Estados de los Hilos

Al igual que los Procesos, los Hilos también tienen un **Ciclo de Vida**. Los **Estados** en Java están recogidos dentro de la clase *java.lang.Thread* y son los siguientes:

- **Nuevo (NEW)**: El Hilo se ha creado, pero no se ha arrancado. En código, se correspondería a hacer algo similar a:
- **Ejecutable (RUNNABLE)**: El Hilo está arrancado y podría estar ejecutándose.
- **Bloqueado (BLOCKED)**: El Hilo ha sido Bloqueado por un monitor.
- **Esperando (WAITING)**: El Hilo está esperando a que otro Hilo realice una acción.
- **Esperando (TIME_WAITING)**: El Hilo espera a que otro Hilo realice una acción. durante un tiempo concreto.
- **Finalizado (TERMINATED):** El Hilo ha completado su ejecución.

Es posible consultar los estados de los **Hilos** mediante **getState ()**.

```
Raton raton = new Raton();
raton.start();

// Comprobamos su estado
if (raton.getState() != Thread.State.RUNNABLE) {
    System.out.printf("Hilo Principal: RUNNABLE%n");
} if (raton.getState() != Thread.State.WAITING) {
    System.out.printf("Hilo Principal: WAITING%n");
} if (raton.getState() != Thread.State.TERMINATED) {
    System.out.printf("Hilo Principal: TERMINATED%n");
}
```

El **Estado** de un **Hilo** puede cambiar por el Sistema Operativo o más normalmente porque el propio código del programa contiene *instrucciones* que los modifican. Un ejemplo es la operación **wait ()**, que detiene temporalmente la ejecución de un **Hilo**.

Programación multihilo: clases y librerías

Java dispone de una serie de clases destinadas a la Programación Multihilo. Se encuentran principalmente en dos paquetes: *java.lang* y *java.util.concurrent*. A continuación, se describen con varios ejemplos para ver sus utilidades.

Paquete java.lang

Este paquete contiene entre otras las clases **Thread** y **Runnable** que hemos visto anteriormente. Ambas son la base del manejo de los **Hilos**. Adicionalmente, también dispone de las clases **Timer** y **TimerTask**.

Clases	
Runnable	Interfaz que deben de implementar las clases que quieran ejecutarse como un Thread. Define el método run ().
Thread	Esta clase implementa Runnable y es un Hilo ella misma. Una clase puede heredar de Thread y sobrescribir run ().
Timer	Permite ejecutar tareas de forma diferida y repetitiva mediante el método schedule ().
TimerTask	Clase abstracta que, al ser heredada y sobrescrito su run (), permite crear tareas para ser ejecutadas por Timer.

Un ejemplo de Timer y TimerTask: un sistema de riego automático cada dos segundos.

```
public class SistemaDeRiego extends TimerTask {

    @Override
    public void run() {
        System.out.printf("Regando...%n");
    }

    public static void main(String[] args) {
        System.out.printf("Hilo Principal arrancado%n");

        // Retardo de 1 segundo, riega cada 2 segundos
        Timer temporizador = new Timer();
        temporizador.schedule(new SistemaDeRiego(), 1000, 2000);

        System.out.printf("Hilo Principal terminado%n");
    }
}
```

Ejercicio 2

Crea el programa **Descanso**. Este programa debe de permanecer 'dormido' durante 30 minutos, y entonces sacar un mensaje de "Toca estirar las piernas". Si quieres, usa las clases **SystemTray** y **TrayIcon** para que Windows lo muestre como una notificación.

Paquete java.util.concurrent

Este paquete contiene clases relacionadas con la programación **Concurrente**. Dicho de otra forma, permiten solucionar los problemas que conlleva que varios Hilos quieran acceder a las mismas variables o recursos a la vez.

Executor

Executor es una *interface* que permite ejecutar tareas de tipo *Runnable*. Su comportamiento es similar al de Timer y TimerTask. Relacionado con Executor tenemos:

Executor	
ExecutorService	Gestiona las tareas asíncronas
ScheduledExecutorService	Permite la planificación de la ejecución de tareas asíncronas
Executors	Fábrica de objetos Executor, ExecutorService, etc.
TimeUnit	Para indicar días, horas, minutos, etc.

Colas de trabajo

Existe un tipo de programas que tienen una estructura de **Colas de Trabajo**, y Java ofrece herramientas para gestionarlas. Hablamos de **Colas de Trabajo** cuando tenemos un programa que tiene que ir ejecutando tareas una detrás de otra, o procesando una serie de mensajes, todo ello de forma concurrente. El problema de estas listas de tareas, como habrás sospechado, es que varios Hilos pueden querer intentar acceder a la vez a un recurso.

ConcurrentLinkedQueue	Una cola enlazada que evita problemas con Hilos
ConcurrentLinkedDeque	Igual, pero con una cola doblemente enlazada
BlockingQueue	Una cola que incluye bloqueos de espera
TransferQueue	Una BlockingQueue especializada para mensajería
BlockingDeque	Una cola doblemente enlazada con bloqueos de espera para la gestión de espacio.

A modo de ejemplo, vamos a ver una cola que recibe escrituras y lecturas de un conjunto de hilos. El primer programa usa estructura **LinkedList** que no es segura frente a hilos, y se produce un error. La segunda usa una **ConcurrentLinkedList**, que evita los problemas que conlleva tener hilos leyendo y escribiendo sobre ella simultáneamente.

Dado que **LinkedList** no es segura contra Hilos, este programa genera un Excepción.

```
public class ColaNoConcurrente implements Runnable{

    // Una cola única para todos los Hilos
    private static Queue <Integer> cola = new LinkedList<Integer>();

    @Override
    public void run() {
        // Cargamos un 10
        cola.add(10);
        for (Integer i : cola) {
            System.out.println(i + ":");
        }
        System.out.println("Tamaño de la cola: " + cola.size());
    }

    public static void main(String[] args) {
        // Lanzamos 10 Hilos
        for (int i = 0; i <10; i++) {
            new Thread (new ColaNoConcurrente()).start();
        }
    }
}
```

En cambio, **ConcurrentLinkedList** permite ejecutar el programa sin problemas.

```
public class ColaConcurrente implements Runnable{

    // Una cola única para todos los Hilos
    private static Queue <Integer> cola = new ConcurrentLinkedDeque<Integer>();

    @Override
    public void run() {
        // Cargamos un 10
        cola.add(10);
        for (Integer i : cola) {
            System.out.println(i + ":");
        }
        System.out.println("Tamaño de la cola: " + cola.size());
    }

    public static void main(String[] args) {
        // Lanzamos 10 Hilos
        for (int i = 0; i <10; i++) {
            new Thread (new ColaConcurrente()).start();
        }
    }
}
```

Sincronizadores

Para realizar una correcta sincronización entre Hilos podemos utilizar las siguientes clases:

Semaphore	Un semáforo que impide continuar la ejecución de los hilos.
CountDownLatch	Permite una sincronización cuando los hilos deben de esperar a que se realicen un conjunto de operaciones.
CyclingBarrier	Permite una sincronización cuando los hilos deben de esperar a que otros hilos alcancen un punto de ejecución concreto
Phaser	Ambas funcionalidades anteriores, pero más flexible
Exchanger	Permite intercambiar elementos entre hilos

A modo de ejemplo, vamos a crear una serie de clases que emplean Exchanger para intercambiar información entre ellas. Ver paquete **exchanger** de los ejemplos. El resultado sería:

Estructuras de datos concurrentes

Si bien Java dispone de un montón de clases e interfaces para almacenar información, no todas ellas están preparadas para trabajar con Hilos. En ciertas ocasiones, sobre todo cuando varios Hilos tratan de leer o escribir en ellas, se generan errores. La clase **Collections** de Java ofrece un montón de métodos estáticos para convertir estructuras de datos en thread-safe.

ConcurrentHashMap	Un HashMap sincronizado
ConcurrentSkipListMap	Un TreeMap sincronizado
CopyOnWriteArrayList	Un ArrayList sincronizado
CopyOnWriteArraySet	Un Set sincronizado

Por ejemplo, este código genera errores cuando varios hilos intentan acceder a leer o modificar la lista *palabras*.

```
public class LectorEscritorInseguro extends Thread {

    private static List<String> palabras = new ArrayList<String>();

    @Override
    public void run() {
        palabras.add(new String("Nueva Palabra"));
        for (String palabra : palabras) {
            palabras.size();
        }
        System.out.println (palabras.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i<100; i++)
            new LectorEscritorInseguro().start();
    }
}
```

Simplemente cambiando el tipo de variable por **CopyOnWriteArrayList** conseguimos solucionar el problema.

```
public class LectorEscritorSeguro extends Thread {

    private static List<String> palabras = new CopyOnWriteArrayList <String>();

    @Override
    public void run() {
        palabras.add(new String("Nueva Palabra"));
        for (String palabra : palabras) {
            palabras.size();
        }
        System.out.println (palabras.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i<100; i++)
            new LectorEscritorSeguro().start();
    }
}
```

Las interfaces ExecutorService, Callable y Future

Estas interfaces permiten ejecutar el Código de forma asíncrona. Esto es, se manda realizar una tarea asíncrona y se deja en espera el tratamiento de la respuesta. **ExecutorService** contiene el marco de ejecución de las demás interfaces.

Métodos de ExecutorService	
awaitTermination	Bloquea el servicio cuando se recibe la petición de apagado hasta que todas las tareas han terminado, se alcanza el tiempo límite de espera o ha habido una interrupción.
invokeAll	Lanza una colección de tareas y recoge el resultado en una lista de objetos Future.
invokeAny	Lanza una colección de tareas y recoge el resultado de la que termine satisfactoriamente
shutdown	El servicio deja de aceptar tareas, espera a que termine las que ya tiene y finaliza
shutdownNow	Finaliza el servicio sin esperar a que terminen las tareas
submit	Envía a ejecución una tarea

Por su parte, **Callable** es similar a Runnable, aunque puede devolver un retorno y lanzar una excepción.

Finalmente, **Future** es una interfaz que representa un resultado futuro generado por un proceso asíncrono. Internamente, deja en suspenso la obtención de un resultado hasta que esté disponible.

Métodos de Future	
cancel	Intenta cancelar la ejecución de una tarea
get	Espera a que termine la tarea y retorna el resultado.
isCancelled	Informa si la tarea se ha cancelado antes de terminar
isDone	Indica si la tarea ha terminado

Es posible utilizar *Runnable* en lugar de *Callable* si no deseamos que la tarea nos proporcione un retorno.

En el siguiente ejemplo, lanzamos a ejecución una tarea asíncrona con un retardo. Hasta que esta no termine, la ejecución del programa no continuará (get()).

```java
public class Lector implements Callable<String> {

    @Override
    public String call() throws Exception {
        String textoLeido = "Me gustan las pelis de accion";
        Thread.sleep(1000); // Añadimos un retardo
        return textoLeido;
    }

    public static void main(String[] args) {
        try {
            // La tarea que queremos ejecutar
            Lector lector = new Lector();

            // El servicio que ejecuta las tareas
            ExecutorService executorService = Executors.newSingleThreadExecutor();

            // El resultado de la tarea
            Future<String> resultado = executorService.submit(lector);

            // ¿Ha terminado?
            String texto = resultado.get();
            if (resultado.isDone()) {
                System.out.println("Resultado = " + texto);
                System.out.println("Tearea finalizada");
            } else if (resultado.isCancelled()){
                System.out.println("Tearea cancelada");
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```