IES SERRA PERENXISA

Unidad didáctica 2
Programación multihilo

Curso 2022/2023
“Programación de servicios y procesos”
CFGS Desarrollo de Aplicaciones Multiplataforma

Basado en los materiales formativos de FP propiedad del Ministerio de Educación, Cultura y Deporte.
Adaptado por: José Miguel Blázquez – Versión: 1.1

Índice

1. Introducción

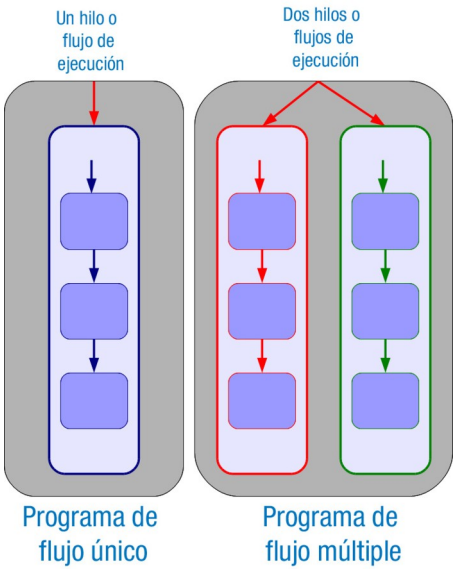
Seguro que en más de una ocasión, mientras te descargabas un fichero desde tu navegador web seguías navegando por Internet e incluso iniciabas la descarga de un nuevo archivo, y todo esto ejecutándose el navegador como un único proceso, es decir, teniendo un único ejemplar del programa en ejecución.

Pues bien, ¿cómo es capaz de hacer el navegador web varias tareas a la vez? Seguro que estarás pensando en la programación concurrente, y así es; pero en este caso en un nuevo enfoque de la concurrencia denominado "programación multihilo".

Los programas realizan actividades o tareas, y para ello pueden seguir uno o más flujos de ejecución. Dependiendo del número de flujos de ejecución podemos hablar de dos tipos de programas:

- **Programa de flujo único**. Es aquel que realiza las actividades o tareas que lleva a cabo una a continuación de la otra, de manera secuencial, lo que significa que cada una de ellas debe concluir por completo, antes de que pueda iniciarse la siguiente.

- **Programa de flujo múltiple**. Es aquel que coloca las actividades a realizar en diferentes flujos de ejecución, de manera que cada uno de ellos se inicia y termina por separado, pudiéndose ejecutar éstos de manera simultánea o concurrente.



La programación **multihilo** o **multithreading** consiste en desarrollar programas o aplicaciones de flujo múltiple. Cada uno de esos flujos de ejecución es un ***thread*** o hilo.

En el ejemplo inicial sobre el navegador web, un hilo se encargaría de la descarga de la imagen, otro de continuar navegando y otro de iniciar una nueva descarga. La utilidad de la programación multihilo resulta evidente en este tipo de aplicaciones. El navegador puede realizar "a la vez" estas tareas, por lo que no habrá que esperar a que finalice una descarga para comenzar otra o seguir navegando.

Cuando decimos "a la vez" recuerda que nos referimos a que las tareas se realizan concurrentemente, pues el que las tareas se ejecuten realmente en paralelo dependerá del Sistema Operativo y del número de procesadores del sistema donde se ejecute la aplicación. En realidad esto es transparente para el programador y usuario, lo importante es la sensación real de que el programa realiza de forma simultánea diferentes tareas.

2. Conceptos sobre hilos

Un **hilo**, denominado también **subproceso**, es un flujo de control secuencial independiente dentro de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila (contexto de ejecución).

Cuando se ejecuta un programa el Sistema Operativo crea un proceso y también crea su primer hilo, el hilo primario, el cual puede a su vez crear hilos adicionales. Desde este punto de vista, un proceso no se ejecuta sino que solo es el espacio de direcciones donde reside el código que es ejecutado mediante uno o más hilos.

Por lo tanto podemos hacer las siguientes **observaciones**:

- Un hilo no puede existir independientemente de un proceso.
- Un hilo no puede ejecutarse por si solo.
- Dentro de cada proceso puede haber varios hilos ejecutándose, pero como mínimo 1.

Un único hilo es similar a un programa **secuencial**; por si mismo no nos ofrece nada nuevo. Es la habilidad de ejecutar varios hilos dentro de un proceso lo que ofrece algo nuevo y útil; ya que cada uno de estos hilos puede ejecutar actividades diferentes al mismo tiempo. Así, en un programa, un hilo puede encargarse de la comunicación con el usuario, mientras que otro hilo descarga una actualización en 2º plano, otro puede acceder a recursos del sistema (cargar sonidos, leer ficheros, …), etc.

2.1. Recursos compartidos por los hilos

Un hilo lleva asociados los siguientes elementos:

- Un identificador único.
- Un contador de programa propio.
- Un conjunto de registros.
- Una pila (variables locales).

Por otra parte, **un hilo puede compartir con otros hilos del mismo proceso** los siguientes recursos:

- Código.
- Datos (como variables globales).
- Otros recursos del sistema operativo, como los ficheros abiertos y las señales.

Es probable que nos venga esta reflexión a la cabeza: "si los hilos de un proceso comparten el mismo espacio de memoria, ¿qué pasa si uno de ellos la corrompe?" La respuesta es que los otros hilos también sufrirán las consecuencias. Recuerda que en el caso de procesos, el sistema operativo normalmente protege a un proceso de otro y si un proceso corrompe su espacio de memoria los demás no se verán afectados

El hecho de que los hilos compartan recursos (por ejemplo, pudiendo acceder a las mismas variables) implica la necesidad de utilizar esquemas de bloqueo y sincronización, lo que puede hacer más difícil el desarrollo de los programas así como su depuración.

Realmente, es en la sincronización de hilos donde reside el arte de programar con hilos; ya que de no hacerlo bien, podemos crear una aplicación totalmente ineficiente o inútil, como por ejemplo, programas que tardan horas en procesar servicios, o que se bloquean con facilidad y que intercambian datos de manera equivocada.

Profundizaremos más adelante en la sincronización, comunicación y compartición de recursos entre hilos dentro del contexto de Java.

2.2. Ventajas y uso de hilos

Como consecuencia de compartir el espacio de memoria, los hilos aportan las siguientes **ventajas sobre los procesos**:

- ✔ Se consumen menos recursos en el lanzamiento y la ejecución de un hilo que en un proceso.
- ✔ Se tarda menos tiempo en crear y terminar un hilo que un proceso.
- ✔ La conmutación entre hilos del mismo proceso o **cambio de contexto** es bastante más rápida que entre procesos.

Es por esas razones, por lo que a los **hilos** se les denomina también **procesos ligeros**.

Y, ¿**cuándo se aconseja utilizar hilos**? Cuando se den alguna o varias de estas circunstancias:

- La aplicación maneja entradas de varios dispositivos de comunicación.
- La aplicación debe poder realizar diferentes tareas a la vez (videojuegos...)
- Interesa diferenciar tareas con una prioridad variada. Por ejemplo, una prioridad alta para manejar tareas de tiempo crítico y una prioridad baja para otras tareas.
- La aplicación se va a ejecutar en un entorno multiprocesador o que debe poder aprovechar ese recurso.

Por ejemplo, imagina la siguiente situación:

- Debes crear una aplicación que se ejecutará en un servidor para atender peticiones de clientes. Esta aplicación podría ser un servidor de bases de datos o un servidor web.
- Cuando se ejecuta el programa, éste abre su puerto y queda a la escucha esperando recibir peticiones.
- Si cuando recibe una petición de un cliente se pone a procesarla para obtener una respuesta y devolverla, cualquier petición que reciba mientras tanto no podrá atenderla, puesto que está ocupado.
- La solución será construir la aplicación con múltiples hilos de ejecución.
- En este caso, al ejecutar la aplicación se pone en marcha el hilo principal que queda a la escucha.
- Cuando el hilo principal recibe una petición, creará un nuevo hilo que se encarga de procesarla y generar la consulta, mientras tanto el hilo principal sigue a la escucha recibiendo peticiones y creando hilos.
- De esta manera se pueden atender consultas de varios clientes.
- Si el número de peticiones simultáneas es elevado, la creación de un hilo para cada una de ellas puede comprometer los recursos del sistema. En este caso, como veremos al final de la unidad, lo resolveremos mejor con un ***pool*** de hilos.

Resumiendo, los hilos son idóneos para programar aplicaciones de entornos interactivos y en red, así como simuladores y animaciones.

Los hilos son más frecuentes de lo que parece. De hecho, todos los programas con interfaz gráfica son multihilo porque los eventos y las rutinas de dibujado de las ventanas corren en un hilo distinto al principal. Por ejemplo en Java, AWT o la biblioteca gráfica Swing usan hilos. Si en "Programación" de 1er curso usaste Swing, ahora entenderás algunas cosas "raras" que aparecían en el código.

3. Multihilo en Java: librerías y clases

Java da soporte al concepto de hilo desde el **propio lenguaje** con algunas clases e interfaces definidas en el paquete j**ava.lang** y con métodos específicos para la manipulación de hilos en la clase **Object**. A partir de la versión JavaSE 5.0, se incluye el paquete **java.util.concurrent** con nuevas utilidades para desarrollar aplicaciones multihilo e incluso aplicaciones con un alto nivel de concurrencia.

3.1. Utilidades de concurrencia del paquete java.lang

Dentro del paquete **java.lang** disponemos de una interfaz y las siguientes clases para trabajar con hilos:

- Clase **Thread**. Responsable de producir hilos funcionales para otras clases y proporciona gran parte de los métodos utilizados para su gestión.
- Interfaz **Runnable**. Proporciona la capacidad de añadir la funcionalidad de hilo a una clase simplemente implementando la interfaz, en lugar de derivándola de la clase Thread, permitiendo herencia "múltiple".
- Clase **ThreadDeath**. Es una clase de error, deriva de la clase Error, y proporciona medios para manejar y notificar errores.
- Clase **ThreadGroup**. Esta clase se utiliza para manejar un grupo de hilos de modo conjunto, de manera que se pueda controlar su ejecución de forma eficiente.
- Clase **Object**. Esta clase no es estrictamente de apoyo a los hilos, pero proporciona unos cuantos métodos cruciales dentro de la arquitectura multihilo de Java. Estos métodos son wait(), notify() y notifyAll().

3.2. Utilidades de concurrencia del paquete java.util.concurrent

El paquete **java.util.concurrent** incluye una serie de clases que facilitan enormemente el desarrollo de aplicaciones multihilo y aplicaciones complejas, ya que están concebidas para utilizarse como bloques de diseño.

Concretamente estas utilidades están dentro de los siguientes paquetes:

- ✔ **java.util.concurrent**. En este paquete están definidos los siguientes elementos:
  - Clases de **sincronización**. *Semaphore*, *CountDownLatch*, *CyclicBarrier* y *Exchanger*.
  - **Interfaces** para separar la lógica de la ejecución, como por ejemplo *Executor*, *ExecutorService*, *Callable* y *Future*.
  - **Interfaces** para gestionar colas de hilos. *BlockingQueque*, *LinkedBlokingQueque*, *ArrayBlockingQueque*, *SynchronousQueque*, *PriorityBlockingQueque* y *DelayQueque*.
- ✔ **java.util.concurrent.atomic**. Incluye un conjunto de clases para ser usadas como variables atómicas en aplicaciones multihilo y con diferentes tipos de dato, por ejemplo *AtomicInteger* y *AtomicLong*.
- ✔ **java.util.concurrent.locks**. Define una serie de clases como uso alternativo a la cláusula *synchronized*. En este paquete se encuentran algunas interfaces como por ejemplo *Lock*, *ReadWriteLock*.

A lo largo de esta unidad estudiaremos las clases e interfaces más importantes de este paquete.

4. Creación de hilos

En **Java**, un hilo se representa mediante una instancia de la clase **java.lang.thread**. Este objeto **Thread** se emplea para iniciar, detener o cancelar la ejecución del hilo de ejecución. Los hilos o threads se pueden implementar o definir de dos formas:

1) **Extendiendo** la clase **Thread**.
2) **Mediante** la **interfaz Runnable** .

En ambos casos, se debe proporcionar una definición del **método *run()***, ya que este método es el que contiene el código que ejecutará el hilo, es decir, su comportamiento.

El procedimiento de construcción de un hilo es independiente de su uso, pues una vez creado se emplea de la misma forma. Entonces, ¿cuando utilizar uno u otro procedimiento?

- ➢ Extender la clase **Thread** es el procedimiento más sencillo pero no siempre es posible. Si la clase ya hereda de alguna otra clase padre, no será posible heredar también de la clase Thread (recuerda que Java no permite la herencia múltiple), por lo que habrá que recurrir al otro procedimiento. No está recomendado.
- ➢ Implementar **Runnable** siempre es posible, es el procedimiento más general y también el más flexible.

Cuando la Máquina Virtual Java (JVM) arranca la ejecución de un programa, ya hay un hilo ejecutándose denominado hilo principal del programa, y controlado por el método main(), que se ejecuta cuando comienza el programa y es el último hilo que finaliza su ejecución, ya que cuando este hilo acaba el programa termina.

Observa este código de ejemplo:

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

    System.out.println("¡Hola mundo!\n");

    //obtiene el hilo donde se está ejecutando el método main
    Thread miHilo = Thread.currentThread();

    //imprime el nombre del hilo en la Salida (función getName())
    System.out.println("El hilo se llama " + miHilo.getName() + "\n");
  }
}
```

4.1. Extendiendo la clase Thread

1. Crear una nueva clase que herede de la clase Thread.
2. Redefinir en la nueva clase el método run() con el código a ejecutar asociado al hilo.
3. Crear un objeto de la nueva clase Thread. Éste será realmente el hilo.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo invocaremos al método start() del objeto thread (el hilo que hemos creado).

En el siguiente ejemplo puedes ver los pasos indicados anteriormente para la creación de un hilo extendiendo la clase Thread. El hilo que se crea (objeto thread hilo1) imprime un mensaje de saludo. Para simplificar el ejemplo se ha incluido el método main() que inicia el programa en la propia clase Saludo.

```
public class Saludo extends Thread {
//clase  que extiende a Thread
   public void run() {
   // se redefine el método run() con el código asociado al hilo
       System.out.println("¡Saludo desde un hilo extendiendo thread!");
   }
   public static void main(String args[]) {
       Saludo hilo1=new Saludo();
       //se crea un objeto Thread, el hilo hilo1
       hilo1.start();
       //invoca a start() y pone en marcha el hilo hilo1
   }
 }
```

4.2. Implementando la interfaz Runnable

1. Declarar una nueva clase que implemente a Runnable.
2. Redefinir en la nueva clase el método run() con el código asociado al hilo.
3. Crear un objeto de la nueva clase.
4. Crear un objeto de la clase Thread pasando como argumento al constructor el objeto cuya clase tiene el método run(). Este será realmente el hilo.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo invocaremos al método **`start()`** del objeto thread (el hilo que hemos creado).

El siguiente ejemplo muestra cómo crear un hilo implementado Runnable. El hilo que se crea (objeto thread hilo1) imprime un mensaje de saludo, como en el caso anterior.

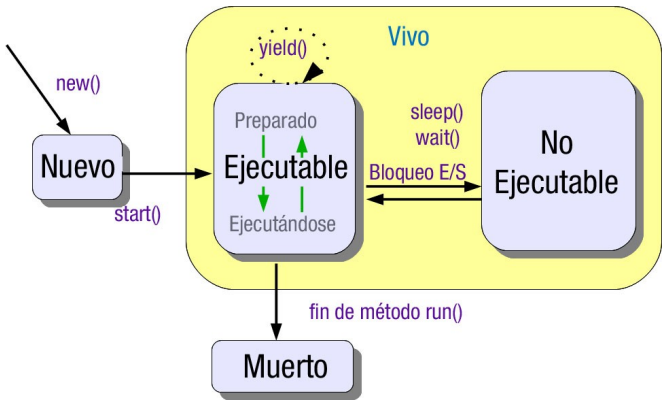
```
public class Saludo implements Runnable {
//clase que implementa a Runnable
    public void run() {
    //se redefine el método run()con el código asociado al hilo
        System.out.println("¡Saludo desde un hilo creado con Runnable!");
    }
    public static void main(String args[]) {
       Saludo  miRunnable=new Saludo();
       //se crea un objeto  Saludo
       Thread hilo1= new Thread(miRunnable);
       //se crea un objeto Thread (el hilo hilo1) pasando como argumento
      // al constructor un objeto Saludo
        hilo1.start();
       //se invoca al método start() del hilo hilo1
    }
}
```

5. Estados de un hilo

El ciclo de vida de un hilo comprende los diferentes estados en los que puede estar un hilo desde que se crea o nace hasta que finaliza o muere. De manera general, los diferentes estados en los que se puede encontrar un hilo son los siguientes:

- **Nuevo** (new): se ha creado un nuevo hilo, pero aún no está disponible para su ejecución.
- **Ejecutable** (runnable): el hilo está preparado para ejecutarse. Puede estar Ejecutándose, siempre y cuando se le haya asignado tiempo de procesamiento, o bien que no esté ejecutándose en un instante determinado en beneficio de otro hilo, en cuyo caso estará Preparado.
- **No Ejecutable o Detenido** (no runnable): el hilo podría estar ejecutándose, pero hay alguna actividad interna al propio hilo que se lo impide, como por ejemplo una espera producida por una operación de Entrada/Salida (E/S). Si un hilo está en estado "No Ejecutable", no tiene oportunidad de que se le asigne tiempo de procesamiento.
- **Muerto** o **Finalizado** (terminated): el hilo ha finalizado. La forma natural de que muera un hilo es finalizando su método run().

El método getState() de la clase Thread, permite obtener en cualquier momento el estado en el que se encuentra un hilo. Devuelve por tanto: NEW, RUNNABLE, NO RUNNABLE o TERMINATED.



5.1. Iniciar un hilo

Cuando se crea un nuevo hilo mediante el método **new()**, no implica que el hilo ya se pueda ejecutar. Para que el hilo se pueda ejecutar debe estar en el estado "**Ejecutable**", y para conseguir ese estado es necesario iniciar o arrancar el hilo mediante el método **start()** de la clase **Thread**.

En los ejemplos anteriores recuerda que teníamos el código **hilo1.start()**; que se encargaba de iniciar el hilo representado por el objeto thread hilo1. El método start() realiza las siguientes tareas:

- **Crea** los **recursos** del sistema necesarios para ejecutar el hilo.
- Se encarga de **llamar** a su método **run()** y lo ejecuta como un subproceso nuevo e independiente.

Es por esto último que cuando se invoca a **start()** se suele decir que el hilo está "corriendo" ("running"), pero recuerda que esto no significa que el hilo esté ejecutándose en todo momento, ya que **un hilo "Ejecutable" puede estar "Preparado" o "Ejecutándose"** según tenga o no asignado tiempo de procesamiento.

Algunas **consideraciones importantes** que debes tener en cuenta son las siguientes:

- Puedes invocar directamente al método **run()**, por ejemplo poner **hilo1.run()**; y se ejecutará el código asociado a **run()** dentro del hilo actual (como cualquier otro método), pero no comenzará un nuevo hilo como subproceso independiente.

- Una vez que se ha llamado al método **start()** de un hilo no se puede volver a realizar otra llamada al mismo método. Si lo haces obtendrás una excepción ***IllegalThreadStateException***.

- El orden en el que inicies los hilos mediante **start()** no influye en el orden de ejecución de los mismos, lo que pone de manifiesto que el orden de ejecución de los hilos es *no-determinístico* (no se conoce la secuencia en la que serán ejecutadas la instrucciones del programa y no tiene porqué coincidir si se lanza el programa varias veces).

En el siguiente recurso didáctico puedes ver un programa que define dos hilos construidos cada uno de ellos por los 2 procedimientos vistos anteriormente. Cada hilo imprime una palabra 5 veces. Observa que, si ejecutas varias veces el programa, el orden de ejecución de los hilos no es siempre el mismo y que no influye en absoluto el orden en el que se inician con start() (el orden de ejecución de los hilos es no-determinístico). **Nota:** puede que tengas que aumentar el número de iteraciones para apreciar las observaciones indicadas anteriormente.

```
public class Hilo_Thread extends Thread {
//clase que extiende a Thread con 2 constructores

    String nombre = "Hilo_Thread";

    public Hilo_Thread(String nb) {
        //constructor 1
        nombre = nb;
    }

    public Hilo_Thread() {
        //constructor 2
    }

    @Override
    public void run() {
        //redefinimos run() con el código asociado al hilo
        for (int i = 1; i <= 5; i++) {
            System.out.println(nombre);
        }
    }
}

public class Hilo_Runnable implements Runnable {
    //clase que implementa Runnable
    public void run() {
        //redefinimos run() con el código asociado al hilo
        for (int i = 1; i <= 5; i++) {
            System.out.println("Hilo_Runnable");
        }
    }
}

public class Main {
   public static void main(String[] args) {
        //creamos 2 hilos del tipo Hilo_Thread usando ambos constructores
        Thread hilo1 = new Hilo_Thread("Jose");
        Thread hilo2 = new Hilo_Thread();

        //creamos un hilo Runnable en un sólo paso
        Thread hilo3 = new Thread(new Hilo_Runnable());

        //ponemos en marcha los 3 hilos
        hilo1.start();
        hilo2.start();
        hilo3.start();
    }
}
```

5.2. Detener temporalmente un hilo

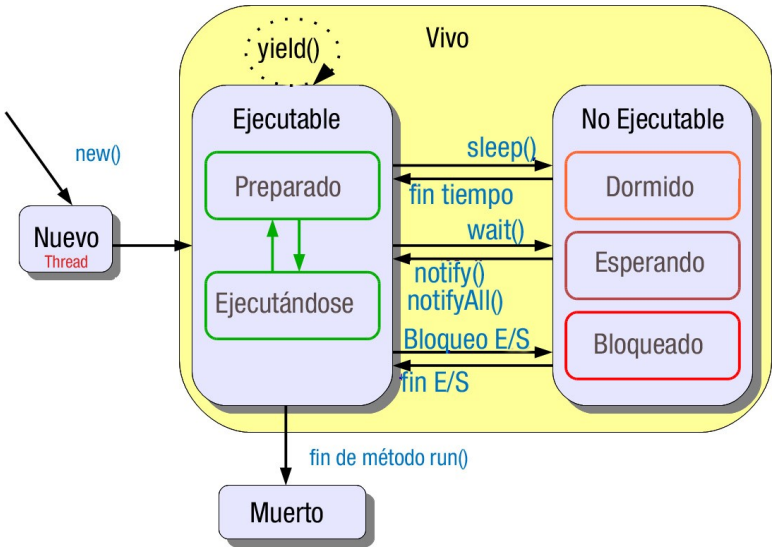
¿Qué significa que un hilo se ha detenido temporalmente? Significa que el hilo ha pasado al estado "No Ejecutable". Y, ¿cómo puede pasar un hilo al estado "No Ejecutable"? Un hilo pasará al estado "No Ejecutable" o "Detenido" por alguna de estas circunstancias:

- El hilo se ha **dormido**. Se ha invocado al método **sleep()** de la clase Thread indicando el tiempo que el hilo permanecerá detenido. Transcurrido ese tiempo, el hilo se vuelve "Ejecutable", en concreto pasa a "Preparado".

- El hilo está **esperando**. El hilo ha detenido su ejecución mediante la llamada al método **wait()** y no se reanudará, pasando a "Ejecutable" (en concreto "Preparado"), hasta que se produzca una llamada al método **notify()** o **notifyAll()** por **otro hilo**. Estudiaremos detalladamente estos métodos de la clase **Object** cuando veamos la **sincronización** y **comunicación** de hilos.

- El hilo se ha **bloqueado**. El hilo está pendiente de que finalice una **operación de E/S** en algún dispositivo, o a la **espera** de algún otro tipo de recurso; y éste ha sido bloqueado por el sistema operativo. Cuando finalice el bloqueo volverá al estado "Ejecutable", en concreto "Preparado".

En la siguiente imagen puedes ver un esquema con los diferentes métodos que hacen que un hilo pase al estado "No Ejecutable", así cómo los que permiten salir de ese estado y volver al estado "Ejecutable".



El método **suspend()** (actualmente obsoleto o *deprecated*) también permite detener temporalmente un hilo. En ese caso se reanudaría mediante el método **resume()** (también en desuso). No debes utilizar estos métodos de la clase Thread ya que no son seguros y provocan muchos problemas. Lo explicamos porque puede que encuentres programas que aún utilizan estos métodos.

5.3. Finalizar un hilo

La forma natural de que finalice un hilo es cuando termina de ejecutarse su método **run()**, pasando al estado 'Muerto'.

Una vez que el hilo ha **muerto**, **no lo puedes iniciar** otra vez con **start()**. Si en tu programa deseas realizar otra vez el trabajo desempeñado por el hilo, tendrás que:

1) Crear un nuevo hilo con new().
2) Iniciar el hilo con start().

Puedes utilizar el método **isAlive()** de la clase **Thread** para comprobar si un hilo está vivo o no. Un hilo se considera que está vivo (**alive**) desde la llamada a su método **start()** hasta su **muerte**. **isAlive()** devuelve verdadero (**true**) o falso

(**false**), según que el hilo esté vivo o no.

Cuando el método **isAlive()** devuelve:

- ✔ False: sabemos que estamos ante un nuevo hilo recién "creado" o ante un hilo "muerto".
- ✔ True: sabemos que el hilo se encuentra en estado "ejecutable" o "no ejecutable".

El método **stop()** de la clase **Thread** (actualmente en **desuso**) también finaliza un hilo, pero es **poco seguro**. No debes utilizarlo. Te lo indicamos aquí simplemente porque puede que encuentres programas utilizando este método.

En el siguiente ejemplo te proporcionamos un programa cuyo hilo principal lanza un hilo secundario que realiza una cuenta atrás desde 10 hasta 1. Desde el hilo principal se verificará la muerte del hilo secundario mediante la función isAlive(). Además mediante el método getState() de la clase Thread vamos obteniendo el estado del hilo secundario. Se usa también el método thread.join() que espera hasta que el hilo muere.

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class Hilo_Auxiliar extends Thread{ //código del hilo
  @Override
  public void run(){
    for(int i=10;i>=1;i--)
      System.out.print(i+",");
  }
}

public class Main {

    public static void main(String[] args) {
          //Crea un nuevo hilo en estado "Nuevo"
          Hilo_Auxiliar hilo1 = new Hilo_Auxiliar();

          //Obtenemos el estado del thread hilo1 y si está vivo o no
          System.out.println("Hilo Aux Estado=" + hilo1.getState());
          System.out.println("¿Vivo?=" + hilo1.isAlive());

          //Inicia el thread hilo1 y pasa al estado Ejecutable
          hilo1.start();

          System.out.println("Hilo Aux Estado=" + hilo1.getState());
          System.out.println("¿Vivo?=" + hilo1.isAlive() + "\n");

          try {
              //espera a que el thread hilo1 muera
              hilo1.join();
          } catch (InterruptedException e) {
              System.out.println(e);
          }

          System.out.println("Hilo Aux Estado=" + hilo1.getState());
          System.out.println("¿Vivo?=" + hilo1.isAlive() + "\n");
    }
}
```

5.4. Dormir un hilo con sleep

¿Por qué puede interesar dormir un hilo? Un ejemplo podría ser en una aplicación gráfica, si no durmiéramos unos instantes al hilo que realiza un cálculo, no le daría tiempo al hilo que dibuja el resultado a presentarlo en pantalla al ser esta operación más lenta que la del cálculo.

El método **sleep()** de la clase **Thread** recibe como argumento el tiempo que deseamos dormir el hilo que lo invoca. Cuando transcurre el tiempo especificado, el hilo vuelve a estar "Ejecutable" ("Preparado") para continuar ejecutándose.

Hay dos formas de llamar a este método:

- ✔ La primera le pasa como argumento un entero (positivo) que representa milisegundos:
  `sleep(long milisegundos)`
- ✔ La segunda le agrega un segundo argumento entero (esta vez, entre 1 y 999999), que representa un tiempo extra en nanosegundos que se sumará al primer argumento:
  `sleep(long milisegundos, int nanosegundos)`

Cualquier llamada a **`sleep()`** puede provocar una **excepción** que el compilador de Java nos obliga a controlar ineludiblemente mediante un bloque **`try-catch.`**

6. Gestión y planificación de hilos

La ejecución de hilos se puede realizar mediante:

- **Paralelismo**. En un sistema con múltiples CPU, cada CPU puede ejecutar un hilo diferente.
- **Pseudoparalelismo**. Si no es posible el paralelismo, una CPU es responsable de ejecutar múltiples hilos. La ejecución de múltiples hilos en una sola CPU requiere la planificación de una secuencia de ejecución.

El **planificador de hilos de Java** (**Scheduler**) utiliza un algoritmo de secuenciación de hilos denominado fixed priority scheduling que está basado en un sistema de prioridades relativas, de manera que el algoritmo secuencia la ejecución de hilos en base a la prioridad de cada uno de ellos.

El funcionamiento del algoritmo es el siguiente:

- El hilo elegido para ejecutarse siempre es el hilo "Ejecutable" de prioridad más alta.
- Si hay más de un hilo con la misma prioridad, el orden de ejecución se maneja mediante un algoritmo por turnos (round-robin) basado en una cola circular FIFO.
- Cuando el hilo que está "ejecutándose" pasa al estado de "No Ejecutable" o "Muerto", se selecciona otro hilo para su ejecución.
- La ejecución de un hilo se interrumpe si otro hilo con prioridad más alta se vuelve "Ejecutable". El hecho de que un hilo con una prioridad más alta interrumpa a otro se denomina "planificación apropiativa" ('preemptive scheduling').

Pero la responsabilidad de ejecución de los hilos es del **Sistema Operativo** sobre el que corre la JVM, y Sistemas Operativos distintos manejan los hilos de manera diferente:

- ✔ En un **Sistema Operativo que implementa time-slicing** (subdivisión de tiempo), el hilo que entra en ejecución se mantiene en ella sólo un micro-intervalo de tiempo fijo o cuanto (quantum) de procesamiento, de manera que el hilo que está "ejecutándose" no solo es interrumpido si otro hilo con prioridad más alta se vuelve "Ejecutable", sino también cuando su "cuanto" de ejecución se acaba. Es el patrón seguido por Linux y por todos los Windows a partir de Windows 95 y NT.

- ✔ En un **Sistema Operativo que no implementa time-slicing,** el hilo que entra en ejecución es ejecutado hasta su muerte; salvo que regrese a "No ejecutable", u otro hilo de prioridad más alta alcance el estado de "Ejecutable" (en cuyo caso, el primero regresa a "preparado" para que se ejecute el segundo). Es el patrón seguido en el Sistema Operativo Solaris.

*Una tecnología comercial como "**Hyper-Threading**", salió al mercado de la mano de Intel desde los primeros Pentium 4, con el objetivo de permitir que los programas que estén preparados para ello ejecuten tareas usando múltiples hilos, lo cual es un procesamiento en paralelo dentro de un único procesador, incrementando así el uso de las unidades de*

ejecución del procesador.

6.1. Prioridad de hilos

En Java, **cada hilo tiene una prioridad** representada por un valor entero entre 1 y 10. Cuanto **mayor** es el valor, **mayor** es la prioridad del hilo.

Por defecto, el hilo principal de cualquier programa (el que ejecuta su método main()) es creado con prioridad 5.

El resto de hilos secundarios (creados desde el hilo principal, o desde cualquier otro hilo en funcionamiento), **heredan** la **prioridad** que tenga en ese momento su hilo padre.

En la clase **Thread** se definen 3 constantes para manejar estas prioridades:

- `MAX_PRIORITY` (= 10). Es el valor que simboliza la máxima prioridad.
- `MIN_PRIORITY` (=1). Es el valor que simboliza la mínima prioridad.
- `NORM_PRIORITY` (= 5). Es el valor que simboliza la prioridad normal.

Además en cualquier momento se puede **obtener** y **modificar** la **prioridad** de un **hilo** mediante los siguientes métodos de la clase **Thread**:

- **`getPriority()`:** Obtiene la prioridad de un hilo. Este método devuelve la prioridad del hilo.
- **`setPriority()`:** Modifica la prioridad de un hilo. Este método toma como argumento un entero entre 1 y 10, que indica la nueva prioridad del hilo.

Java tiene 10 niveles de prioridad, y éstos no tienen por qué coincidir con los del sistema operativo sobre el que está corriendo. Por ello, se recomienda utilizar las constantes MAX_PRIORITY, NORM_PRIORITY y MIN_PRIORITY.

Podemos conseguir aumentar el rendimiento de una aplicación multihilo gestionando adecuadamente las prioridades de los diferentes hilos, por ejemplo, utilizando una prioridad alta para tareas de tiempo crítico y una prioridad baja para otras tareas menos importantes.

En el siguiente ejemplo se declara un hilo cuya tarea es llenar un vector con 20000 caracteres. Se inician 15 hilos con prioridades diferentes, 5 con prioridad máxima, 5 con prioridad normal y 5 con prioridad mínima. Al ejecutar el programa comprobarás que los hilos con prioridad más alta tienden a finalizar antes. Observa que se usa también el método **`yield()`** del que hablaremos en el siguiente apartado.

```
public class Hilo extends Thread {

    public Hilo() { //hereda la prioridad del hilo padre
    }

    public Hilo(int prioridad) {
        //establece la prioridad indicada
        this.setPriority(prioridad);
    }

    @Override
    public void run() {

        String strCadena = "";
        //agrega 30000 caracteres a una cadena vacía
        for (int i = 0; i < 20000; ++i) {
            //imprime el valor en la Salida
            strCadena += "A";
            //yield() sugiere al planificador Java seleccionar otro hilo
            yield();
        }
```

```
        System.out.println("Hilo de prioridad " + this.getPriority()
                + " termina ahora");
    }
}

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

        int contador = 5;

        //vectores para hilos de distintas prioridades
        Thread[] hiloMIN = new Thread[contador];
        Thread[] hiloNORM = new Thread[contador];
        Thread[] hiloMAX = new Thread[contador];

        //crea los hilos de prioridad mínima
        for (int i = 0; i < contador; i++) {
            hiloMIN[i] = new Hilo(Thread.MIN_PRIORITY);
        }

        //crea los hilos de prioridad normal
        for (int i = 0; i < contador; i++) {
            hiloNORM[i] = new Hilo();
        }

        //crea los hilos de máxima prioridad
        for (int i = 0; i < contador; i++) {
            hiloMAX[i] = new Hilo(Thread.MAX_PRIORITY);
        }

        System.out.println("Los hilos de mayor prioridad terminan antes...\n");

        //inicia los hilos
        for (int i = 0; i < contador; i++) {
            hiloMIN[i].start();
            hiloNORM[i].start();
            hiloMAX[i].start();
        }

    }
}
```

6.2. Hilos egoístas y programación expulsora

En un Sistema Operativo que no implemente time-slicing puede ocurrir que un hilo que entra en "ejecución" no salga de ella hasta su muerte, de manera que no dará ninguna posibilidad a que otros hilos "preparados" entren en "ejecución" hasta que él muera. Este hilo se habrá convertido en un **hilo egoísta**.

Por ejemplo, supongamos la siguiente situación de ejemplo en un Sistema Operativo **sin time-slicing**:

- La tarea asociada al método **run()** de un hilo consiste en imprimir 100 veces la palabra que se le pasa al constructor más el número de orden.
- Se inician 2 hilos en **main()**, uno imprimiendo "**Azul**" y otro "**Rojo**".
- El hilo que sea seleccionado en primer lugar por el planificador se ejecutará íntegramente, por ejemplo, el que imprime "Rojo" 100 veces. Después se ejecutará el otro hilo, tal y como muestra la imagen parcial de la derecha.
- Este hilo tiene un comportamiento egoísta.

En un Sistema Operativo que **si implemente time-slicing** la ejecución de esos hilos se entremezcla, tal y como muestra la imagen parcial, lo cual indica que no hay comportamiento egoísta de ningún hilo, esto es, el Sistema operativo combate los hilos egoístas.

Según lo anterior, un mismo programa Java se puede ejecutar de diferente manera según el Sistema Operativo donde corra. Entonces, ¿que pasa con la **portabilidad** de Java? Java da solución a este problema mediante lo que se conoce como programación expulsora a través del método **`yield()`** de la clase **`java.lang.Thread`**:

`yield()` hace que un hilo que está "ejecutándose" pase a "preparado" para permitir que otros hilos de la misma prioridad puedan ejecutarse. Sobre el método **`yield()`** y el egoísmo de los **threads** debes tener en cuenta que:

- ✔ El funcionamiento de **`yield()`** no está garantizado, puede que después de que un hilo invoque a **`yield()`** y pase a "preparado", éste vuelva a ser elegido para ejecutarse.
- ✔ No debes asumir que la ejecución de una aplicación se realizará en un Sistema Operativo que implementa time-slicing.
- ✔ En la aplicación debes incluir adecuadamente llamadas al método **`yield()`**, incluso a **`sleep()`** o **`wait()`**, si el hilo no se bloquea por una Entrada/Salida.

El siguiente código muestra la forma de invocar a **`yield()`** dentro del método **`run()`** de un hilo. Ten en cuenta que si la invocación se hace desde un hilo **`Runnable`** tendrás que poner **`thread.yield();`**

```
public class Color extends Thread {
    String color;
    public Color (String c){
        color = c;
    }
   public void run(){
       //imprime 100 veces el valor: color + i
       for(int i=1;i<=100;i++)
            System.out.println(color + i);
            yield();
    }
}

public class Main {
 public static void main(String[] args) {
       //se crean 2 hilos: para rojo y azul
       Color hrojo = new Color ("Rojo");
       Color hazul = new Color ("Azul");
       //se inician los hilos para su ejecución
       hrojo.start();
       hazul.start();
    }
}
```

7. Sincronización y comunicación de hilos

Los ejemplos realizados hasta ahora utilizan hilos independientes; una vez iniciados los hilos, éstos no se relacionan con los demás y no acceden a los mismos datos u objetos, por lo que no hay conflictos entre ellos. Sin embargo, hay ocasiones en las que distintos hilos de un programa necesitan establecer alguna relación entre sí y compartir recursos o información. Se pueden presentar las siguientes situaciones:

- 2 o más hilos compiten por obtener un mismo recurso, por ejemplo dos hilos que quieren escribir en un mismo fichero o acceder a la misma variable para modificarla.
- 2 o más hilos colaboran para obtener un fin común y para ello, necesitan comunicarse a través de algún recurso. Por ejemplo un hilo produce información que utilizará otro hilo.

En cualquiera de estas situaciones, es necesario que los hilos se ejecuten de manera controlada y coordinada para evitar posibles interferencias que pueden desembocar en programas que se bloquean con facilidad y que intercambian datos de manera equivocada.

Para conseguir esto tenemos:

- **Sincronización**. Es la capacidad de informar de la situación de un hilo a otro. El objetivo es establecer la secuencialidad correcta del programa.
- **Comunicación**. Es la capacidad de transmitir información desde un hilo a otro. El objetivo es el intercambio de información entre hilos para operar de forma coordinada.

En Java la sincronización y comunicación de hilos se consigue mediante:

- **Monitores**. Se crean al marcar bloques de código con la palabra **`synchronized`**.
- **Semáforos**. Podemos implementar nuestros propios semáforos, o bien utilizar la clase **`Semaphore`** incluida en el paquete **`java.util.concurrent`**.
- **Notificaciones**. Permiten comunicar hilos mediante los métodos **`wait`**(), **`notify`**() y **`notifyAll`**() de la clase **`java.lang.Object`**.

Por otra parte, Java proporciona en el paquete **java.util.concurrent** varias clases de sincronización que permiten la sincronización y comunicación entre diferentes hilos de una aplicación multithreading, como son: **Semaphore**, **CountDownLatch**, **CyclicBarrier** y **Exchanger**.

En los siguientes apartados veremos ejemplos de todo esto.

7.1. Información compartida entre hilos

Las **secciones críticas** son aquellas secciones de código que **no** pueden ejecutarse **concurrentemente**, pues en ellas se encuentran los recursos o información que comparten diferentes hilos y que por tanto pueden ser problemáticas.

Veamos un ejemplo. En él, se pone de manifiesto el problema conocido como la "**condición de carrera**", que se produce cuando varios hilos acceden a la vez a un mismo recurso, por ejemplo a una variable, cambiando su valor y obteniendo de esta forma un valor no esperado de la misma.

tiempo	thread1 { c = c + 1; }	thread2 { c = c - 1; }
t0	c = 100	
t1	Lee 100	
t2	Incrementa a 101	
t3		Lee 100
t4	Escribe 101	
t5		Decrementa a 99
t6		Escribe 99
final	c = 99	

La forma de proteger las secciones críticas es mediante **sincronización**. La **sincronización** se consigue mediante:

- **Exclusión mutua**. Asegurar que un hilo tiene acceso a la sección crítica de forma exclusiva y por un tiempo finito.
- **Por condición**. Asegurar que un hilo no progrese hasta que se cumpla una determinada condición.

En **Java**, la **sincronización** para el acceso a recursos compartidos se basa en el concepto de **monitor**.

7.2. Monitores. Métodos synchronized

En **Java**, un **monitor** es una **porción de código protegida** por un **mutex** o **lock**. Para crear un monitor en Java hay que marcar un bloque de código con la palabra ***synchronized***, pudiendo ser ese bloque un método completo o cualquier segmento de código.

Añadir ***synchronized*** a un método significará que:

- Hemos creado un monitor asociado al objeto.
- Sólo un hilo puede ejecutar el método ***synchronized*** de ese objeto a la vez.
- Los hilos que necesitan acceder a ese método ***synchronized*** permanecerán bloqueados y en espera.
- Cuando el hilo finaliza la ejecución del método ***synchronized***, los hilos en espera de poder ejecutarlo se desbloquearán. El planificador Java seleccionará a uno de ellos.

En el siguiente código se muestra un sencillo ejemplo que simula el acceso simultáneo de 4 terminales a un servidor utilizando monitores Java.

```
public class ServidorWeb { //clase que simula los accesos a un servidor
    private int cuenta;
    public ServidorWeb() {
        cuenta = 0;
    }
    public synchronized void incrementaCuenta() { //método sincronizado
        System.out.println("hilo " + Thread.currentThread().getName()
                + "----- Entra en Servidor");
        //cuenta cada acceso al servidor y muestra el número de accesos
        cuenta++;
        System.out.println(cuenta + " accesos");
    }
}

public class Hilo_Terminal extends Thread {
```

```
    private ServidorWeb servidor;
    public Hilo_Terminal(ServidorWeb s) {
        this.servidor = s;
    }

    @Override
    public void run() {
        //método que incrementa la cuenta de accesos
        for (int i = 1; i <= 10; i++) { //se simulan 10 accesos
            servidor.incrementaCuenta();
            yield();
         }
        }
    }

public class Main {

    public static void main(String[] args) {
        ServidorWeb servidor = new ServidorWeb();

        Hilo_Terminal hterminal1 = new Hilo_Terminal(servidor);
        Hilo_Terminal hterminal2 = new Hilo_Terminal(servidor);
        Hilo_Terminal hterminal3 = new Hilo_Terminal(servidor);
        Hilo_Terminal hterminal4 = new Hilo_Terminal(servidor);

        hterminal1.start();
        hterminal2.start();
        hterminal3.start();
        hterminal4.start();
    }
}
```

7.3. Monitores. Segmentos de código *synchronized*

Hay casos en los que no se puede o no interesa sincronizar un método. Por ejemplo, no podremos sincronizar un método que no hemos creado nosotros y que por tanto no podemos acceder a su código fuente para añadir **synchronized** en su definición. La forma de resolver esta situación es poner las llamadas a los métodos que se quieren sincronizar dentro de segmentos sincronizados de la siguiente forma:

```
synchronized (objeto){
   // sentencias segmento;
}
```

En este caso el funcionamiento es el siguiente:

- El objeto que se pasa al segmento es el objeto donde está el método que se quiere sincronizar.
- Dentro del segmento se hará la llamada al método que se quiere sincronizar.
- El hilo que entra en el segmento declarado ***synchronized*** se hará con el monitor del objeto, si está libre; o se bloqueará en espera de que quede libre. El monitor se libera al salir el hilo del segmento de código ***synchronized***.
- Sólo un hilo puede ejecutar el segmento ***synchronized*** a la vez.

Debes tener en cuenta que:

- Declarar un método o segmento de código como sincronizado ralentizará la ejecución del programa, ya que la adquisición y liberación de monitores genera una sobrecarga.
- Siempre que sea posible, por legibilidad del código, es mejor sincronizar métodos completos.
- Al declarar bloques ***synchronized*** puede aparecer un nuevo problema, denominado **interbloqueo** (lo veremos más adelante).

Muchos métodos de las clases predefinidas de Java ya están sincronizados. Por ejemplo, el método de la clase Component de Java AWT que agrega un objeto MouseListener a un Component (para que MouseEvents se registren en el MouseListener) está sincronizado. Si compruebas el código fuente de AWT y Swing, encontrarás que el prototipo de este método es: **`public synchronized void addMouseListener(MouseListener l)`**

7.4. Comunicación entre hilos con métodos de java.lang.Object

La comunicación entre hilos la podemos ver como un mecanismo de auto-sincronización, que consiste en lograr que un hilo actúe solo cuando otro ha concluido cierta actividad (y viceversa).

Java soporta comunicación entre hilos mediante los siguientes métodos de la clase **java.lang.Object.**

- **`wait():`** Detiene el hilo (pasa a "no ejecutable"), el cual no se reanudará hasta que otro hilo notifique que ha ocurrido lo esperado.
- **`wait(long tiempo):`** Como el caso anterior, solo que ahora el hilo también puede reanudarse (pasar a "ejecutable·) si ha concluido el tiempo pasado como parámetro.
- **`notify():`** Notifica a uno de los hilos puestos en espera para el mismo objeto que ya puede continuar.
- **`notifyAll():`** Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar.

*La llamada a estos métodos se realiza dentro de bloques **synchronized**.*

2 problemas clásicos que permiten ilustrar la necesidad de sincronizar y comunicar hilos son:

1) El problema del **Productor-Consumidor**. Permite modelar situaciones en las que se divide el trabajo entre los hilos. Modela el acceso simultáneo de varios hilos a una estructura de datos u otro recurso, de manera que unos hilos producen y almacenan los datos en el recurso y otros hilos (consumidores) se encargan de eliminar y procesar esos datos.

2) El problema de los **Lectores-Escritores**. Permite modelar el acceso simultáneo de varios hilos a una base de datos, fichero u otro recurso, unos queriendo leer y otros escribir o modificar los datos.

7.5. El problema del interbloqueo (deadlock)

El **interbloqueo** o bloqueo mutuo (**deadlock**) consiste en que uno a más hilos se bloquean o esperan indefinidamente.

A dicha situación se llega:

- Porque cada hilo espera a que le llegue un aviso de otro hilo que nunca le llega.
- Porque todos los hilos, de forma circular, esperan para acceder a un mismo recurso.

El problema del bloqueo mutuo, en las aplicaciones concurrentes, se podrá dar fundamentalmente cuando un hilo entra en un bloque ***synchronized***, y a su vez llama a otro bloque ***synchronized***, o bien al utilizar clases de **java.util.concurrent** que llevan implícita la exclusión mutua.

Otro problema, menos frecuente, es la **inanición** (**starvation**), que consiste en que un hilo es desestimado para su ejecución. Se produce cuando un hilo no puede tener acceso regular a los recursos compartidos y no puede avanzar, quedando bloqueado. Esto puede ocurrir porque el hilo nunca es seleccionado para su procesamiento o bien porque otros hilos que compiten por el mismo recurso se lo impiden.

Está claro que los programas que desarrollemos deben estar exentos de estos problemas, por lo que habrá que ser cuidadosos en su diseño.

7.6. La clase Semaphore

La clase **Semaphore** del paquete **java.util.concurrent**, permite definir un semáforo para controlar el acceso a un recurso compartido. Para crear y usar un objeto **Semaphore** haremos lo siguiente:

1) Indicar al constructor **`Semaphore(int permisos)`** el total de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. Este valor coincide con el número de hilos que pueden acceder a la vez al recurso.
2) Indicar al semáforo mediante el método **`acquire()`** que queremos acceder al recurso, o bien mediante **`acquire(int permisosAdquirir)`** para indicar cuántos permisos se quieren consumir al mismo tiempo.
3) Indicar al semáforo mediante el método **`release()`** que libere el permiso, o bien mediante **`release(int permisosLiberar)`**, cuantos permisos se quieren liberar al mismo tiempo.
4) Hay otro constructor **`Semaphore (int permisos, boolean justo)`** que mediante el parámetro justo permite garantizar que el primer hilo en invocar **`adquire()`** será el primero en adquirir un permiso cuando sea liberado. Esto es, garantiza el orden de adquisición de permisos, según el orden en que se solicitan.

¿Desde dónde se deben invocar estos métodos? Esto dependerá del uso de **Semaphore**:

- Si se usa para proteger secciones críticas, la llamada a los métodos **`acquire()`** y **`release()`** se hará desde el recurso compartido o sección crítica, y el número de permisos pasado al constructor será 1.

- Si se usa para comunicar hilos, en este caso un hilo invocará al método **`acquire()`** y otro hilo invocará al método **`release()`** para así trabajar de manera coordinada. El número de permisos pasado al constructor coincidirá con el número máximo de hilos bloqueados en la cola o lista de espera para adquirir un permiso.

En el siguiente código tienes un ejemplo del uso de ***Semaphore*** para proteger secciones críticas o recursos compartidos. Es el ejemplo que vimos del acceso simultáneo de 4 terminales a un Servidor, pero resuelto ahora con la clase ***Semaphore*** en vez de con ***synchronized***.

```
import java.util.concurrent.Semaphore;

public class ServidorWeb { //clase que simula los accesos a un servidor
    private int cuenta;
    public ServidorWeb() {
        cuenta = 0;
    }
    public void incrementaCuenta() {
        //método sincronizado
        //muestra el hilo que entra en el Servidor
        System.out.println("hilo " + Thread.currentThread().getName()
                + "----- Entra en Servidor");
        //se incrementa la cuenta de accesos
        cuenta++;
        //muestra el número de accesos
        System.out.println(cuenta + " accesos");
    }
}

public class Hilo_Terminal extends Thread {

    private ServidorWeb servidor;
    private Semaphore semaforo;

    public Hilo_Terminal(ServidorWeb s, Semaphore se) {
        this.servidor = s;
        this.semaforo = se;
    }
```

```
    @Override
    public void run() {
        //la tarea del hilo es invocar a incrementaCuenta()
        // simulando un acceso al servidor

        for (int i = 1; i <= 10; i++) //se simulan 10 accesos al servidor
        {
            try {
                //en cada acceso se adquiere el recurso
                //y si está ocupado se bloquea
                semaforo.acquire();
            } catch (InterruptedException ex) {
            }
            //adquirido el recurso
            //invoca a este método para simular el acceso
            //al servidor incrementado la cuenta de accesos
            servidor.incrementaCuenta();
            //libera el recurso o permiso
            semaforo.release();
            yield();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        //semáforo para las secciones críticas de esta clase (permisos 1)
        Semaphore semaforo = new Semaphore(1);

        ServidorWeb servidor = new ServidorWeb();

        Hilo_Terminal hterminal1 = new Hilo_Terminal(servidor, semaforo);
        Hilo_Terminal hterminal2 = new Hilo_Terminal(servidor, semaforo);
        Hilo_Terminal hterminal3 = new Hilo_Terminal(servidor, semaforo);
        Hilo_Terminal hterminal4 = new Hilo_Terminal(servidor, semaforo);

        hterminal1.start();
        hterminal2.start();
        hterminal3.start();
        hterminal4.start();
    }
}
```

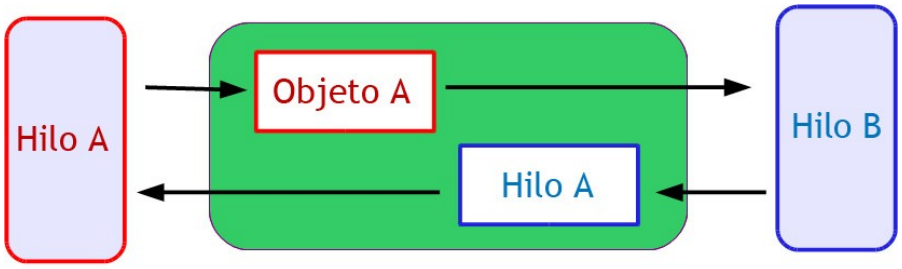
7.7. La clase Exchanger

La clase **Exchanger**, del paquete **java.util.concurrent**, establece un punto de sincronización donde se intercambian objetos entre dos hilos. La clase **Exchanger<V>** es genérica, lo que significa que tendrás que especificar en **<V>** el tipo de objeto a compartir entre los hilos.

Existen dos métodos definidos en esta clase:

- **`exchange(V x).`**
- **`exchange(V x, long timeout, TimeUnit unit).`**

Ambos métodos **`exchange()`** permiten intercambiar objetos entre dos hilos. El hilo que desea obtener la información, esperará realizando una llamada al método **`exchange()`** hasta que el otro hilo sitúe la información utilizando el mismo método, o hasta que pase un periodo de tiempo establecido mediante el parámetro *timeout* .



El funcionamiento, tal y como puedes apreciar en la imagen anterior, sería el siguiente:

1) Dos hilos (A y B) intercambiarán objetos del mismo tipo, objetoA y objetoB.
2) El hilo A invocará a **`exchange(objetoA)`** y el hilo B invocará a **`exchange(objetoB)`**.
3) El hilo que procese su llamada a **`exchange(objeto)`** en primer lugar, se bloqueará y quedará a la espera de que lo haga el segundo. Cuando eso ocurra y se libere el bloqueo sobre ambos hilos, la salida del método **`exchange(objetoA)`** proporcionará el objeto objetoB al hiloA, y la del método **`exchange(objetoB)`** el objeto objetoA al hiloB.

¿**Cuándo puede ser útil Exchanger**? Los intercambiadores se emplean típicamente cuando un hilo productor está rellenando una lista o búfer de datos, y otro hilo consumidor los está consumiendo.

De esta forma cuando el consumidor empieza a tratar la lista de datos entregados por el productor, el productor ya está produciendo una nueva lista. Precisamente esta es la principal utilidad de los intercambiadores: que la producción y el consumo de datos puedan ocurrir concurrentemente.

7.8. La clase CountDownLatch

La clase **`CountDownLatch`** del paquete **`java.util.concurrent`** es una utilidad de sincronización que permite que 1 o más **threads** esperen hasta que otros **threads** finalicen su trabajo.

El funcionamiento esquemático de **`CountDownLatch`** o "cuenta atrás de cierre" es el siguiente:

- Implementa un punto de espera que denominaremos "puerta de cierre", donde 1 o más hilos esperan a que otros finalicen su trabajo.
- Los hilos que deben finalizar su trabajo se controlan mediante un contador llamado "cuenta atrás".
- Cuando "cuenta atrás" llega a cero se reanudará el trabajo del hilo o hilos interrumpidos y puestos en espera.
- No será posible volver a utilizar la "cuenta atrás", es decir, no se puede reiniciar. Si fuera necesario reiniciar la "cuenta atrás" habrá que pensar en utilizar la clase **`CyclicBarrier`** (siguiente apartado).

Los aspectos más importantes al usar la clase **`CountDownLatch`** son los siguientes:

- ✔ Al constructor **`countDownLatch(int cuenta)`** se le indica, mediante el parámetro "cuenta", el total de hilos que deben completar su trabajo, que será el valor de la "cuenta atrás".
- ✔ El hilo en curso desde el que se invoca al método **`await()`** esperará en la "puerta de cierre" hasta que la "cuenta atrás" tome el valor cero. También se puede utilizar el método **`await(long tiempoespera, TimeUnit unit)`**, para indicar que la espera será hasta que la cuenta atrás llegue a cero o bien se sobrepase el tiempo de espera especificado mediante el parámetro **`tiempoespera`**.
- ✔ La "cuenta atrás" se irá decrementando mediante la invocación del método **`countDown()`**, y cuando ésta llega al valor cero se libera el hilo o hilos que estaban en espera, continuando su ejecución.
- ✔ No se puede reinicar o volver a utilizar la "cuenta atrás" una vez que ésta toma el valor cero. Si esto fuera necesario, entonces debemos pensar en utilizar la clase **`CyclicBarrier`**.
- ✔ El método **`getCount()`** obtiene el valor actual de la "cuenta atrás" y generalmente se utiliza durante las pruebas y depuración del programa.

En el siguiente enlace puedes ver un ejemplo de cómo utilizar **`CountDownLatch`** para sumar todos los elementos de una matriz. Cada fila de la matriz es sumada por un hilo. Cuando todos los hilos han finalizado su trabajo se ejecuta el procedimiento que realiza la suma global.

```
import java.util.concurrent.CountDownLatch;

/**************************************************************************
 * suma el total de 10 tandas de números dispuestos en una matriz. Para obtener
 * la suma de cada tanda, se lanza un hilo auxiliar controlado por una cuenta
 * atrás de cierre
 *
 * el propósito de la cuenta atrás de cierre es que el hilo que va a realizar
 * la suma total (en nuestro caso, el hilo principal), espere a que cada uno de
 * los 10 hilos auxiliares complete la suma de su tanda
 */

public class Main {
    private static int tabla[][] = {
        {1},
        {1, 1},
        {1, 2, 1},
        {1, 3, 3, 1},
        {1, 4, 6, 4, 1},
        {1, 5, 10, 10, 5, 1},
        {1, 6, 15, 20, 15, 6, 1},
        {1, 7, 21, 35, 35, 21, 7, 1},
        {1, 8, 28, 56, 70, 56, 28, 8, 1},
        {1, 9, 36, 84, 126, 126, 84, 36, 9, 1}};
    //array para guardar la suma de los elementos de cada tanda o fila
```

```
        private static int resultadoTanda[];

    /**************************************************************************
     * clase que define el hilo auxiliar, cuyo método run() se encarga de sumar
     * los elementos de la tanda de números recibida por su constructor
     *
     * el constructor recibe también un objeto CountDownLatch de control
     */
    private static class SumaTanda extends Thread {

        //índice de la tanda
                    int t;
                    //objeto de control
        CountDownLatch cdl;

        SumaTanda(CountDownLatch cdl, int t) {
            this.cdl = cdl;
            this.t = t;
        }

        /**************************************************************************
         * método run que suma los elementos de la tanda recibida por el
         * constructor
         *
         * cuando finaliza esta suma y se almacena el valor, se llama al
         * método countDown() de la barrera
         */
        @Override
        public void run() {
            int elementos = tabla[t].length;
            //número de elementos de la tanda

            int sumaTanda = 0;
            //acumulador parcial

            for (int i = 0; i < elementos; i++) {
                sumaTanda += tabla[t][i];
                //agrega el elemento de la tanda al parcial
            }

            resultadoTanda[t] = sumaTanda;
            //guarda en resultadoTanda la suma de la tanda t

            //muestra un mensaje
            System.out.println("La suma de los elementos de la tanda "
                    + t + " es: " + sumaTanda);

            //finalizada la suma de los elementos de la tanda y almacenado
            //el valor, el hilo llama al método countDown() de la barrera

            try {
                cdl.countDown();
                //un elemento menos en la cuenta atrás
            } catch (Exception ex) {
                //no hace nada
            }
        }
    }

    /**************************************************************************
     * realiza la suma total de los elementos de la matriz, cuando el objeto
     * CountDownLatch que controla los hilos axiliares lo permite
```

```
     */
    public static void main(String args[]) {

        final int ntandas = tabla.length;
        //número total de tandas (10, en este ejemplo)

        int sumaTotal = 0;
        //acumulador total

        resultadoTanda = new int[ntandas];
        //dimensiona a 10 el vector que almacenará las sumas de los elementos
        //de cada tanda

        CountDownLatch cdl = new CountDownLatch(ntandas);
        //objeto tipo CountDownLatch para 10 hilos (uno para cada tanda de
        //números). Este objeto pondrá en espera cada hilo desde donde se
        //invoque su método await() (en nuestro caso, sólo el hilo principal),
        //hasta que cada uno de los 10 hilos que controla realice una llamada
        //su método countDown()

        //mensaje de espera
        System.out.println("Obteniendo la suma de los elementos de "
                + "cada tanda...\n");

        //lanza un hilo por cada tanda de elementos (10 hilos)
        for (int i = 0; i < ntandas; i++) {
            new SumaTanda(cdl, i).start();
            //cada nuevo hilo recibe el objeto CountDownLatch de control, y el
            //índice de la tanda cuyos elementos debe sumar
        }

        try {
            cdl.await();
            //coloca el hilo desde donde se ejecuta esta llamada al método
            //await() (el hilo principal, en nuestro caso), a la
            //espera de que cada hilo controlado por la cuenta atrás
            //llame al método countDown().
            //Ningún hilo controlado llamará a este método hasta que no haya
            //completado la suma de su tanda
        } catch (Exception ex) {
            //no hace nada
        }

        //cuando se reanuda el hilo principal, todos los hilos controlados
        //por la cuenta atrás han terminado de sumar su tanda. Por tanto, es el
        //momento de realizar la suma total

        for (int i = 0; i < ntandas; i++) {
            sumaTotal += resultadoTanda[i];
            //agrega el resultadoTanda al total
        }

        //imprime la suma total
        System.out.println("\nTodas la tandas han sido "
                + "sumadas. Total: " + sumaTotal);
    }
}
```

7.9. La clase CyclicBarrier

La clase **`CyclicBarrier`** del paquete **`java.util.concurrent`** es una utilidad de sincronización que permite que uno o más threads se esperen hasta que todos ellos finalicen su trabajo.

El funcionamiento esquemático de **`CyclicBarrier`** o "barrera cíclica" es el siguiente:

- Implementa un punto de espera que llamaremos "barrera", donde cierto número de hilos esperan a que todos ellos finalicen su trabajo.
- Finalizado el trabajo de estos hilos, se dispara la ejecución de una determinada acción o bien el hilo interrumpido continúa su trabajo.
- La barrera se llama cíclica, porque se puede volver a utilizar después de que los hilos en espera han sido liberados tras finalizar todos su trabajo, y también se puede reiniciar.

Los aspectos más importantes al usar la clase **`CyclicBarrier`** son los siguientes:

- ✔ Indicar al constructor **`CyclicBarrier(int hilosAcceden)`** el total de hilos que van a usar la barrera mediante el parámetro **`hilosAcceden`**. Este número sirve para disparar la barrera.
- ✔ La barrera se dispara cuando llega el último hilo.
- ✔ Cuando se dispara la barrera, dependiendo del constructor, CyclicBarrier(int hilosAcceden) o **`CyclicBarrier(int hilosAcceden, Runnable acciónBarrera)`** se lanzará o no una acción, y entonces se liberan los hilos de la barrera. Esa acción puede ser realizada mediante cualquier objeto que implemente **`Runnable`**.
- ✔ El método principal de esta clase es **`await()`** que se utiliza para indicar que el hilo en curso ha concluido su trabajo y queda a la espera de que lo hagan los demás.

Otros métodos de esta clase que puedes utilizar son:

- El método **`await(long tiempoespera, TimeUnit unit)`** funciona como el anterior, pero en este caso el hilo espera en la barrera hasta que los demás finalicen su trabajo o se supere el **`tiempoespera`**.
- El método **`reset()`** permite reiniciar la barrera a su estado inicial.
- El método **`getNumber Waiting()`** devuelve el número de hilos que están en espera en la barrera.
- El método **`getParties()`** devuelve el número de hilos requeridos para esa barrera

Veamos un ejemplo parecido al anterior, pero ahora resuelto con **`CyclicBarrier`**. Cada fila de la matriz ahora representa los valores recaudados por un cobrador. Cada fila es sumada por un hilo. Cuando 5 de estos hilos finalizan su trabajo, se dispara un objeto que implementa **`Runnable`** para obtener la suma recaudada hasta el momento. Como la matriz del ejemplo tiene 10 filas, la suma de sus elementos se hará mediante una barrera de 5 hilos y que se utilizará por tanto de forma cíclica dos veces.

```
import java.util.concurrent.CyclicBarrier;

/*************************************************************************
 * suma el total de 10 tandas de números dispuestos en una matriz. Para obtener
 * la suma de cada tanda o fila, se lanza un hilo controlado por una barrera
 *  CyclicBarrier de 5 hilos. Cada tanda de la matriz representa los valores
 * recaudados por un cobrador.
 *
 * el propósito de la barrera es desencadenar un procedimiento que suma los
 * totales de cada tanda. En este ejemplo, supondremos que interesa ir
 * acumulando los valores recaudados cada 5 cobradores, 5 tandas
 *
 * como tenemos 10 hilos auxiliares, la barrera desencadenará este
 * procedimiento 2 veces, lo que impica que se ejecutará de
 * forma cíclica
 *
```

```
 * la primera vez que lo hace, sólo hay 5 hilos finalizados. Luego la suma
 * obtenida, será sólo una parte del total buscado (cada hilo no finalizado
 * contribuye con un 0)
 *
 * sin embargo, la segunda vez todos los hilos habrán terminado. En
 * este caso, la suma obtenida será el total buscado
 */

public class Main {

    //matriz de 10 tandas o filas de números
    private static int tabla[][] = {
        {1},
        {1, 1},
        {1, 2, 1},
        {1, 3, 3, 1},
        {1, 4, 6, 4, 1},
        {1, 5, 10, 10, 5, 1},
        {1, 6, 15, 20, 15, 6, 1},
        {1, 7, 21, 35, 35, 21, 7, 1},
        {1, 8, 28, 56, 70, 56, 28, 8, 1},
        {1, 9, 36, 84, 126, 126, 84, 36, 9, 1}};
    private static int resultadoTanda[];
    //resultadoTanda de la suma de los elementos de cada tanda

    /**************************************************************************
     * clase que define el hilo auxiliar, cuyo método run() se encarga de sumar
     * los elementos de la tanda de números recibida por su constructor
     *
     * el constructor recibe también un objeto CyclicBarrier de control
     */
    private static class SumaTanda extends Thread {
    //clase que implementa un hilo
        int t;
        //índice de la tanda (en este caso, un entero de 0 a 4)
        CyclicBarrier barreraCiclica;
        //barrera cíclica de control

        /**************************************************************************
         * constructor
         */
        SumaTanda(CyclicBarrier barreraCiclica, int t) {
            this.barreraCiclica = barreraCiclica;
            this.t = t;
        }

        /**************************************************************************
         * método run que suma los elementos de la tanda recibida por el
         * constructor
         *
         * cuando finaliza esta suma y se almacena el valor, se incrementa
         * en una unidad el número de hilos en espera dentro de la barrera
         *
         * cuando ese número de elementos en espera sea el indicado más abajo
         * por el constructor de la barrera (5 en este caso), se desencadenará
         * el procedimiento que obtiene la suma de todos ellos
         */
        @Override
        public void run() {
            //comportamiento del hilo
            int elementos = tabla[t].length;
            //número de elementos de la tanda
```

```
            int sumaParcial = 0;
            //acumulador parcial

            for (int i = 0; i < elementos; i++) {
                sumaParcial += tabla[t][i];
                //agrega el elemento de la tanda al parcial
            }

            resultadoTanda[t] = sumaParcial;
            //guarda el resultadoTanda de la suma de la tanda

            //muestra un mensaje en consola
            System.out.println("La suma de los elementos de la tanda "
                    + t + " es: " + sumaParcial);

            try {
                barreraCiclica.await();
                //un hilo más que ha completado su trabajo y por tanto en espera
                //dentro de la barrera
            } catch (Exception ex) {
                //no hace nada
            }
        }
    }

    /**************************************************************************
     * realiza la suma total de los elementos de la matriz, mediante el método
     * sumaParcial de un objeto CyclicBarrier
     */
    public static void main(String args[]) {

        final int ntandas = tabla.length;
        //número total de tandas (10, en este caso)

        resultadoTanda = new int[ntandas];
        //vector de sumas de cada tanda

        /**********************************************************************
         * procedimiento de suma parcial que se ejecutará cada vez que se
         * complete la barrera, implementado mediante la clase Runnable
         */
        Runnable sumaParcial = new Runnable() {

            int totalAcumulado;
            //acumulador total

            //suma los resultados de cada tanda (las que no hayan terminado
            //sumaran 0)
            public void run() {
                totalAcumulado = 0;
                //reinicia el total

                for (int i = 0; i < ntandas; i++) {
                    totalAcumulado += resultadoTanda[i];
                    //agrega el resultadoTanda al parcial
                }

                //imprime la suma total
                System.out.println("\nBarrera completada. Total acumulado: "
                        + totalAcumulado + "\n");
            }
```

```
        };

        CyclicBarrier barreraCiclica = new CyclicBarrier(5, sumaParcial);
        //crea una Barrera de Control que desencadenará un procedimiento
        //sumaParcial, cuando el número número de elemento en espera dentro de
        //ella sea 5. Este procedimiento será disparado por el último hilo desde
        //el que se invoque el método await() de la barrera

        //lanza un nuevo hilo para cada tanda
        for (int i = 0; i < ntandas; i++) {
            new SumaTanda(barreraCiclica, i).start();
            //cada nuevo hilo recibe la Barrera Cíclica de control, y el
            //índice de la tanda sobre la que actuará
        }
    }
}
```

Profundicemos más en algunas de las **diferencias** semánticas entre las 2 últimas clases:

Como se indica en las definiciones, **CyclicBarrier** permite que varios hilos se esperen entre sí, mientras que **CountDownLatch** permite que uno o más hilos esperen a que se completen varias tareas.

*En resumen, **CyclicBarrier** mantiene un **recuento** de **subprocesos**, mientras que **CountDownLatch** mantiene un **recuento** de **tareas**.*

8. Aplicaciones multihilo

Una aplicación multihilo debe reunir las siguientes propiedades:

- ✔ **Seguridad**. La aplicación no llegará a un estado inconsistente por un mal uso de los recursos compartidos. Esto implicará sincronizar hilos asegurando la exclusión mutua.
- ✔ **Viveza**. La aplicación no se bloqueará o provocará que un hilo no se pueda ejecutar. Esto implicará un comportamiento no egoísta de los hilos y ausencia de interbloqueos e inanición.

La **corrección** de la aplicación se mide en función de las propiedades anteriores, pudiendo tener:

- ➢ Corrección **parcial**. Se cumple la propiedad de seguridad. El programa termina y el resultado es el deseado.
- ➢ Corrección **total**. Se cumplen las propiedades de seguridad y viveza. El programa termina y el resultado es el correcto.

Por tanto, al desarrollar una aplicación multihilo habrá que **tener en cuenta los siguientes aspectos**:

- ➢ La situación de los hilos en la aplicación: hilos independientes o colaborando/compitiendo.
  - Independientes. No será necesario sincronizar y/o comunicar los hilos.
  - Colaborando y/o compitiendo. Será necesario sincronizar y/o comunicar los hilos, evitando interbloqueos y esperas indefinidas.
- ➢ Gestionar las prioridades, de manera que los hilos más importantes se ejecuten antes.
- ➢ No todos los Sistemas Operativos implementan time-slicing.
- ➢ La ejecución de hilos es no-determinística.

Por lo general, las aplicaciones multihilo son más difíciles de desarrollar y complicadas de depurar que una aplicación secuencial o de un solo hilo; pero si utilizamos las librerías que aporta el lenguaje de programación podemos obtener algunas ventajas:

- ➢ **Facilitar la programación**. Requiere menos esfuerzo usar una clase estándar que desarrollarla para realizar la

misma tarea.

- **Mayor rendimiento**. Los algoritmos utilizados han sido desarrollados por expertos en concurrencia y rendimiento.
- **Mayor fiabilidad**. Usar librerías o bibliotecas estándar, que han sido diseñadas para evitar interbloqueos (deadlocks), cambios de contexto innecesarios o condiciones de carrera, nos permiten garantizar un mínimo de calidad en nuestro software.
- **Menor mantenimiento**. El código que generemos será más legible y fácil de actualizar.
- **Mayor productividad**. El uso de una API estándar permite mejor coordinación entre desarrolladores y reduce el tiempo de aprendizaje.

Teniendo en cuenta esto último, cuando vayas a desarrollar una aplicación multihilo debes hacer uso de las utilidades que ofrece el propio lenguaje. Esto facilitará la puesta a punto del programa y su depuración.

8.1. Otras utilidades de concurrencia

Además de las utilidades de sincronización que hemos visto en apartados anteriores, el paquete **`java.util.concurrent`** incluye estas otras utilidades de concurrencia:

- La interfaz **`Executor`**
- Colecciones.
- La clase **`Locks`**
- Variables atómicas

El programador de tareas **`Executor`** es una interfaz que permite:

- Realizar la ejecución de tareas en un único hilo en segundo plano (como eventos Swing), en un hilo nuevo, o en un pool de hilos
- Diseñar políticas propias de ejecución y añadirlas a **`Executor`**.
- Ejecutar tareas mediante el método **`execute()`**. Estas tareas tienen que implementar la interfaz **`Runnable`**.
- Hacer uso de diferentes implementaciones de **`Executor`**, como **`ExecutorService`**.

Entre las **colecciones** hay que destacar:

- La interfaz **`Queque`**. Es una colección diseñada para almacenar elementos antes de procesarlos, ofreciendo diferentes operaciones como inserción, extracción e inspección.
- La interfaz **`BlockingQueque`**, diseñada para colas de tipo productor/consumidor, y que son thread-safe (aseguran un funcionamiento correcto de los accesos simultáneos multihilo a recursos compartidos). Son capaces de esperar mientras no haya elementos almacenados en la cola.
- Implementaciones concurrentes de **`Map`** y **`List`**.

La **clase** de **bloqueos**, **`java.util.concurrent.locks`**, proporciona diferentes implementaciones y diversos tipos de bloqueos y desbloqueos entre métodos. Su funcionalidad es equivalente a **`Synchronized`**, pero proporciona métodos que hacen más fácil el uso de bloqueos y condiciones. Entre ellos.

- El métdo **`newCondition()`**, que permite tener un mayor control sobre el bloqueo y genera un objeto del tipo **`Condition`** asociado al bloqueo. Así el método **`await()`** indica cuándo deseamos esperar, y el método **`signal()`** permite indicar si una condición del bloqueo se activa, para finalizar la espera.
- La implementación **`ReentranLock`**, permite realizar exclusión mutua utilizando monitores. El método **`lock()`** indica que deseamos utilizar el recurso compartido, y el método **`unlock()`** indica que hemos terminado de utilizarlo.

Las **variables atómicas** incluidas en las utilidades de concurrencia clase j**`ava.util.concurrent.atomic`**, permiten definir recursos compartidos, sin la necesidad de proteger dichos recursos de forma explícita, ya que ellas internamente realizan dichas labores de protección.

Si desarrollamos aplicaciones multihilo más complejas, por ejemplo para plataformas multiprocesador y sistemas con multi-núcleo (multi-core) que requieren un alto nivel de concurrencia, será muy conveniente hacer uso de todas estas utilidades.

8.2. La interfaz Executor y los pools de hilos

Cuando trabajamos con **aplicaciones tipo servidor**, éstas tienen que atender un número masivo y concurrente de peticiones de usuario, en forma de tareas que deben ser procesadas lo antes posible. Al principio de la unidad ya te indicamos mediante un ejemplo la conveniencia de utilizar hilos en estas aplicaciones, pero si ejecutamos cada tarea en un hilo distinto, se pueden llegar a crear tantos hilos que el incremento de recursos utilizados puede comprometer la estabilidad del sistema. Los pools de hilos ofrecen una solución a este problema.

*"Un **pool de hilos** (thread pools) es un contenedor dentro del cual se crean y se inician un número limitado de hilos, para ejecutar todas las tareas de una lista."*

Para declarar un pool, lo más habitual es hacerlo como un objeto del tipo **ExecutorService** utilizando alguno de los siguientes métodos de la clase estática **Executors**:

- **`newFixedThreadPool(int numeroHilos)`**: crea un pool con el número de hilos indicado. Dichos hilos son reutilizados cíclicamente hasta terminar con las tareas de la cola o lista.

- **`newCachedThreadPool()`**: crea un pool que va creando hilos conforme se van necesitando, pero que puede reutilizar los ya concluidos para no tener que crear demasiados. Los hilos que llevan mucho tiempo inactivos son terminados automáticamente por el pool.
- **`newSingleThreadExecutor()`**: crea un pool de un solo hilo. La ventaja que ofrece este esquema es que si ocurre una excepción durante la ejecución de una tarea, no se detiene la ejecución de las siguientes.

- **`newScheduledExecutor()`** : crea un pool que va a ejecutar tareas programadas cada cierto tiempo, ya sea una sola vez o de manera repetitiva. Es parecido a un objeto Timer, pero con la diferencia de que puede tener varios threads que irán realizando las tareas programadas conforme se desocupen.

Los objetos de tipo **`ExecutorService`** implementan la interfaz **`Executor`**. Esta interfaz define el método **`execute(Runnable)`**, al que hay que llamar una vez por cada tarea que deba ser ejecutada por el pool (la tarea se pasa como argumento del método).

La interface **`ExecutorService`** proporciona una serie de métodos para el control de la ejecución de las tareas, entre ellos el método **`shutdown()`**, para indicarle al pool que los hilos no se van a reutilizar para nuevas tareas y deben morir cuando finalicen su trabajo.

En el siguiente ejemplo se define una clase que implementa **`Runnable`** cuya tarea es generar e imprimir 10 números aleatorios. Se creará un pool de 2 hilos capaz de realizar 30 de esas tareas.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.Random;

public class NumerosAleatorios implements Runnable {
    /*****************************************************************
     * compone una cadena de diez números aleatorios menores que 50, separados
     * por ','
     */
    public void run() {

        String strReturn = "";
        Random random = new Random();
```

```
        for (int i = 0; i <10; i++) {
            strReturn += random.nextInt(50) + ", ";
            Thread.yield();
        }

        System.out.println("Números aleatorio obtenidos por "
                + Thread.currentThread().getName() + ": " + strReturn);
    }
}

public class Main {
    /*************************************************************************
     * ejecuta ocho veces la tarea NumerosAleatorios que imprime diez números
     * aleatorios menores que cincuenta, mediante un pool de tan sólo dos hilos
     */
    public static void main(String[] args) {

        //define un pool fijo de dos hilos
        ExecutorService executor = Executors.newFixedThreadPool(2);

        //pasa 30 tareas NumerosAleatorios al pool de 2 hilos
        for (int i = 1; i <= 30; i++) {
            executor.submit(new NumerosAleatorios());
        }

        //ordena la destrucción de los hilos del pool al finalizar tareas
        executor.shutdown();
    }
}
```

8.3. Gestión de excepciones

Para gestionar las excepciones de una aplicación multihilo puedes utilizar el método **`uncaughtExceptionHandler()`** de la clase **thread**, que permite definir un manejador de excepciones.

Para crear un manejador de excepciones haremos lo siguiente:

- Crear una clase que implemente la interfaz **`thread`**.**`UncaughtExceptionHandler`**.
- Implementar el método **`uncaughtException()`**.

Por ejemplo, podemos crear un manejador de excepciones que utilizarán todos los hilos de una misma aplicación de la siguiente forma:

```
public class ManejadorExcepciones implements Thread.
//manejador de excepciones para toda la aplicación
   UncaughtExceptionHandler{
    //implementa el método uncaughtException()
    public void uncaughtException(Thread t, Throwable e){
        System.out.printf("Thread que lanzó la excepción: %s \n", t.getName());
        //muestra en consola el hilo que produce la exceción
        e.printStackTrace();
        //muestra en consola la pila de llamadas
    }
}
```

[El manejador sólo mostrará qué hilo ha producido la excepción y la pila de llamadas de la excepción]

8.4. Depuración y documentación

Dos tareas muy importantes cuando desarrollamos software de calidad son la depuración y documentación de nuestras aplicaciones.

- ✔ Mediante la **depuración** trataremos de corregir fallos y errores de funcionamiento del programa.
- ✔ Mediante la **documentación** interna aportaremos legibilidad a nuestros programas.

La **depuración** de aplicaciones multihilo es una tarea difícil debido a que:

- La ejecución de los hilos tiene un comportamiento no **determinístico**.
- Hay que controlar **varios flujos** de ejecución.
- Aparecen nuevos **errores potenciales** debidos a la **compartición** de recursos entre varios hilos:
- **Errores** porque no se cumple la **exclusión mutua**.
- **Errores** porque se produce **interbloqueo**.

Podemos realizar seguimientos de la pila de Java tanto estáticos como dinámicos, utilizando los siguientes métodos de la clase **thread**:

- `dumpStack()`. Muestra una traza de la pila del hilo (thread) en curso.
- `getAllStackTraces()`. Devuelve un Map de todos los hilos vivos en la aplicación. (Map es la interfaz hacia objetos `StackTraceElement`, que contiene el nombre del fichero, el número de línea, y el nombre de la clase y el método de la línea de código que se está ejecutando).
- `getStackTrace()`. Devuelve el seguimiento de la pila de un hilo en una aplicación.

Tanto `getAllStackTraces()` como `getStackTrace()` permiten grabar los datos del seguimiento de pila en un log.

A la hora de **documentar** una aplicación multihilo no debemos escatimar en comentarios. Si son importantes en cualquier aplicación, con más motivo lo serán en una aplicación multihilo debido a su mayor complejidad.

Para documentar nuestra aplicación Java, utilizaremos el generador **JavaDoc**, que de forma automática genera la documentación de la aplicación a partir del código fuente. Como ya debes saber del módulo de "Programación", este sistema consiste en incluir comentarios en el código, utilizando las etiquetas /** y */, que después pueden procesarse y generar un conjunto de páginas navegables HTML.