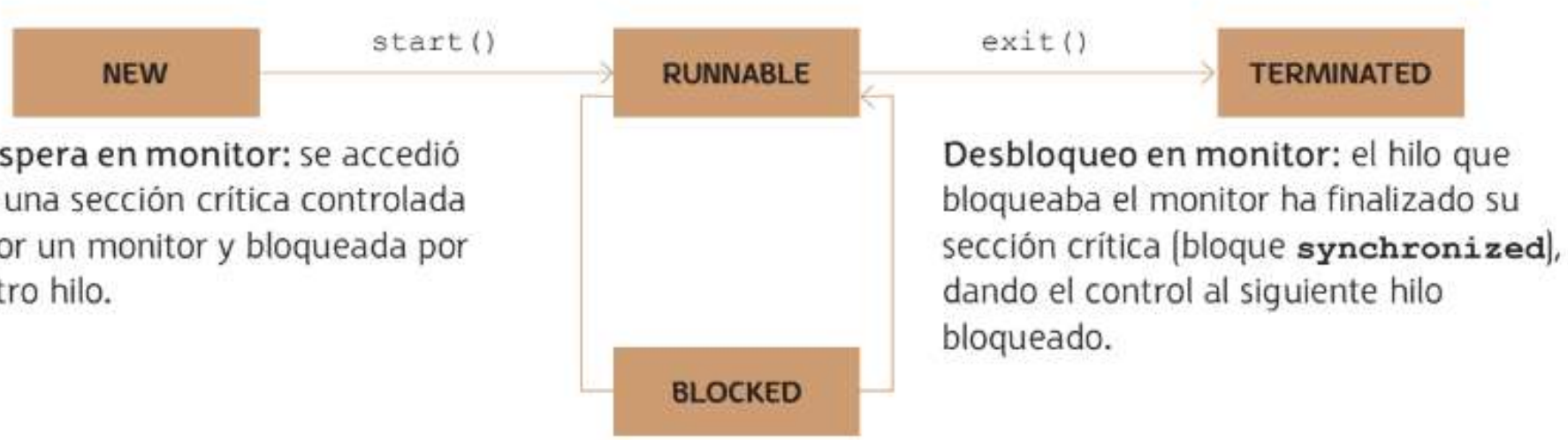
```
class MiClase implements Runnable {

    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**.

2.2.C SYNCRONIZED 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:



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.

Ejercicio

Sumatorio con threads.- descarga el código

En la unidad anterior vimos cómo realizar de una forma bastante rudimentaria el cálculo de la suma de los números comprendidos en un rango definido por dos índices. Para ello, dividimos el proceso en varios subprocesos y los lanzamos concurrentemente, comunicando los diferentes procesos mediante la entrada y la salida.

Vamos a realizar este mismo ejercicio, pero utilizando hilos de ejecución

tarea II

Argumento Mayor

Se pide realizar una aplicación que calcule el máximo de una lista proporcionada por la línea de comandos. El programa dividirá esta lista en varias sublistas y utilizará diferentes hilos para calcular el máximo de cada sublista. El máximo de todos ellos se mantendrá en un objeto compartido, que mantendrá en cada momento el valor máximo calculado hasta ese momento.

- Deberás crear tres clases diferentes, una que será el objeto compartido, otra que calcule el valor máximo de una sublista y actualice el objeto compartido, y la principal, que se encargará de dividir la lista de valores en sublistas y generar los hilos.

- Los siguientes fragmentos pueden resultarte útiles:

Creación de un ArrayList de enteros a partir de los argumentos:

```
ArrayList<Integer> numeros=new ArrayList<>();
      for(int i=0; i<args.length; i++){
          numeros.add(Integer.parseInt(args[i]));
      }
```

Creación de un ArrayList como sublista de otro (numeros):

```
ArrayList<Integer> subArrayList = new
 ArrayList<Integer>(numeros.subList(ini, fin));
```

Vamos a implementar el mismo programa pero ahora el recurso compartido será un AtomicInteger en lugar de una variable protegida por sincronización, podemos usar la clase AtomicInteger de Java, que proporciona operaciones atómicas seguras para ser usadas en programación concurrente, sin necesidad de usar bloques sincronizados explícitamente.

Descarga el código del aula virtual.

En la programación concurrente, **las condiciones de carrera** son situaciones en las que dos o más hilos de ejecución acceden y manipulan un recurso compartido de manera simultánea, **lo que puede llevar a resultados inesperados e inconsistentes**.

Este problema ocurre cuando el comportamiento del programa depende del orden en que se ejecutan esos hilos, lo que significa que, si se alteran las interacciones entre ellos, el resultado final puede variar. Por ejemplo, si un hilo está actualizando un valor mientras otro hilo intenta leerlo al mismo tiempo, puede suceder que el segundo hilo obtenga un valor incorrecto o desactualizado.

Para evitar estas condiciones de carrera, es crucial implementar mecanismos de sincronización y coordinación, que aseguran que solo un hilo pueda acceder a un recurso compartido a la vez, garantizando así la coherencia y la integridad de los datos en aplicaciones concurrentes.

Otra situación es el **interbloqueo**, también conocido como **deadlock**, es una situación en la programación concurrente en la que dos o más hilos de ejecución se bloquean entre sí, impidiendo que cualquiera de ellos continúe su ejecución.
Esto ocurre cuando cada hilo espera por un recurso que está siendo sostenido por otro hilo, formando un ciclo de espera.

S*i el Hilo A tiene el recurso 1 y está esperando por el recurso 2, mientras que el Hilo B tiene el recurso 2 y está esperando por el recurso 1, ninguno de los hilos puede avanzar.*

El interbloqueo puede llevar a una paralización del sistema, por lo que es crucial implementar estrategias de prevención, detección y recuperación para manejarlo adecuadamente en aplicaciones concurrentes.

3. COORDINACIÓN DE THREADS

La sincronización es esencial al trabajar con múltiples procesos, ya que ayuda a evitar inconsistencias en los resultados. Para que los hilos de ejecución colaboren de manera efectiva y no se pierda información, es fundamental coordinar su trabajo.

En programación concurrente, hay varios problemas clásicos de sincronización que reflejan situaciones de la vida real. Las soluciones a estos problemas pueden aplicarse a otros casos similares. Algunos de estos problemas incluyen:

- Los lectores y escritores
- Los filósofos comensales
- El barbero dormilón
- Los productores y consumidores

Cada uno de estos problemas trata sobre cómo manejar el acceso a recursos compartidos entre diferentes hilos y ofrece soluciones que garantizan una correcta sincronización. Estas soluciones se pueden implementar usando mecanismos como semáforos o monitores.

Ahora, nos enfocaremos en el problema de los productores y consumidores, donde exploraremos cómo los hilos de ejecución producen y consumen información de manera coordinada y sincronizada, utilizando monitores.

En el ámbito de la programación concurrente, donde múltiples hilos de ejecución operan simultáneamente, es fundamental gestionar el acceso a los recursos compartidos para evitar problemas como condiciones de carrera.
Una de las herramientas más efectivas para lograr esto son los monitores, que permiten la coordinación entre hilos. Repasemos qué son los monitores y veamos cómo se utilizan para facilitar la coordinación en entornos concurrentes.
Un monitor es un mecanismo de sincronización que permite el acceso controlado a un recurso compartido por múltiples hilos. En Java, cada objeto actúa como un monitor, y se pueden utilizar métodos de sincronización para gestionar el acceso a sus miembros. Los monitores son especialmente útiles porque:

- Aseguran la exclusión mutua: Solo un hilo puede acceder a la sección crítica del código (la parte que manipula el recurso compartido) a la vez.
- Permiten la espera activa: Los hilos pueden esperar hasta que una condición específica se cumpla antes de continuar su ejecución.

3.1. MÉTODOS DE COORDINACIÓN: WAIT(), NOTIFY(), Y NOTIFYALL()

Los métodos wait(), notify(), y notifyAll() son fundamentales para la coordinación entre hilos que utilizan monitores.

1. **wait()**
   - Descripción: El método wait() hace que el hilo que lo invoca libere el bloqueo del monitor y entre en un estado de espera.
   - Uso: Se utiliza cuando un hilo necesita esperar por una condición que no se cumple.
2. **notify()**
   - Descripción: El método notify() despierta a un único hilo que está esperando en el monitor del objeto.
   - Uso: Se utiliza para notificar a un hilo que debe reevaluar su condición de espera.

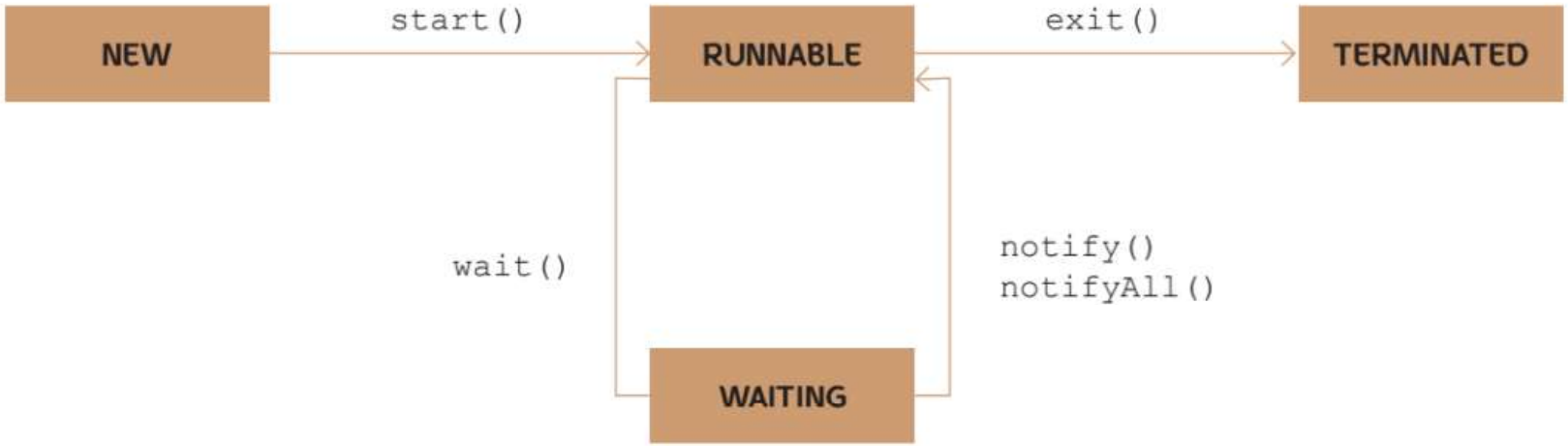
3. **notifyAll()**
   - Descripción: El método notifyAll() despierta a todos los hilos que están esperando en el monitor del objeto.
   - Uso: Se utiliza cuando múltiples hilos necesitan ser notificados para reevaluar su condición.

Los monitores son una herramienta poderosa en programación concurrente que facilitan la coordinación entre hilos. Al utilizar los métodos wait(), notify(), y notifyAll(), los hilos pueden comunicarse de manera efectiva, asegurando que los recursos compartidos sean utilizados de forma segura y eficiente. Esto ayuda a evitar condiciones de carrera y garantiza que el sistema funcione de manera coherente, incluso en entornos de ejecución concurrente.

- wait(): Hace que un hilo libere el monitor y espere.
- notify(): Despierta a un solo hilo que está esperando.
- notifyAll(): Despierta a todos los hilos que están esperando.

WAIT(), NOTIFY() Y NOTIFYALL() Y ESTADOS

Todo el proceso anterior, en el modelo de estados de un hilo, se vería del siguiente modo:



Como vemos, un hilo que se encontraba en ejecución pasa a estado **`WAITING`** cuando invoca al método **`wait()`**, ya sea un productor o un consumidor. Cuando el hilo o hilos que estaban en espera reciben la excepción **`InterruptedException`**, bien con un **`notify()`** o con un **`notifyAll()`**, estos vuelven al estado de ejecución, donde vuelven a comprobar la disponibilidad de los datos.

Problema del productor-consumidor

El problema del productor-consumidor es un clásico en programación concurrente que ilustra la necesidad de sincronización entre hilos que producen y consumen recursos compartidos.

En este problema, hay dos tipos de hilos: productores y consumidores.

En este escenario, un productor genera datos y los coloca en un recurso compartido, mientras que un consumidor extrae estos datos.

Sin una adecuada sincronización, se pueden presentar problemas como la condición de carrera y el interbloqueo.

Descarga el código Productor-Consumidor

El programa comienza creando una instancia del recurso compartido y luego inicia dos hilos: uno para el productor y otro para el consumidor.

El productor genera datos (del 0 al 4) y los coloca en el recurso, mientras que el consumidor extrae estos datos. **La sincronización se asegura** mediante el uso de wait() y notifyAll(), evitando condiciones de carrera, **que el productor no produzca más datos si ya hay uno en el recurso y que el consumidor no intente consumir sin que haya datos disponibles**.

A través del uso de monitores y métodos de sincronización, como wait() y notify(), es posible resolver este problema de manera efectiva, asegurando una correcta interacción entre productores y consumidores.

Excepción InterruptedException

¿Qué es InterruptedException?

- Descripción: La excepción InterruptedException es lanzada cuando un hilo en espera es interrumpido de manera inesperada. Esto puede suceder, por ejemplo, cuando un hilo es interrumpido mientras está en espera debido a que otro hilo ha invocado el método interrupt() sobre él.
- Importancia: Es importante manejar esta excepción adecuadamente para garantizar que el hilo pueda finalizar su trabajo de manera segura y liberar cualquier recurso que esté utilizando.

Tarea III

Productor-consumidor con una pila de enteros

3.2. SEMÁFOROS

Los semáforos son una de las herramientas fundamentales en la programación concurrente. Se utilizan para controlar el acceso a recursos compartidos en entornos donde múltiples hilos pueden intentar acceder al mismo recurso al mismo tiempo. El objetivo de los semáforos es evitar condiciones de carrera, interbloqueos y garantizar que los recursos se utilicen de manera eficiente

En Java, los semáforos se pueden implementar utilizando la clase Semaphore que se encuentra en el paquete **java.util.concurrent**.

Un semáforo es una variable de control que se utiliza para gestionar el acceso a un recurso compartido por múltiples hilos. Se compone de **un contador** y **dos operaciones principales.**

- **acquire()**: Se utiliza para solicitar acceso al recurso. Si el contador del semáforo es mayor que cero, se decrementa y se permite el acceso. Si es cero, el hilo se bloquea hasta que el semáforo se libere.
- **release()**: Se utiliza para liberar el recurso, incrementando el contador del semáforo y permitiendo que otros hilos adquieran acceso.

Tipos de Semáforos

- **Semáforo Binario**: Puede tener solo dos estados (0 y 1). Es útil para proteger recursos de manera exclusiva.
- **Semáforo Contador**: Puede tener un valor entero positivo y permite que varios hilos accedan a un recurso limitado.

Crear un semáforo con un contador de 3 (permite hasta 3 hilos acceder al mismo tiempo)
private static Semaphore semaforo = new Semaphore(3);

Descarga código Semaforo1

Creación del Semáforo: Se crea un semáforo con un solo permiso (new Semaphore(1)), lo que significa que solo un hilo puede acceder al recurso a la vez.

Métodos:

- acquire(): Un hilo intenta adquirir el semáforo. Si otro hilo ya tiene acceso, este hilo se bloqueará hasta que el semáforo se libere.
- release(): El semáforo se libera al finalizar el trabajo, permitiendo que otros hilos puedan acceder al recurso.

Se crean dos hilos que intentan acceder al mismo recurso compartido, demostrando cómo los semáforos gestionan el acceso concurrente.

Problema de los filósofos comensales

Cinco filósofos están sentados alrededor de una mesa circular, y cada filósofo puede comer o pensar, pero no al mismo tiempo.
Para comer, cada filósofo necesita dos tenedores (uno a la izquierda y otro a la derecha). Hay cinco tenedores (uno entre cada par de filósofos).

- Problemas: Si cada filósofo toma un tenedor, todos pueden quedarse esperando por el segundo tenedor, generando un bloqueo mutuo (deadlock). También puede haber inanición si un filósofo nunca tiene acceso a ambos tenedores.

El problema de los filósofos comensales es un clásico en la teoría de la concurrencia y se utiliza para ilustrar los problemas de sincronización y las condiciones de carrera que pueden surgir en un entorno de múltiples hilos. Este problema involucra a varios filósofos que alternan entre pensar y comer, y se enfrentan a un problema de recursos compartidos (los tenedores).

Descarga el código del Aula Virtual

El código proporcionado para el problema de los filósofos, cada filósofo está diseñado para pensar y comer indefinidamente, lo que significa que el programa nunca termina por sí mismo. Esto es una característica del problema de los filósofos comensales: los filósofos se quedan atrapados en un bucle infinito alternando entre pensar y comer.

¿Cómo Terminar el Programa? Aquí hay algunas sugerencias:

- Contador de Comidas: Agregar un contador que limite el número de comidas por filósofo.
- Temporizador: Hacer que el programa termine después de un tiempo determinado.
- Interrupción Manual: Terminar el programa manualmente (Ctrl+C en la consola).
- ...

Tarea IV. agrega un contador de comidas a cada fiósofo

```
Filósofo 4 está comiendo.
Filósofo 0 ha terminado de comer.
Filósofo 0 está pensando.
Filósofo 4 ha terminado de comer.
Filósofo 4 está pensando.
Filósofo 3 está comiendo.
Filósofo 1 ha terminado de comer.
Filósofo 1 está pensando.
Filósofo 0 está comiendo.
Filósofo 3 ha terminado de comer.
Filósofo 2 está comiendo.
Filósofo 3 está pensando.
Filósofo 0 ha terminado de comer.
Filósofo 4 está comiendo.
Filósofo 0 está pensando.
Filósofo 2 ha terminado de comer.
Filósofo 2 está pensando.
Filósofo 1 está comiendo.
Filósofo 4 ha terminado de comer.
Filósofo 3 está comiendo.
Filósofo 4 está pensando.
Filósofo 3 ha terminado de comer.
Filósofo 3 está pensando.
Filósofo 1 ha terminado de comer.
Filósofo 1 está pensando.
Filósofo 0 está comiendo.
Filósofo 2 está comiendo.
Filósofo 0 ha terminado de comer.
Filósofo 4 está comiendo.
Filósofo 0 ha comido 3 veces y ha terminado.
Filósofo 4 ha terminado de comer.
Filósofo 4 ha comido 3 veces y ha terminado.
Filósofo 2 ha terminado de comer.
Filósofo 1 está comiendo.
Filósofo 3 está comiendo.
Filósofo 2 está pensando.
Filósofo 1 ha terminado de comer.
Filósofo 1 ha comido 3 veces y ha terminado.
Filósofo 3 ha terminado de comer.
Filósofo 3 ha comido 3 veces y ha terminado.
Filósofo 2 está comiendo.
Filósofo 2 ha terminado de comer.
Filósofo 2 ha comido 3 veces y ha terminado.
Todos los filósofos han terminado de comer.
```

4.LA LIBRERÍA JAVA.UTIL.CONCURRENT

La librería java.util.concurrent ofrece un amplio conjunto de clases (como Semaphore) e interfaces (entre las que se encuentran la API ExecutorService y la interfaz Callable) orientadas a gestionar la programación concurrente de manera más sencilla y eficiente.

- **ExecutorService,**
  - Se encarga de separar las tareas de creación de threads, su ejecución y su administración, encapsulando la funcionalidad y mejorando el rendimiento.
  - Ofrece una capa de abstracción para el manejo de hilos, permitiendo que los desarrolladores trabajen con tareas concurrentes sin tener que lidiar directamente con la complejidad de crear y gestionar hilos manualmente.
  - En lugar de crear hilos con new Thread() y gestionarlos manualmente, la API de ExecutorService proporciona métodos para ejecutar tareas de manera eficiente, reutilizando un grupo de hilos (thread pool).
  - proporciona un conjunto de métodos estandarizados.
- **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.

ExecutorService como API

ExecutorService ess una interfaz dentro de la librería java.util.concurrent que forma parte de una API más amplia destinada a la ejecución de tareas concurrentes LA API DE CONCURRENCIA DE JAVA.

La razón por la que se le llama API es porque ExecutorService no actúa solo; está diseñado para ser utilizado junto con otras clases e interfaces (como Executors, Future, ThreadPoolExecutor, etc.) que, en conjunto, proporcionan un marco completo para manejar la concurrencia en Java.

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 la 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.

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

class Suma implements Callable<Long> {
   ...
      @Override
      public Long call() {
      ...
      return result;
```

Un tipo genérico en Java es una característica del lenguaje que permite definir clases, interfaces y métodos con un parámetro de tipo, lo que proporciona una forma de crear estructuras de datos y algoritmos que pueden operar sobre diferentes tipos de datos de manera segura y reutilizable.
Al utilizar tipos genéricos, se puede escribir código más flexible y evitar la necesidad de realizar conversiones de tipo explícitas, ya que los tipos se verifican en tiempo de compilación.
Esto mejora la legibilidad del código y reduce la posibilidad de errores en tiempo de ejecución.

```java
class GenericCallable<T> implements Callable<T> {
    private final T value;

    public GenericCallable(T value) {
        this.value = value;
    }

    @Override
    public T call() throws Exception {
        // Simulamos algún procesamiento que devuelve el valor proporcionado
        Thread.sleep(1000); // Simula una operación que toma tiempo
        return value;
    }
}
```

Es importante mencionar que, a la hora de usar tipos genéricos, deben ser de tipo envoltorio (como Integer, Double, Character, etc.) en lugar de tipos primitivos (como int, double, char, etc.), debido a que los tipos genéricos no pueden trabajar directamente con tipos primitivos.
Por ejemplo, al declarar una lista genérica, se debe utilizar List<Integer> en lugar de List<int>.

Como principal diferencia 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 valor de retorno del método call, el cual, como vemos, utiliza ahora un return para devolver dicho valor.
Para usar esto de forma aíncrona, usamos la clase FutureTask, introducida en java 5, que se puede utilizar para realizar tareas aíncronas ya que implementa la interfaz Runnable y, por tanto puede lanzarse como un hilo.
Creamos un objeto de tipo FutureTask, de forma genérica, a partir del objeto creado de la clase Callable.

```
claseCallable miCallable=new classCallable(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();
```

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();
```

Veamos un ejemplo. Descargamos el código sumatorio_de_un_rango_de_valores del aula virtual.

Tarea V.- Obtención del elemento mayor en una lista
Partiendo del código visto, se pide implementar la búsqueda del mayor entero en una lista de valores proporcionados como argumentos, dividiendo la lista en sublistas y lanzando un Thread para cada sublista.

- Al igual que hemos hecho en el caso anterior, suma por rango de valores, no vamos a necesitar ningún objeto compartido.
- Deberás utilizar simplemente una variable para almacenar el máximo de los valores que vayamos obteniendo a partir de los Future.

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 vemos a continuación.

4.2. LA INTERFAZ EXECUTORSERVICE

ExecutorService, nos permite simplificar la ejecución de las tareas asíncronas. Para ellos 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, permitiendo a los desarrolladores gestionar y ejecutar diversas tareas en paralelo de manera eficiente.
Por ejemplo. el método de factoría newFixedThreadPoll 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);
```

A parte 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 sencilla, ya que solamenente requiere de instancias de objetos de tipo Runnable o Callable y ella se encarga del resto de las tareas.

Así pues, una vez que disponemos del servicio ExecutorService instanciado, tenemos a nuestra disposición diferentes métodos para asignarle tareas.

https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/concurrent/Executors.html

4.2.A. LA INTERFAZ 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 la forma inmediata, sino que se obtendrá en el momento en el 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.

Métodos interfaz Future

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.

https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/concurrent/Future.html

4.2.B .ASIGNACIÓN DE TAREAS AL EXECUTORSERVICE

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);`

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html

4.2.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();
```

Descarga el código Código ExecutorServiceSuma del Aula Virtual

Tarea VI.-Argumento mayor usando la interfaz Callable y ExecutorService

4.3. COLAS CONCURRENTES: BLOCKINGQUEUE

Las colas es que tienen una estructura FIFO(Fisrt In- First Out). En java este comportamiento lo ofrece la interfaz Queue, que posee varias clases que la implementan, como ArrayQueue, ArrayDeque o AbstractQueue, entre otras.

La interfaz BlockingQueue ofrece un comportamiento similar, con la diferencia de que además este comportamiento ofrece mecanismos de acceso seguros en operaciones concurrentes [thread-safe], lo que nos será de gran utilidad a la hora de abordar problemas de sincronización, como el de los productores-consumidores
Mediante esta interfaz, los hilos productores deberán esperar a producir si la cola está llena, y los consumidores esperar a consumir mientras la cola esté vacía. Todo el comportamiento que generábamos mediante monitores en el objeto compartido nos lo gestionará ahora la cola.

Esta interfaz aporta dos métodos a las colas: put y take, que serían las equivalentes a add y remove, pero con la particularidad de que las operaciones se realizan de forma segura.

Veamos las diferentes implementaciones de esta interfaz en la siguiente tabla.

Clase	Descripción
`ArrayBlockingQueue`	Implementa una cola mediante un array, por lo que tendrá una longitud fija. Las operaciones de **`put`** y **`take`** asegurarán que no se sobreescriban las entradas.
`LinkedBlockingQueue`	Implementa una cola mediante una lista enlazada, donde cada elemento de la lista es un nodo nuevo, con lo que en principio podría crecer indefinidamente (hasta llegar a **`Integer.MAX_VALUE`**). Su constructor admite un entero para especificar el número de elementos máximo que puede almacenar. Las operaciones de **`put`** y **`take`** asegurarán que no se sobreescriban las entradas.
`DelayQueue`	Implementa una cola que contiene elementos que implementan la interfaz **`Delayed`**. Los objetos que implementan esta interfaz requieren de un retraso (**`delay`**) para operar sobre ellos. Estas colas son útiles, por ejemplo, cuando un consumidor únicamente puede consumir elementos después de un tiempo.

Clase	Descripción
`LinkedTrasnferQueue`	Incluye el método **`transfer`**, que permite a un productor esperar a que se consuma un ítem, de modo que los consumidores puedan manejar el flujo de mensajes producidos.
`SynchronousQueue`	Contiene como mucho un elemento, de modo que representa una forma sencilla de intercambiar datos entre dos hilos.
`ConcurrentLinkedQueue`	Se trata de una cola no bloqueante, apta para la programación de sistemas reactivos.

https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

Ejercicio:

Se pide realizar una adaptación al problema del productor-consumidor planteado anteriormente. pero utilizando una cola concurrente [BlockingQueue], de modo que el comportamiento que presente sea de tipo FIFO, es decir, los elementos se consumirán en el mismo orden que se producen.

Se nos pide implementar una cola de longitud fija [3 elementos], para ello vamos a utilizar la clase ArrayBlockingQueue, que utiliza un array para implementar la cola y nos ofrece las operaciones de put y take para almacenar (producir) y recuperar(consumir) elementos.

Esta cola será el objeto compartido entre productor y consumidor, de modo que se limiten a añadir y obtener elementos de la cola, sin necesidad de utilizar monitores en sus accesos.

Descarga el código del Aula Virtual. Productor-Consumidor-BlockingQueue.